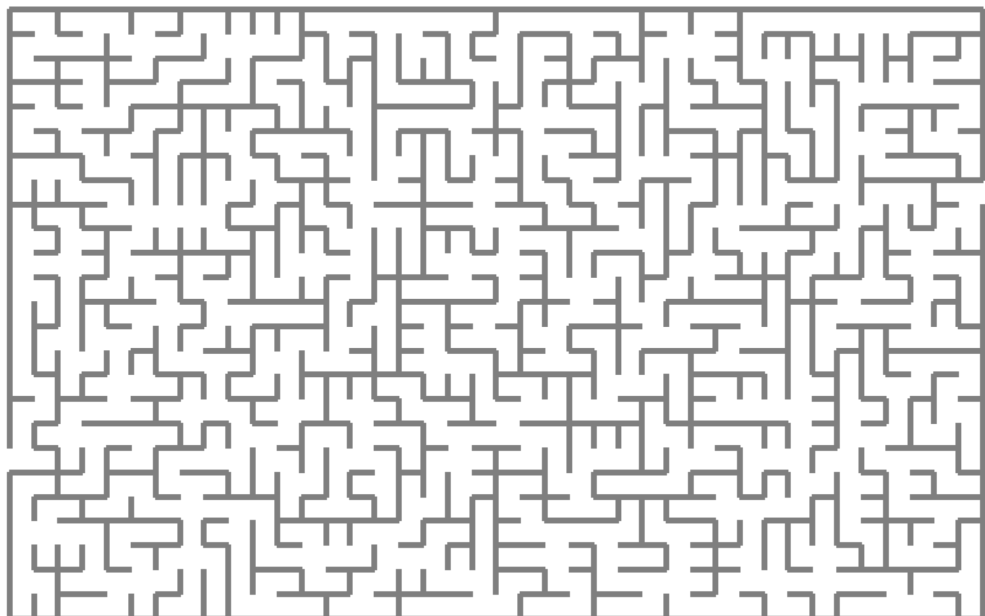
Probabilistic Testing for Weak Memory Concurrency

Version of October 6, 2022



Mingyu Gao

Probabilistic Testing for Weak Memory Concurrency

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Mingyu Gao
born in Nanjing, Jiangsu Province, China

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

© 2022 Mingyu Gao.

Probabilistic Testing for Weak Memory Concurrency

Author: Mingyu Gao
Student id: 5216281
Email: m.gao-2@student.tudelft.nl

Abstract

The Probabilistic Concurrency Testing (PCT) algorithm provides theoretical guarantees for the probability of detecting concurrency bugs in a sequential consistency memory model, but its theoretical guarantees do not apply to weak memory concurrency. The weak memory concurrency refers to the modern compiler's optimization that relaxes the sequential consistency requirements. The PCT approach is based on the sequential consistency interleaving semantics, 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.

Based on the PCT algorithm transforming the concurrency bug to the ordering constraints(bug depth), this thesis presents Probabilistic Concurrency Testing for Weak Memory (PCTWM) to capture the concurrency behavior in weak memory programs, further revising the notion of the bug depth to the constraints of communication relations between events.

We implement both the PCT and PCTWM algorithms on top of the state-of-the-art weak memory testing tool - C11Tester. We empirically evaluate the bug detection ability of the PCTWM on a set of well-known weak memory program benchmarks. Our results show that PCTWM can detect concurrency bugs more frequently than C11Tester.

Key Words : concurrency bug, probabilistic testing, weak memory model

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. B. Kulahcioglu Ozkan, Dr. S. Chakraborty, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. van Genderen, Faculty EEMCS, TU Delft

Preface

Without the assistance and support of the generous people in my life, it would not have been feasible for me to complete this master's thesis. I will only be able to identify a few of those individuals in this article due to space limitations.

It would not have been possible to finish this master thesis without the help, support, and patience of my principle supervisors - Dr. Burcu Ozkan and Dr. Soham Chakraborty. Not to mention their guidance and outstanding understanding of weak memory and concurrency testing. I am indebted to them in an extraordinary degree because they have provided me with invaluable assistance in the form of sound advice, unwavering support, and enjoyable company on both an academic and a personal level.

I would want to express my gratitude to my fiancée Yuheng for always providing me with personal support as well as a high level of care. Even though he is currently in China and the Covid-19 has kept us apart for the past two years, he never misses an opportunity to get in touch with me and never fails to offer his support and encouragement. Because my parents have consistently shown their unwavering support for me, a simple word of gratitude on my part is not enough to adequately reflect my gratitude to them.

I would like to express my gratitude for the financial, academic, and technical support that was provided by the faculty and employees of the Electrical Engineering, Mathematics, and Computer Science department at Delft University of Technology, particularly my thesis advisor Prof. Arie van Deursen that provided the kind support for this research. The resources provided by the university's libraries have been of critical importance. I would also like to express my gratitude to the Department of the thesis as well as the board of examiners for all of the help that they have provided since the beginning of my thesis project in 2021.

Last but not the least, I would like to thank my friends in the Netherlands, for giving me happiness and encouragement throughout the whole year.

Mingyu Gao
Delft, the Netherlands
October 6, 2022

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

2 Background 3
2.1 Weak Memory Concurrency 3
2.2 PCT v.s. Naive Random Testing 6
2.3 C11Tester - Automatic Testing Tool 6

3 Overview 7
3.1 C11Tester 7
3.2 A Naive Application of PCT to Weak Memory 10
3.3 Revising Concurrency Bug Depth 10
3.4 PCTWM: PCT for Weak Memory 11

4 Weak Memory Concurrency Model 13
4.1 Event 13
4.2 Relation 13
4.3 Execution 14
4.4 Example 15

5 Algorithm and Examples 17
5.1 PCT Algorithm and Theoretical Guarantee 17
5.2 PCTWM Algorithm 23

6 Implementation 35
6.1 C11Tester Implementation and Plugins for Algorithms 35

6.2 PCT and PCTWM: Parameters 37
6.3 PCT Implementation 38
6.4 PCTWM Implementation 40

7 Experiment and Evaluation **45**
7.1 Benchmarks 45
7.2 RQ1. Bug Detection Ability with Estimated Parameters 48
7.3 RQ2. Bug Detection Ability Varying Bug Depth and History 48
7.4 RQ3. Bug Detection Ability Comparison: C11Tester v.s. PCT v.s. PCTWM 50
7.5 RQ4: PCTWM vs PCT: Bug Detection Performance 53
7.6 RQ5: Does or why PCTWM cause the overhead in the C11Tester? 55
7.7 RQ 6. Parameter's effect on PCTWM algorithm 56

8 Related Work **59**
8.1 Concurrency and consistency 59
8.2 Concurrency Bugs Types 59
8.3 Concurrency testing 60
8.4 Techniques for Detecting Concurrency Bugs in the Weak Memory Model . 60

9 Conclusions and Future Work **63**
9.1 Summary 63
9.2 Future work 64
9.3 Self-Reflection 64

Bibliography **65**

A Glossary **75**
A.1 Experiment Set Up 75

B Requirements and Guidelines **77**
B.1 Requirements 77
B.2 Guidelines 78

List of Figures

2.1 Concurrent Program Examples 4
2.2 Concurrent Program Example - SB in Weak Memory Model 4
2.3 Examples of memory order and synchronization in C/C++11 semantics. 5

3.1 The Mechanism of C11Tester. C11Tester replaces *pthread* library. 7
3.2 The workflow of C11Tester. Using **random** strategy to pick thread and read-from event when exploring the execution. 8
3.3 The Step-by-step example of the randomized strategy in C11Tester. The dot-line box is the available thread or action set offered by C11Tester each step. . . 9

4.1 Relations in Program SB 2.1a. 15
4.2 Concurrent Program Example - **SB** with different memory order types to illustrate C/C++ Concurrency. *mo* is the abbreviation of the memory order of actions. Arrow *sb* represents the 'sequence-before' relation. Arrow rf represents the 'read-from' relation. Arrow *sw* means the *RELEASE* and *ACQUIRE* actions are synchronized. Arrow *SC* is the total order for *SC* actions. 16

5.1 Concurrent Program Examples 18
5.2 A concurrent program example: P3. Bug depth $d = 3$. 19
5.3 MP1: execution a = 1, b = 1 with views and bags. 31
5.4 Three test executions for the Program MP2 32

6.1 PCT and PCTWM implementation on C11Tester. Above the dotted line is the thread scheduling level. Below the dotted line is the communication relation(read-from value selection) value. 37

7.1 Bug Hitting Rate - Varying h in PCTWM 51
7.2 Bug Hitting Rate for All Nine Benchmarks 51
7.3 Benchmark:Dekkfer-fences 52
7.4 Bug Hitting Rate - Inserting Relaxed Writes 54
7.5 Bug Hitting Rate - Varying d in PCTWM 57

Chapter 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 [36, 35, 52, 14, 28, 11] to provide platform-independent abstractions on the hardware and processors. These concurrency primitives empower programmers to achieve greater performance from the architectures, 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 which 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 concurrency. Unexpected interleavings of concurrent threads in crucial settings might lead to the system's failure, entering undefined states with catastrophic repercussions.

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 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 [21] is a randomized 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. Hence, 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 concurrency?* We investigate this question in this thesis project 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*, PCTWM samples a test execution with *d communication relations* between the concurrent program events. Roughly, the bug depth of a weak memory program execution represents the number of execution points in which the variable values visible to a thread are communicated to another thread regardless of how the previous operations in these threads are scheduled in the execution.

In this thesis project, we implemented both the PCT and the PCTWM algorithm on top of C11tester [51], 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. Chapter 2 provides the required background on weak memory concurrency and PCT. Chapter 3 presents an overview of our approach. Chapter 4 discusses the axiomatic model of weak memory concurrency model which focus in this work. Chapter 5 presents the PCT and PCTWM algorithm. Chapter 6 explains how we utilize the interface in C11Tester and implement the PCT and PCTWM algorithm. Chapter 7 provides the details of our experimental evaluation and results. Finally, the related workChapter 8 and conclusionChapter 9 are discussed.

Chapter 2

Background

In this chapter, we discuss the background knowledge for the thesis topic. We first illustrate the origin of the weak memory model, especially the C/C++11 weak memory model. And then we point out the concurrency problem brought on by this weak memory model. To tackle this problem, we list the normal approaches from two angles - randomized testing and concurrency testing, which serve as the foundation for our algorithm - PCTWM.

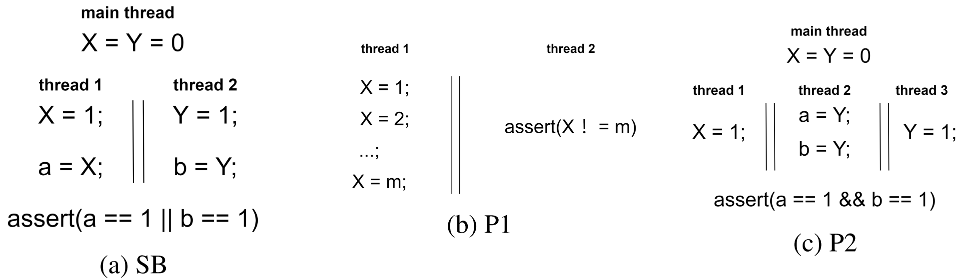
2.1 Weak Memory Concurrency

The memory consistency model formally specifies how the memory model will appear to the programmer. Modern memory model mostly adopts the shared memory model, which can be classified into two kinds - strong(SC) and weak(relaxed). Shared memory concurrency is a dominant programming paradigm where threads communicate through shared memory accesses.

Strong Memory Model The strong memory model, also called the sequential consistency(SC) model, is proposed by Lamport[43]. Shared memory concurrent programs are usually explained by sequential consistency (SC) [43] where shared memory accesses in each thread execute in syntactic order, and threads interleave arbitrarily. In the strong memory(SC) model, a read operation should return the value of the 'last' write operation to the same memory location. The definition of 'last' is the latest write action of the global *program order*, even though each thread contains the respective 'last' write on each location.

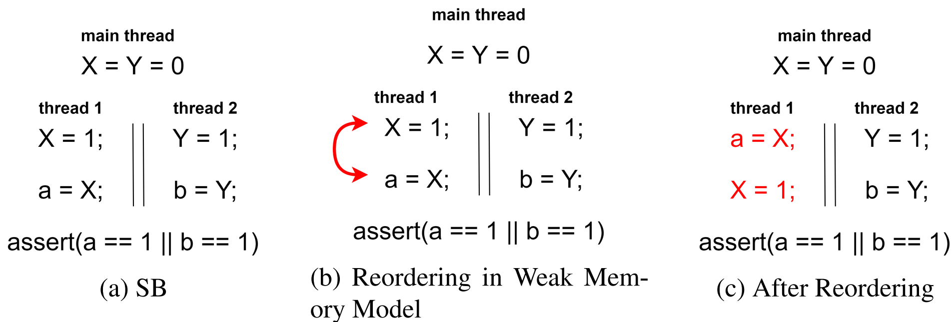
Weak Memory Model These year, modern processors tend to use the weak memory model(i.e., relaxed memory model, shared memory model). Because weak memory model can enhance the performance and the efficiency of computing.

However, concurrent systems usually exhibit additional program behaviors which cannot be explained by interleaving execution or sequential consistency. For example, when a concurrent program is running in a weak memory model, the write and read operations are uncertain since the operations reside on different processors[3]. As there in no global linear time in the weak memory model, the constraint on reads cannot be simply that they



(a) SB (b) P1 (c) P2

Figure 2.1: Concurrent Program Examples



(a) SB (b) Reordering in Weak Memory Model (c) After Reordering

Figure 2.2: Concurrent Program Example - SB in Weak Memory Model

read from the ‘most recent’ write.These additional non-SC executions are known as weak memory behaviors, and these concurrency models are known as weak memory concurrency.

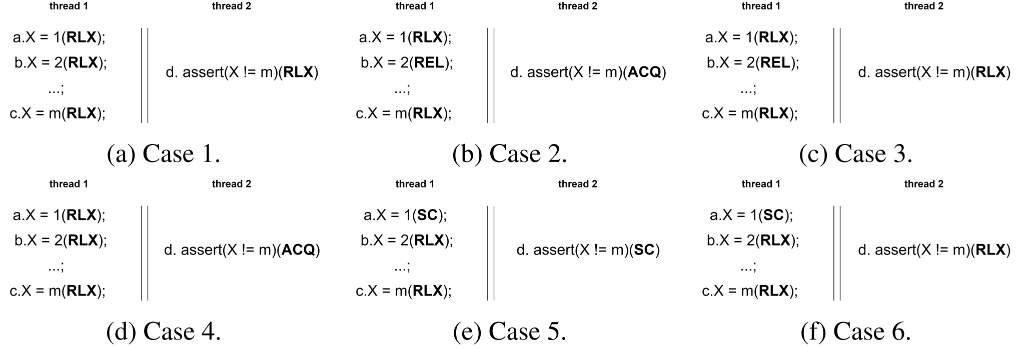
Take Program SB 2.2 as an example to show why the weak memory model is error-prone. In the SC model, the assertion cannot be met. Because in the SC model, the action writing 1 to X and Y happens before the read actions, and the read action reads from the latest value of X and Y.

However, various concurrency models such as x86 [64], Arm [5] architectures allow the non-SC outcome $a = b = 0$ that violates the assertion. The effect of these concurrency models can be simply understood as the reordering of instructions, as shown in the Figure 2.2b. This reordering results in a reading from the event $X = 0$ and then hits the assertion.

2.1.1 C/C++11 Weak Memory Model

C/C++11 Memory Model To program these weak-memory architectures, programming languages like C/C++ [36, 35] provides platform-independent abstractions which also allow this outcome and various non-SC behavior in general. These subtle behaviors affect the correctness of program behaviors and hence require careful analysis.

The C/C++11 weak memory model is specified in the chapter 1,29 and 30 in the C++ draft[13]. Betty et.al[9] define the C/C++11 memory model as the form of executable program interleaving which is constrained by the mathematical semantics. To be specific, the execution is evaluated based on a mathematical graph where nodes represent events and edges are the relations between events(nodes) on the same thread. Batty et al[12] then fur-



(a) Case 1. (b) Case 2. (c) Case 3.

(d) Case 4. (e) Case 5. (f) Case 6.

Figure 2.3: Examples of memory order and synchronization in C/C++11 semantics.

ther simplify the sequential-consistency(SC) atomic semantics in C11 by relaxing the SC operations in one execution from totally ordered to partially ordered.

C/C++ Concurrency[36, 35] C/C++ has various kinds of accesses that affect the behavior of a shared memory concurrent program. To begin with, it provides a plain or non-atomic (*na*) load and stores access. In addition, C11 also has atomic accesses of four kinds: load, store, atomic updates (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 (*ACQREL*), and sequentially consistent (*SC*). Based on the kind of operation and memory order we categorize the accesses as follows:

$$acc ::= read_{o_r} \mid read_{o_w} \mid RMW_{o_u} \mid fence_{o_f} where \tag{2.1}$$

$$o_r ::= NA \mid RLX \mid ACQ \mid SC \tag{2.2}$$

$$o_w ::= NA \mid RLX \mid REL \mid SC \tag{2.3}$$

$$o_u ::= NA \mid ACQ \mid REL \mid ACQ_REL \mid SC \tag{2.4}$$

$$o_f ::= ACQ \mid REL \mid ACQ_REL \mid SC \tag{2.5}$$

For instance, an access is *acquire* if its order is once of *ACQ*, *REL*, *SC*. Similarly an access is *release* if its order is one of *REL*, *ACQREL*, or *SC*. The release and acquire accesses establish synchronization, for instance, when an acquire read reads from a release write.

For example, in Figure 2.3a, the memory order for all events are *RELAXED*, and there is no synchronization. In Figure 2.3b, event *d* and *b* can synchronize as one is *RELEASE* and another is *ACQUIRE*. In Figure 2.3e, event *d* and *a* can synchronize as they are all *SC*. However, even if the memory order of event *d* is *ACQUIRE* in Figure 2.3d, it cannot synchronize with any other event. Because there is no event with *RELEASE* memory order. So does for Figure 2.3c and Figure 2.3f. In conclusion, if two events can synchronize with each other, both of their memory orders have requirements.

Going forward, in *later* we discuss the formal model of C/C++ concurrency in detail.

2.2 PCT v.s. Naive Random Testing

PCT is a randomized concurrency testing algorithm that provides theoretical guarantees on the probability of detecting concurrency bugs. The key to its probabilistic guarantee is the notion of "bug depth", which is defined as the number of ordering constraints between the concurrent events of a program. Given the concurrency bug depth d and the number of instructions n in the program as inputs, PCT randomly generates a test execution that encodes a particular ordering of events with d scheduling constraints. Instead of sampling an execution from the set of all possible thread interleavings of size $O(t^n)$ for t threads, it samples from the set of executions $O(n^d)$ with d thread interleavings. Consequently, it guarantees a lower bound on the probability of detecting bugs that is exponentially low only in the depth parameter d.

Example: Program P1 2.1b Consider Program P1 2.1b (all the memory accesses are of *SC* memory order) which has a concurrency bug that occurs when the second thread reads $X = m$. The manifestation of the bug requires a single scheduling constraint, i.e., it requires the assertion statement in the second thread to be executed after the $X = m$ statement in the first thread. Given $d = 1$, PCT samples out of two $d = 1$ executions: It either chooses a schedule that runs all instructions in the first thread before the second thread or runs the second thread before the first thread. Therefore, it hits the bug with a probability of $1/2$.

However, a naive randomized testing algorithm that chooses the next action to run from the set of all enabled actions detects the bug only with the probability of $1/2^m$. To detect the bug, it must choose the action in the first thread among the two enabled actions for the first m actions of the execution. The mechanism and implementation of the PCT algorithm will be further discussed in Chapter 5 and Chapter 6.

Based on PCT algorithm, we propose a PCTWM algorithm, which revises the definition of bug depth in PCT algorithm and offers the theoretical guarantee for detecting concurrency bug on the weak memory model. The PCTWM is given and demonstrated in Chapter 5 and Chapter 6.

2.3 C11Tester - Automatic Testing Tool

C11Tester[51] is the state-of-art automatic testing tool for detecting concurrency bugs in C/C++11[8]. We implement our advanced algorithms - PCT and PCTWM, on it. The C/C++11 semantics is declarative(i.e. axiomatic), and program executions are not represented as traces of interleaved actions but rather as partially ordered graphs[8], which have to satisfy several consistency constraints. C11Tester adopts the graph theory to record, abstract, and check the program's execution.

C11Tester bounds the exploration space of an execution in two steps. The C11Tester first offers a set of available threads and actions whenever the execution needs. These choices are filtered by C11Tester to make sure not violate the C/C++11 relaxed atomic semantics[8].

The detailed introduction of C11Tester are in Chapter 3.

Chapter 3

Overview

In this chapter, we motivate the extension of PCT [21] for weak memory programs and present an overview of our PCTWM algorithm. First, we briefly revise the PCT algorithm in comparison to naive random testing. Then, we show that the theoretical guarantee provided by PCT does not apply to weak memory programs. Finally, we provide the two key steps for extending PCT to weak memory programs: (i) revising the notion of concurrency bug depth and (ii) extending PCT to PCTWM to generate test executions from a set of executions bounded by the revised notion of bug depth d and the history bound h.

3.1 C11Tester

C11Tester defines a new *pthread* library3.1, replacing the default library when compiling.

3.1.1 Mechanism of C11Tester

C11Tester focuses on two parts - atomic actions and thread scheduling. The PCT and PCTWM algorithms replaces the random selection of threads and read-from events in C11Tester.

Workflow of C11Tester Figure 3.2 concludes how C11Tester finishes an execution exploration.

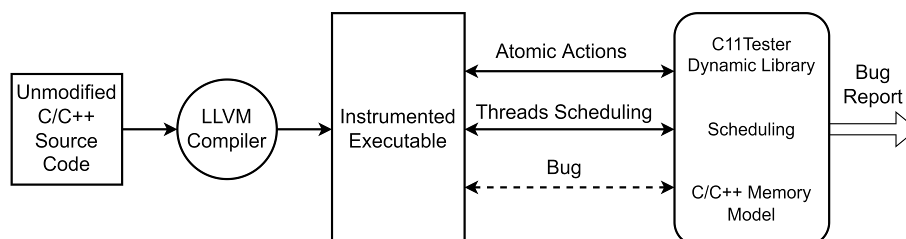


Figure 3.1: The Mechanism of C11Tester. C11Tester replaces *pthread* library.

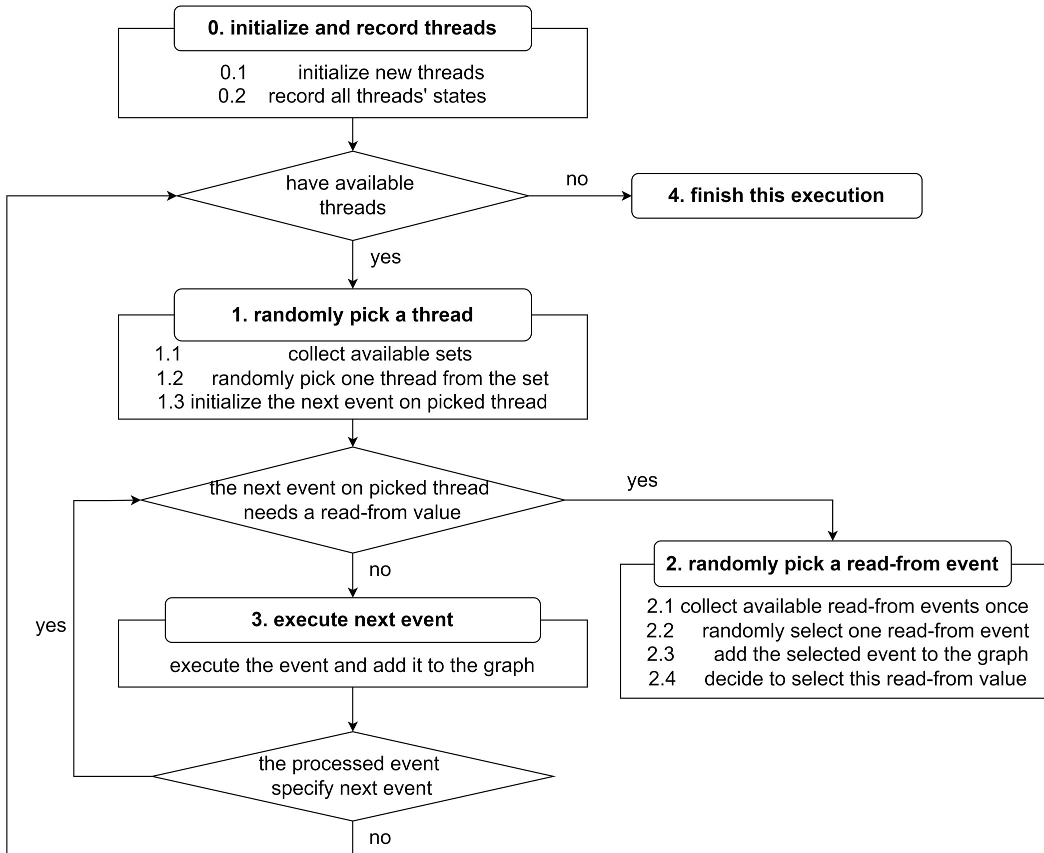
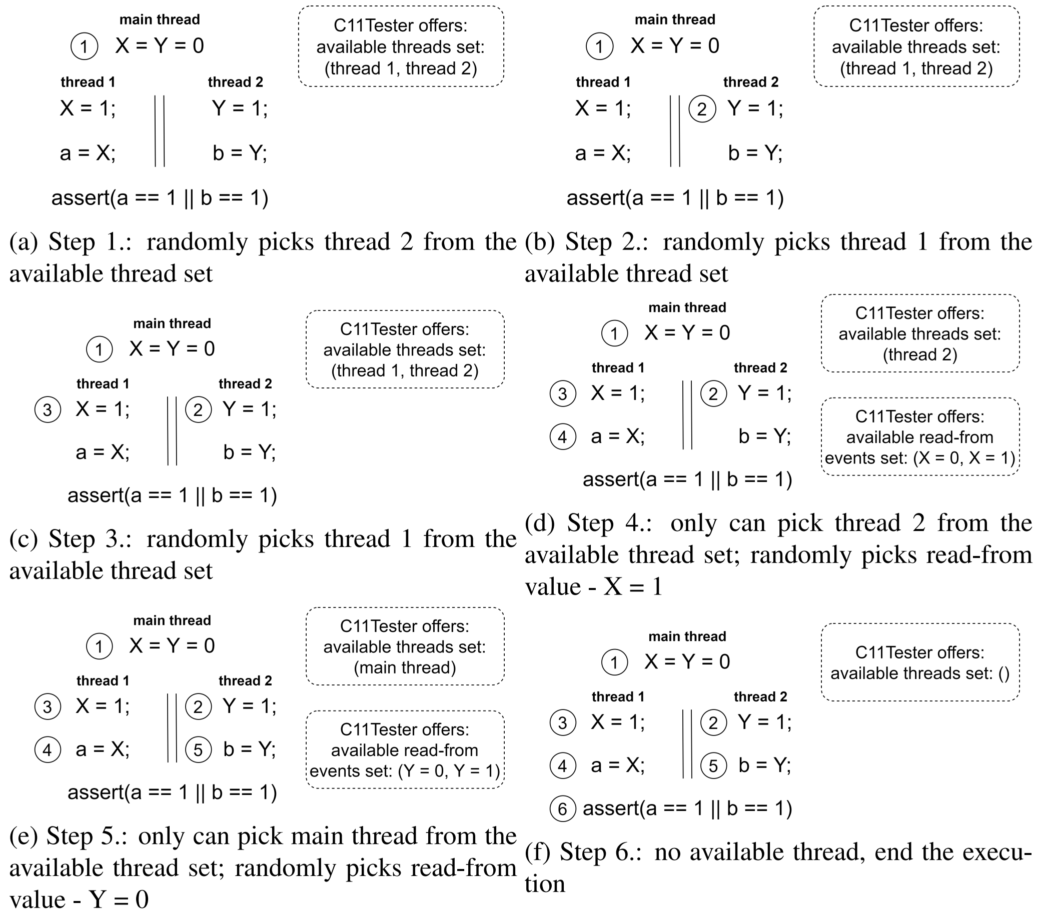


Figure 3.2: The workflow of C11Tester. Using **random** strategy to pick thread and read-from event when exploring the execution.

C11Tester first initialize each new thread and keeps recording their states(Step 0. in Figure 3.2). Every time C11Tester needs to decide to execute which thread(Step 1. in Figure 3.2) or read from which event(Step 2. in Figure 3.2), it first traverses each thread and collects the set of available thread(Step 1.1. in Figure 3.2) or read-from event(Step 2.1. in Figure 3.2) based on the C/C++11 semantics. The C11Tester adopts naive random strategy when it picks a thread(Step 1.2. in Figure 3.2) or a read-from event(Step 2.2. in Figure 3.2). After executing an event(Step 3. in Figure 3.2), C11Tester checks whether this event specifies the next event to executes(e.g., a RMW action is processed as a serial of a read action and a write action). The C11Tester explores an execution until there is no available thread(Step 4. in Figure 3.2).

3.1.2 Randomized Algorithm in C11Tester

C11Tester adopts the randomized strategy when it picks a thread to execute and selects a read-from event for a read action. The main task of C11Tester to explore an execution contains two steps: i) collects available choices(threads, events); ii) randomly select one from



(a) Step 1.: randomly picks thread 2 from the available thread set

(b) Step 2.: randomly picks thread 1 from the available thread set

(c) Step 3.: randomly picks thread 1 from the available thread set

(d) Step 4.: only can pick thread 2 from the available thread set; randomly picks read-from value - X = 1

(e) Step 5.: only can pick main thread from the available thread set; randomly picks read-from value - Y = 0

(f) Step 6.: no available thread, end the execution

Figure 3.3: The Step-by-step example of the randomized strategy in C11Tester. The dot-line box is the available thread or action set offered by C11Tester each step.

these available choices. When C11Tester collects the set of available threads, it checks the state of each thread to make sure it is not in sleep or blocked. And when collecting available read-from events, it checks whether executing it will violate the C/C++11 semantics.

Here we give an example(Program SB 2.1a) to see how C11Tester randomly schedules. Step 1, only main thread is created and C11Tester executes the write action on it(Figure 3.3a). Two child threads are created, C11Tester collects them. Step 2(Figure 3.3b), C11Tester randomly picks thread 2 and executes the next event($Y = 1$). C11Tester collects available thread sets: $(thread1, thread2)$. Step 4(Figure 3.3c), C11Tester chooses thread 1 and executes next event on thread 1($X = 1$). Still, thread 1 and 2 are available. It randomly picks thread 1 and goes to Step 4(Figure 3.3d). This time, the read event($a = X$) needs a read-from value. After traversing all threads, it offers an available read-from set:$(X = 0, X = 1)$. After randomly picking one write event, thread 1 is finished and no longer available. C11Tester only finds one available thread:$(thread2)$. In Step 5(Figure 3.3e), C11Tester offers available values on the location of variable Y and finishes thread

2. Now, only the main thread is available. In Step 6(Figure 3.3f), C11Tester executes the last action and finishes the execution because of no more available thread.

This execution 3.3 is just one possible of execution when C11Tester explores the execution of Program SB 2.1a. As the C11Tester randomly picks the threads and read-from events, its search space is a power of the number of threads. PCT and PCTWM algorithms introduced in this thesis replace the randomized selection in C11Tester.

3.2 A Naive Application of PCT to Weak Memory

Though the PCT algorithm [21] is effective in SC model and guarantees the probability to hit the bug. Its probabilistic guarantee for the lower bound on the probability of detecting bugs does not apply to weak memory programs. Here we use some examples to prove why the probabilistic guarantee does not work for the weak memory model.

Examples: Program 2.1b Paragraph 2.2 explains in Chapter 2 why the probability of hitting the bug in Program 2.1b in SC model is $1/2$.

But under weak memory concurrency, this execution does not necessarily hit the bug. Because, a read event can read from from any write event in the first thread. To be specific, if the memory order for all read and write events in Program 2.1b are *RELAXED*, then the read action can read from any write value rather than it can only read from the latest write event in the global program order. The violation occurs only if it reads the value of X from the last write event. Then the lower bound for the probability of hitting the bug becomes $\frac{1}{2m}$. This probability contains two parts. The first part is to schedule the write event $'X = m'$ on thread 1 before read event $'readX'$ on thread 2, whose probability is $\frac{1}{2}$. The second part is to pick the correct write event $'X = m'$ from the m write events, whose probability is $\frac{1}{m}$.

The Program 2.1b shows that the behavior of the weak memory programs does not depend only on the thread interleavings, but also 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 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* [21].

3.3 Revising Concurrency Bug Depth

The existing notion of bug depth does not capture weak memory concurrency bugs. Consider Program 2.1a. 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 and 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 communicate 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 2.1a 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 *view*s.

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: 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 2.1b is $d = 1$ under both notions. Under SC, the bug exposes in a single thread ordering. 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 $X = m$ to the read event in the second thread.

3.4 PCTWM: PCT for Weak Memory

Here we informally introduce the key ideas in the PCTWM algorithm, which we will elaborate in Chapter 4 and Chapter 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 keeps the updates made available to this thread.

For example, the $d = 0$ execution of Program 2.1a 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 2.1b 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 first thread.

Besides the number of communication relations d, PCTWM further parametrizes the execution space using a history bounding parameter h. This parameter considers the characteristics of the weak memory model - concurrency bug being highly related to the mistaken reading from some certain values. For this reason, PCTWM uses the history - h, to restrict 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 as a communication destination can read from only h possible values instead of all values collected by the C11Tester, further reducing the sample set of executions.

We provide the formal definition of a communication relation, thread-local view, the complete PCTWM algorithm, and its theoretical guarantees in Chapter 4. Same as the PCT algorithm, the PCTWM algorithm also samples the program's interleaving with a bounded number of preemptions by changing thread priorities. The implementation on C11Tester[51] is discussed in Chapter 6. The PCTWM recognizes the communication events and uses a global counter to record these events by order. Through randomly pick-

ing *d* communication events as the communication relation's destination, PCTWM controls the interleaving with selecting communication relations' source. The two types of views - *thread-local view* and *global view*, separately represents the view for the variables on current thread and the view for all the variables linked with the current thread, bounding the number of preemptions by controlling reading from which view.

Chapter 4

Weak Memory Concurrency Model

In this chapter, 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 relations between these events.

4.1 Event

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

For memory accesses, loc and Val denote the corresponding memory location and the read or written value. In case of fences, $loc = Val = \perp$. A successful RMW results in an update event (U) and on failure generate a read event (R). The set of read, write, update, and fence events are R, W, U, and F respectively. The memory locations are initialized at the start of the execution, represented by a set of non-atomic write events.

4.2 Relation

Various binary relations connect the events in an execution. Hence before explaining the relations we discuss the notations.

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 $imm(B)$ denotes the immediate relation: $imm(B)(x,y) \triangleq B(x,y) \land \nexists z B(x,y) \land B(y,z)$. 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. $[A](x,y) \triangleq x = y \land x \in A$.

An execution has the following relations between events: 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.

- Relation *poloc* relates same-location *po*-related events. i.e. $poloc\,(a,b) \triangleq po(a,b) \wedge a.loc = b.loc$.
- From-read (fr) 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 r and w' are in fr relation.
- We adopt the synchronizes-with (*sw*) relation from RC20 [53]. Relation happens-before (*hb*) is the transitive closure of *po* and *sw* relations.

$$poloc \triangleq \{(a,b) \mid po\,(a,b) \wedge a.loc = b.loc\} \tag{4.1}$$
$$fr \triangleq rf^{-1};mo \setminus [E] \tag{4.2}$$
$$sw \triangleq [E_{\sqsupseteq REL}]\,;([F]\,;po)^{?}\,;rf^{+};(po;[F])^{?}\,;[E_{\sqsupseteq ACQ}] \tag{4.3}$$
$$hb \triangleq (po \cup sw)^{+} \tag{4.4}$$

4.3 Execution

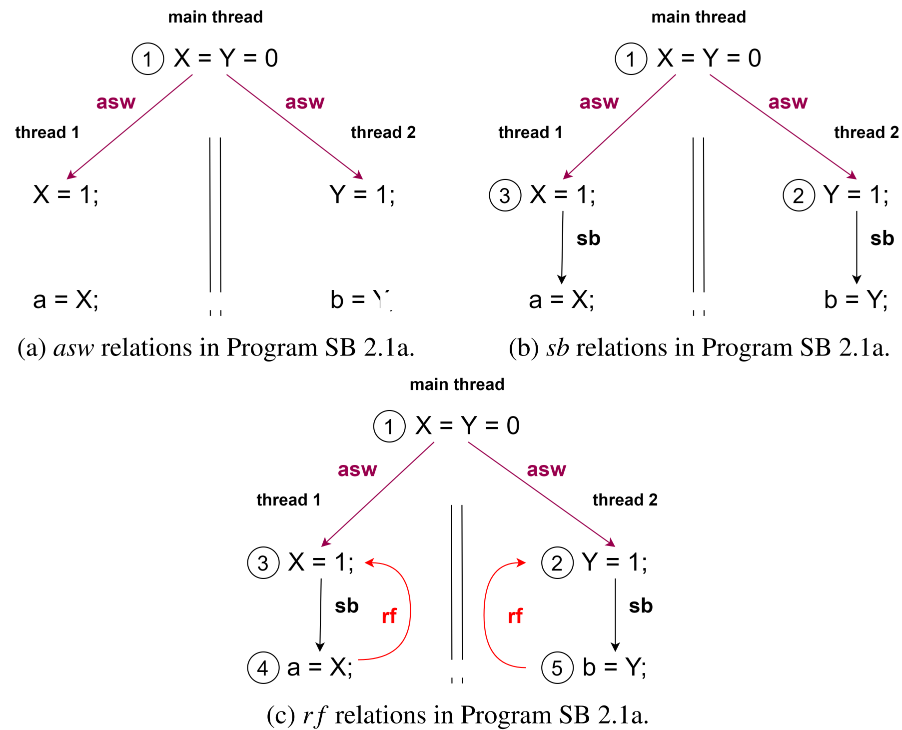
An execution $X = \langle E, po, rf, mo, SC \rangle$ is a tuple where $X.E$ is the set of events and $X.po$, $X.rf$, $X.mo$, $X.SC$ are set of po, rf, mo, SC relations between the events in $X.E$. We represent execution by an *execution graph* where events are represented by nodes are relations are represented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

Consistency Axioms C11 defines the following axioms to check if an execution is consistent.

- (coherence) The events accessing the same memory location are totally ordered due to the coherence property. Therefore $(poloc \cup rf \cup fr \cup mo)$ is acyclic.
- (Atomicity) The RMW accesses execute atomically. Hence $(fr;mo) = \emptyset$ holds.
- (irrMOSC) The *mo* and *SC* orders agree on same-location accesses, that is, $(mo;SC)$ is irreflexive.
- (SC) The SC accesses are globally ordered. There is a number of SC order definitions [8, 81, 11, 42, 46, 51].

  We follow the one from C11Tester [51], that is, $(hb \cup rf \cup SC)$ is acyclic.

  Note that the (SC) axiom enforces that *hb* is irreflexive (an action cannot *happens-before* itself) [8, 81]. Moreover, as $po \subseteq hb$, the (SC) constraint also enforces that $(po \cup rf)$ is acyclic and in consequence forbids out-of-thin-air reads.



(a) *asw* relations in Program SB 2.1a.

(b) *sb* relations in Program SB 2.1a.

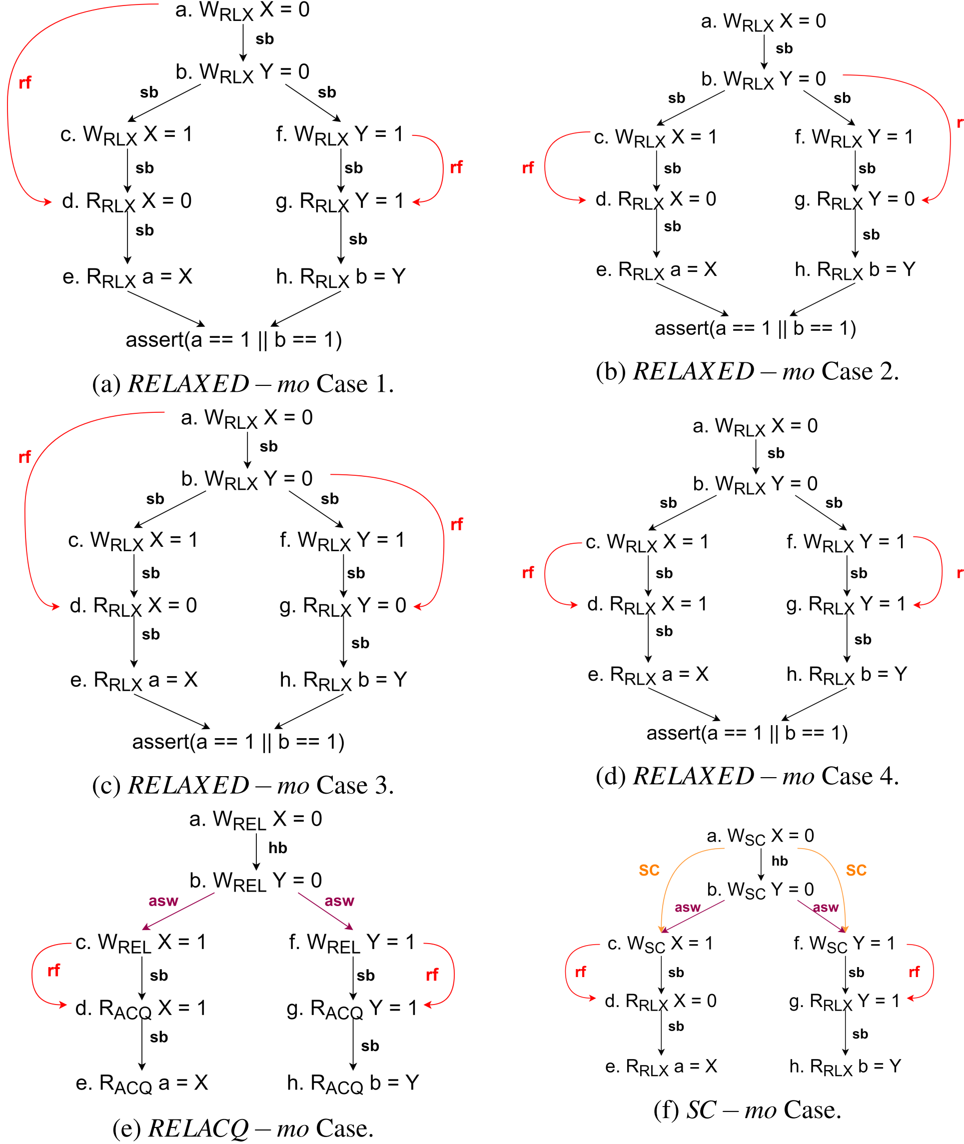
(c) rf relations in Program SB 2.1a.

Figure 4.1: Relations in Program SB 2.1a.

4.4 Example

Take Program SB 2.1a as an example to see the relations between different events in a program. In the Figure 4.1a, there is a *asw*(*additional-synchronizes-with*) relation between the main thread and two child threads - thread 1 and thread 2. The creation of a child thread always generates the *asw* relation. In thread 1 and thread 2, the write event is respectively *sequenced-before*(*sb*) the read event. But if all events are $RELAXED - mo$, the rf(*read-from*) relations may link the read-from value 0 or 1 with the read action.

Besides the relations in Program SB 2.1a, the read-from(rf relations) may be different in a weak memory model. In Figure 4.2, events in Program SB 2.1a have different *mo*(*memory order*), which reflects that *mo* can affect the range of read-from events. From Figure 4.2a to Figure 4.2d, events are all $RELAXED$, so the read action can read from both value 0 and value 1. When we change the *mo* of write actions from $RELAXED$ to *SC* or $RELEASE$, the C/C++11 semantics bounds the range of read-from values. If the *mo* of write events are updated to $RELEASE$ and *mo* of read events are updated to $ACQUIRE$, there is a *sw* relation between a $RELEASE$ and a $ACQUIRE$ event, which makes the read action read from the most 'recent' $RELEASE$ write. That is to say, event d reads from event c and event g reads from event f. If the *mo* of write and read actions become *SC*, there is a *SC* relation between them. Also, the *SC* read action can only read from the most 'recent' *SC* write event. Event d can only read from event c and event g read from event f.



(a) *RELAXED* − *mo* Case 1.

(b) *RELAXED* − *mo* Case 2.

(c) *RELAXED* − *mo* Case 3.

(d) *RELAXED* − *mo* Case 4.

(e) *RELACQ* − *mo* Case.

(f) *SC* − *mo* Case.

Figure 4.2: Concurrent Program Example - **SB** with different memory order types to illustrate C/C++ Concurrency. *mo* is the abbreviation of the memory order of actions. Arrow *sb* represents the 'sequence-before' relation. Arrow rf represents the 'read-from' relation. Arrow *sw* means the *RELEASE* and *ACQUIRE* actions are synchronized. Arrow *SC* is the total order for *SC* actions.

Chapter 5

Algorithm and Examples

In this section, we first explain the randomized strategy C11Tester used when it explores the execution. Then we formalize the description of the classical PCT scheduler. Then we extend the application of the PCT algorithm to the weak memory model and propose the new algorithm - PCTWM, i.e., PCT for weak memory.

5.1 PCT Algorithm and Theoretical Guarantee

5.1.1 PCT: Designed for Concurrent Programs on SC Model

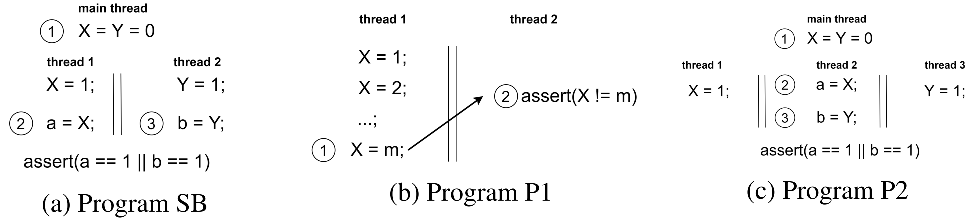
The PCT(Probabilistic Concurrency Testing) algorithm is proposed by Burckhardt et al.[21] to tackle the problem of the low efficiency in concurrency bug detection due to the time and resources in stress testing. The PCT algorithm is based on the critical observation that concurrency bugs can be viewed as unexpected interleaving over certain instructions [50][57]. This observation converts the bug detection to the proper scheduling of these relevant instructions no matter how many ways it can schedule instructions that are not relevant to the problem. Burckhardt et al. illustrates PCT's bug-finding capacity both theoretically and empirically by applying it to production-scale concurrent applications.

PCT algorithm generates a test execution with d thread scheduling constraints. The test execution for a concurrent program requires selecting the next event(thread) to execute. PCT algorithm realizes this thread selection and restricts the execution to switch threads only at $d-1$ thread priority change points.

5.1.2 Formal Definitions

Before explaining the PCT algorithm, we first give some formal definitions related to the PCT algorithm.

Thread Interleaving A concurrent program is composed of several threads with different events(i.e., actions) on each thread. The sequence of events on these threads is unique with each execution but may vary across executions. The sequence of events in one execution is the thread interleaving.



(a) Program SB (b) Program P1 (c) Program P2

Figure 5.1: Concurrent Program Examples

Definition 1 (Thread Interleaving/Sequence)**.** *Define T to be a set of thread identifiers and the T^* to be the set of finite sequences of elements from T. The sequence S is the element in T^*, i.e., $S \in T^*$. The initial length of the sequence $length(S)$ is zero and will increase in later scheduling, which means $length(S) \geq 0$.*

The element in a sequence S is the event t from threads, expressed as $t \in T$. So we can mark the element in the sequence as $S[n] = t, 0 \leq n \leq length(s)$. The sequences $S_1 \in T^*$ is the prefix of the sequence $S \in T^*$ if there is another sequences S_2 and $S = S_1S_2$.

We use Program SB 5.1a as an example to introduce thread interleaving. In the beginning, the sequence(i.e., scheduler) length is 0 and in Figure 4.1a, the main thread is first scheduled, adding 1 to the scheduler length. To locate the write action on the main thread, we can use $S[1]$. Then thread 2 is scheduled and the write event 'Y=1' is executed, also adding 1 to the scheduler length. When C11Tester switches between these threads randomly, the scheduler length is increased.

Schedule The schedule of a program defines its thread interleaving. Burckhardt et al.[21] can represent a program abstractly by its schedules and each schedule is simply a sequence of events.

Definition 2 (Schedule)**.** *Define $T = \mathbb{N}$ to be the set of thread identifiers. Define $Sched = T^*$ to be the set of all scheduling.*

The *schedule* is written as a sequence of thread identifiers and an execution can be abstractly expressed as its *schedule*. Still for the Program SB in Figure 4.2, the schedule can be written as the sequence '2112', where thread 2 takes one step, followed by two steps of thread 1, followed by another step of thread 1.

Though the schedule serves as an abstraction of the program state, it does mean that we can schedule any thread at any state. Because a thread may be blocked or in sleep at some states. A program can be scheduled only when it is enabled at that state.

Event Labeling The action a thread is going to execute is called an event and it is used for defining the ordering constraints. The event labeling is to mark what event the thread(t) is going to execute if scheduled next after scheduler S.

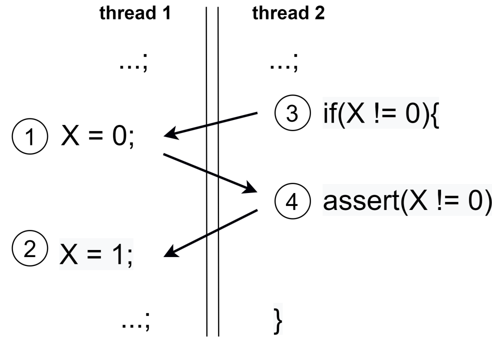


Figure 5.2: A concurrent program example: P3. Bug depth d = 3.

Definition 3 (Event Labeling)**.** *An event labeling E defines a set of event labels L_E and each label $a \in L_E$ belongs to a particular thread $thread_E(a)$. A function $next_E(S,t)$ tells what event label the thread t will execute if scheduled next after scheduler S.*

Let P be a program. An event labeling E is a triple $(L_E, thread_E, next_E)$ where L_E is a set of event labels, $thread_E$ is a function $L_E \to T$. Function $next_E \mid P \times T \to (L_E \cup \bot)$, which satisfies the following conditions:

- *(Affinity) If $next_E(S,t) = a$ for some $a \in L_E$, then $thread_E(a) = t$.*
- *(Stability) If $next_E(S,t) = a$ for some $a \in L_E$, and if $t \neq t'$, then $next_E(St',t) = a$.*
- *(Uniqueness) If $next_E(S_1,t) = next_E(S_1S_2,t) = a$ for some $a \in L_E$, then $t \notin S_2$.*
- *(NotFirst) $next_E(\varepsilon,t) = \bot, \forall t \in T$.*

In the Figure 4.2, thread 1 and thread 2 in Program SB 5.1a emits different events at different states. In Figure 4.1a, the next event on thread 1 is the write event 'X=1' and the next event on thread 2 is the write event 'Y=1'. And at the state in Figure 4.1b, the next event on thread 1 is the read event 'a=X' and the next event on thread 2 is the read event 'b=Y'.

Shared Access Event We define the *shared access event* to distinguish the events related to concurrency bugs. Because some action types are irrelevant to a concurrency bug, e.g., $thread_create$ or $thread_join$.

Definition 4 (Shared Access Event)**.** *The shared access event represents the events involved in thread-interleaving, containing all write, read, and fence actions.*

Bug Depth Burckhardt et al.[21] defines the bug depth as the minimum number of ordering constraints that are sufficient to guide the scheduler to hit the bug.

Definition 5 (Bug Depth)**.** *The formalization of bug depth - d, is the minimal size of scheduling constraints between event labelings that can guarantee to trigger the bug.*

Taking Program P3 5.2 as an example. Its bug depth d is 3. To hit successfully hit the assertion, scheduling generated by PCT should guarantee three ordering constraints: i) executing the 'zero' value check before writing zero to X; ii) switching to thread 1 after the 'zero' value check but before executing the branch; iii) inserting a priority change point after writing '0' but before writing value '1' to X.

Event Tuple The *event tuple* is a tuple that represents the necessary ordering of the events related to a certain bug. Once these events are scheduled as the order in this tuple, it can successfully hit the bug.

Definition 6 (Event Tuple)**.** *A tuple including d events is expressed as* $< E_1, E_2, \ldots, E_d >$*. Once a schedule can meet this order of* $E_1, E_2, \ldots, E_d$*, it can hit a bug with depth of* d*.*

Here are event tuples for the former four concurrency program examples. The event tuple of Program P1 5.1b to hit the assertion is { ① , ② }. For Program P2 5.1c, its event tuple is { ① , ② } to hit the bug. And for Program P3 5.2, its bug depth is 3 and the related event tuple is comparatively complex: { ③ , ① , ④ , ② }. If PCT scheduling satisfies the events ordering in this tuple, it can hit the assertion bug in Program P3 5.2.

The event tuple, can be viewed as a transformation of the ordering constraints, is critical to hit a bug. It is important to stress that the order of event in a tuple is strict. There should be no other events between the elements in a tuple.

5.1.3 The PCT Algorithm

The PCT algorithm - PCT(d,k) in Algorithm 1, takes the bug depth d as the bug depth and the instruction number$k$ as test parameters. The definition of the depth(d) of a concurrency bug is the minimal number of scheduling constraints necessary to locate the bug. Intuitively, bugs having a greater depth sample from an exponentially larger set of schedules are therefore by nature more difficult to locate. If a program has n threads, the initial priorities for each thread should vary from d to $d+n$. As for the total instruction(shared access events) number - k, we will generate $d-1$ priority changes in these k shared access events(instructions). That is to say, priority change points can be viewed as randomly picking $d-1$ different locations from all k locations in the program. The change priority points generated by parameters - d and k will decide whether we can meet certain ordering constraints.

Priority of Thread PCT algorithm adds a new feature to threads - priority. The priority is an integer larger than or equal to zero. The priority is given when a new thread is created and recorded until the end of execution. PCT realizes the scheduling with by controlling the thread's priority.

Initial Priority When a thread is created, no matter main thread or child thread, PCT gives it an initial priority. The range of initial priorities for a program with n threads and a d-depth bug is $[d, d+n-1]$.

Definition 7 (Thread Priority)**.** *For a program with n threads, PCT sets a n-size list to record the priority for each thread. Each thread in the program is given an initial priority when it is created. The range of initial priorities is from d to* $d+n-1$.

For example, Program SB 5.1a has main thread and two child threads, with a 0-depth bug. The range of initial priorities is [0,2]. Program P1 5.1b also has main thread and two child threads, whose bug depth is 1, initial priorities varying from 1 to 3. And for Program P2 5.1c, its bug depth is 0 and has four threads(main thread and three child threads). So initial priorities will be 0,1,2, and 3.

Priority Change Points PCT always picks the highest-priority thread to execute. So PCT can switch between different threads when it updates the priority of a thread. At a priority change point, PCT lowers the priority of a thread.

As PCT gives each thread a different initial priority and it always executes the highest-priority thread, there exists an initial order of each thread when the thread is created. For example, in Program P1 5.1b, if we give thread 1 priority 2 and thread 2 priority 1, PCT executes thread 1 first. With the initial priorities, PCT already meets the ordering constraint in Program P2 5.1c, executing the write event 'X=m' happen before the read event. This means that initial priorities can guarantee one ordering constraint.

Therefore, PCT picks $d-1$ points rather than d points when detecting a bug with depth of d.

Definition 8 (Priority Change Points)**.** *When PCT detects a d-depth bug, it randomly picks* $d-1$ *integers from* $[1,k]$. *k is the number of all shared access events. The list that saves the* $d-1$ *points is written as* $[d_1,\ldots,d_{d-1}]$.

For example, in Program P3 5.2, it has four events: ① ② ③ ④. This d = 3 execution of Program P3 5.2 picks two locations in all instructions as priority change points. Again, here PCT picks $d-1$(i.e.,two) switching points rather than d(i.e.,three) points because different initial priority assigned by PCT can guarantee one ordering constraint. The detailed explanation of PCT scheduling Program P3 is discussed later in Chapter 5.1.4.

Procedure *PCT* In Algorithm 1, we present the PCT algorithm based on the procedure of execution generation in C11Tester by sampling from the $d-bounded$ test executions. PCT takes bug depth d and instruction number k as input.

PCT maintains a list of *threads* that records the threads' ids according to their priorities, from high to low. PCT always chooses the next event on the current highest-priority thread. It schedules threads in order(i.e., priorities from high to low) and switches between them at randomly selected tuple of $d-1$ events, $[d_1,..,d_{d-1}]$, which are picked from the array $[1,k]$. The switching points(i.e., the priority change points) - $[d_1,\ldots,d_{d-1}]$, are kept in a $d-1$-size vector, whose elements are randomly picked from the range of $[1,k]$.

Vector *threads* is used to keep all threads' priorities and the function $highestPrEnabled(threads)$ will return the current highest-priority thread in it. We also adopt some parameters defined in C11Tester. The set of all currently available threads - $enabled(t)$ and the next enabled event on thread t at state s - $next(s,t)$. In each scheduling, PCT will check whether it

meets a priority-change point with the help of the vector that saves change priority points - $[d_1, \ldots, d_{d-1}]$ and $indexOf(i, list)$ to get the index of the element i in the list.

Algorithm 1 PCT

Data: *schedulerLength* // the count of scheduler length(number of calling the scheduler)
Data: $[d_1, \ldots, d_{d-1}]$ // list of $d-1$ distinct integers, initialized randomly between $[1, k]$
Data: t, e, s // the current executed thread, event and current execution state

```
 1: procedure PCT(d,k)
 2:     while enabled(s) ≠ ∅ do
 3:         for t ∈ enabled(s) do
 4:             if t ∉ threads then
 5:                 threads[randomIdx] ← t            ▷ give a new thread random priority
 6:             end if
 7:         end for
 8:         t ← highestPrEnabled(threads)           ▷ find the thread with highest priority
 9:         e ← next(s,t)                           ▷ get next event on selected thread
10:         schedulerLength ← schedulerLength + 1   ▷ increase scheduler length
11:         if schedulerLength ∈ indexOf([d1,...,dd−1]) then
12:             Index ← indexOf(schedulerLength,[d1,...,dd−1])
13:             threads[Index] ← t                  ▷ lower the thread's priority
14:             continue
15:         end if
16:         execute(e)                              ▷ execute next event on selected thread
17:     end while
18: end procedure
```

When exploring the execution space, PCT always selects the thread t with the highest priority from the enabled set(line 8 in Algorithm 1) and the next enabled event on thread t(line 9 in Algorithm 1). PCT counts the scheduler length and increases by one every time(line 10 in Algorithm 1). On line 11 in Program 1, PCT checks if the current scheduler length is among the selected $d-1$ locations. If this is the case, it means PCT now meets a thread switching(priority change) point. PCT will first find out the related index of this switching point(line 12 in Algorithm 1). Second, PCT puts the current highest-priority thread - t to this related index among all threads(line 13 in Algorithm 1), which can be viewed as lowering the priority of thread -t. Then, PCT executes the next event(line 16 in Algorithm 1). After the thread priority update(line 13), in the next loop(line 2 in Algorithm 1), PCT will repeat this procedure.

5.1.4 Examples and Lower Bound on Probability to Hit the Bug

Example: $d = 3$ Consider the Program P3 in Figure 5.2. In this example, the bug depth is 3. The three scheduling constraints are i) checking variable X not '0' before writing '0' to X; ii) switching to thread 2 before writing '1' to X but after writing '0' to X; iii) writing '1' to X after the assertion. PCT needs $d-1(2)$ switching points to hit the assertion bug.

In Figure 5.2, the event tuple to hit the bug in Program P3 is { ③ , ①, ④, ② }. The first constraint { ③, ①} can be satisfied by the initial priorities - thread 2 being given higher initial priority than that of thread 1. The second constraint:{①, ④}, is realized by PCT updating the priority of thread 2 before executing ③ after executing ③. For the third constraint:{ ④, ①}, PCT updates the priority of thread 1 before executing ② after executing ①.

Example: $d = 0$ **and** $d = 1$ As mentioned above, for a bug with a depth of d, PCT picks $d-1$ switching points. But the number of switching points should larger than or equal to zero. To avoid confusion, we gives the example of program with $d = 0$ and $d = 1$. For a bug depth lower than 1, PCT does not pick any switching point but assigning initial priorities to each thread and executes them one by one.

Program SB 5.1a includes an order violation, whose bug depth is 0. In the weak memory model, the order violation appears when the *mo* of events are *RELAXED*. As shown in Figure 4.2a4.2b4.2c4.2d, the order violation can be triggered. As its bug depth is 0, PCT does not pick any switching point. PCT only assigns the initial priority to thread 1 and thread 2. PCT either executes thread 1 first, followed by thread 2 or executes thread 2 first, followed by thread 1. These two executions generated by PCT have no difference in the probability of hitting the bug. As the bug lies in the selection of read-from values. This bug can be triggered by at least one of the read events on thread 1 or thread 2 that read from the write event on the main thread($X = Y = 0$).

Program P1 5.1b includes an assertion violation, whose bug depth is 1. As its bug depth is 1, PCT does not pick any switching point. If PCT gives thread 1 a higher initial priority, it executes thread 1 first. After finishing thread 1, PCT switches to thread 2. If PCT can pick the write event 'X=m' for thread 2 to read from, it hits the bug. But if PCT gives thread 2 a higher initial priority, it executes thread 2 first and PCT cannot read from the write event 'X=m', which makes it impossible to hit the bug. The probability to hit the assertion in Program P2 5.1c is $\frac{1}{2m}$.

5.1.5 Theoretical Guarantee(Lower Bound on Probability)

For a bug with the depth of d, PCT picks $d-1$ change points uniformly among all k shared access events. Picking switching points is the first part of the lower bound on probability, which is $\frac{1}{k^{d-1}}$. The second point we need to consider is that one scheduling constraint is guaranteed by the initial priority. For example, in Program P3 5.2, thread 2 should be given a higher priority at first. The probability of giving a proper initial priority is $\frac{1}{n}$. Combining these two aspects, we give a lower bound on the theoretical probability of the PCT algorithm to hit a 'd'-depth bug - $\frac{1}{nk^{d-1}}$.

5.2 PCTWM Algorithm

5.2.1 Application of PCT to Weak Memory

While PCT guarantees a lower bound on the probability of detecting bugs, its theoretical guarantees do not apply to weak memory programs. We demonstrate this on a variant of

Program 5.1b where all the load and store accesses to the variable X are relaxed accesses. In that case, an execution that schedules all operations of the first thread before the second thread does not necessarily produce the bug. The load operation in the second thread can get X's value from any of the store operations in the first thread.

Different from SC programs, the behavior of the weak memory programs depends on not only the thread interleavings but also on the selection of the store operations that the load operations get the values from. Based on this observation, we revise the notion of concurrency bug depth to capture the effect of the store operations on the load operations on the same variables.

The traditional notion of concurrency bug depth based on the scheduling order of the program actions does not apply to weak memory programs.

We demonstrate that the existing notion does not capture weak memory concurrency bugs using Program 5.1a. The program exhibits a buggy behavior when both variables a and b load the value 1. The traditional notion of bug depth is defined as *the minimum number of thread scheduling constraints that are sufficient to produce the bug* [21]. However, the bug in Program 5.1a does not depend on the scheduling order of the actions. Rather, the bug exposes in the *communication* of the effect of a particular store operation in the first thread to the load operation in the second thread.

5.2.2 Revising Concurrency Bug Depth

We revise the concurrency bug depth definition to capture *thread communication.* rather than *thread interleavings*. More specifically, we define the depth of a concurrency bug as the minimum number of *communication relation*s between the concurrent events in the execution. Roughly, a communication relation between two concurrent events communicates the effects of some store operations to a load operation. For example, the depth of the concurrency bug in Program 5.1a is $d = 0$ since it does not require any communication between its threads. That is, the actions in each thread only access the values of the variables that are available in their thread-local *view*s.

Notice that the revised definition of the bug depth extends the existing notion that uses thread interleavings. For the case of sequentially consistent programs, every thread interleaving induces a communication relation: the effects of all the store operations in an executing thread are communicated to the other threads at the thread interleavings. For example, the depth of the concurrency bug in Program 5.1b is $d = 1$ using both notions. The bug exposes in the presence of a single communication relation between its events, i.e., the communication of the effect of the store $X = m$ to the load operation in the second thread. As another example, the assertion violation in Program 5.1c has bug depth of $d = 2$. It requires the communication of (i) the store operation on X in the first thread to the read operation on X in the second thread and (ii) the store operation to Y in the third thread to the read operation Y in the second thread.

5.2.3 PCTWM: PCT for Weak Memory

PCTWM extends PCT to generate an execution with d communication relations instead of d ordering constraints. Similar to PCT, PCTWM takes the number of instructions k_{com}[1] in a program and the concurrency bug depth d as input and samples an execution from the d-bounded set of executions. However, different from PCT, PCTWM uses the revised definition of bug depth; it does not sample from the executions with d thread scheduling constraints, but from the executions with d communication relations. Roughly, it (i) chooses d operations as communication destinations among the set of *communication events* (e.g., load operations) and orders them to schedule in a particular order, (ii) chooses a source operation (e.g., a store operation) for each communication event before scheduling them. Bounding the number of communication relations by d restricts the amount of thread interaction in an execution. Without any restrictions, a relaxed load operation can potentially read from any store operation on the same variable. However, bounding an execution to have only d communication relations allows only d operations to read from an external value that is not yet communicated to the loading thread. It restricts the behavior of the rest of the load operations to read the values available in their thread local *views*.

For example, the $d = 0$ execution of Program 5.1a 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 5.1b 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 first thread. Consider another program, give in the Figure 5.1c. A $d = 1$ execution of the program allows the communication of either the value of X stored in the first thread or the value of Y stored in the third thread to the load operations in the second thread. An execution that loads both $a = 1$ and $b = 1$ is of concurrency depth $d = 2$ since it requires two communication relations.

Besides the number of communication relations d, PCTWM further parametrizes the execution space using a history bounding parameter h. The history 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 as a communication destination can read from only h possible values instead of m values, further reducing the sample set of executions.

For example, an execution of Program 5.1b with $d = 1$ and $h = 2$ detects the concurrency bug with probability $1/2$. First, it chooses $d = 1$ communication destinations. This example has only one possible communication destination, i.e., load operation in the assertion statement. Then, it chooses a source operation for the communication relation within a history bound $h = 2$. In this example, it can select to read from either $X = (m-1)$ or $X = m$ each with the probability of 1/2, the latter hitting the bug.

We provide the formal definition of a communication relation, the complete PCTWM algorithm, and its theoretical guarantees in Section 5.2.7.

[1]Instead of the total number of instructions, PCTWM takes the number of *communication events* as we describe in Chapter 5.

The PCTWM algorithm extends PCT to weak memory programs in a *memory model agnostic* way so that its guarantees apply 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.

Based on the underlying memory model these concepts can be defined.

It extends PCT by checking (i) whether the event may communicate with an event of another thread, i.e., it is a *communication event* and (ii) maintaining the local views of the threads for each variable.

5.2.4 Formal Definitions

Definition 9 (Communication relation). *Following the (SC) constraint in C11Tester model (see Chapter 2), we consider inter-thread* rf*, hb, SC as com relations, that is,* $com \triangleq (rf \cup hb \cup SC) \setminus po$.

Definition 10 (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 a 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 a SC, or read, or acquire event.*

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

Definition 11 (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 12 (View). *A view is a map from locations to a set of maximal-mo events. Given an execution* $\langle E, po, rf, mo, SC \rangle$, $view(x) = maximal_{mo}(E_x)$ *holds where* E_x *are the set of write or update events.*

- *Combine views on a location x. We write* $\sqcup_{mo}(view_1(x), view_2(x))$ *to compute the maximal view from* $view_1(x)$ *and* $view_2(x)$ *for a given location x, i.e.* $maximal(view_1(x) \cup view_2(x), mo)$.
- *Combine views on all memory locations. Similarly, we write* $\sqcup_{mo}(view_1, view_2)$ *to compute* $\sqcup_{mo}(view_1(x), view_2(x))$ *for all memory locations x.*

In one execution, each thread may maintain its own view. We write $t.view$ *to denote the view of thread* t*. Essentially, a thread view maintains the latest write or update events observed by the thread for each memory location.*

Definition 13 (History depth)**.** *The history depth h bounds a read event in one execution to read-from an event that does not have more than h $imm(mo)$-related successors.*

5.2.5 The PCTWM Algorithm

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

Generating a test execution for a weak memory program requires (i) selecting the next event to execute and (ii) selecting the behavior of the event (e.g., selecting which event to read from). The PCTWM algorithm binds these two choices and restricts the execution to switch threads only at d points that correspond to the external reads or synchronization of the inter-thread events. PCTWM sorts the threads in random order and switches between them at d points that can observe the effects of the events from the other threads. It restricts the weak memory behavior of the rest of the events to access their local thread views.

We present the PCTWM algorithm (see Algorithm 2) following the structure of the C11Tester [51] by (i) incorporating d-bounded test generation in PCT [21], and (ii) maintaining the thread-local views for computing the behavior of communication events.

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

PCTWM maintains a list of *threads* that keeps the thread id's 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 order w.r.t. 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 $[1, k_{com}]$. We also keep a set of events, *reordered*, at which the threads switch. These d events are singled out to potentially read from externally written values of the accessed variables. 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 that are enabled in state s, and $next(s,t)$ to refer to the next enabled event in thread t at state s. We also use $highestPrEnabled(threads)$ 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 5 in Algorithm 2) and the next enabled event e of t (line 6). 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 the communication events encountered in the execution (line 8 in Algorithm 2) and check if that event is among the randomly selected d events (line 9 in Algorithm 2). If this is the case, we delay the execution of its thread by reducing its priority (line 11) and adding the event to the set *reordered* (line 12). We update the priority of the thread to the value, say d_e, corresponding to e's index in the tuple

Algorithm 2 PCTWM

Input: The bound on the number of comm. events k_{com}, bug depth d, history depth h
Data: $threads$ // the list of threads in ascending order of priorities, the first d positions are initially $null$
Data: $[d_1, \ldots, d_d]$ // list of d distinct integers, initialized randomly between $[1, k_{com}]$
Data: $reorderedEvents$ // the set of event ids reordered with a thread priority change, initially empty
Data: $numEvents$ // the count of comm. events observed, initially 0

1: **procedure** PCTWM(k_{com}, d, h)
2:     **while** enabled(s) $\neq \emptyset$ **do**
3:         **for** $th \in enabled(s)$ and $th \notin threads$ **do**
4:             //insert th to a random index after d in $threads$
5:             $t \leftarrow getHighestPrEnabled(threads)$
6:             $e \leftarrow next(s, t)$
7:             **if** $isCommunicationEvent(e)$ **then**
8:                 $i \leftarrow i + 1$
9:                 **if** $i \in \{d_1, \ldots, d_d\}$ **then**
10:                     $Index \leftarrow indexOf(i, [d_1, ..., d_d])$
11:                     $threads[Index] \leftarrow t$
12:                     $reordered \leftarrow reordered \cup \{e\}$
13:                     continue
14:                 **end if**
15:             **end if**
16:             executeAndUpdateView(s, e)
17:         **end for**
18:     **end while**
19: **end procedure**

1: **procedure** ISCOMMUNICATIONEVENT(t, e)
2:     return $e \in (SC \cup \mathscr{R} \cup F_{\sqsupseteq ACQ})$
3: **end procedure**

Algorithm 3 executeAndUpdateView(s, e): Executes e and updates the thread view

1: **procedure** EXECUTEANDUPDATEVIEW(s, e)
2: $b \leftarrow \bot$
3: $x \leftarrow e.loc$
4: **if** $e \in SC$ **then**
5: $e' \leftarrow getSC(t, e)$
6: $t.view \leftarrow \bigsqcup(t.view, e'.bag)$
7: **end if**
8: **if** $e \in \mathscr{W}$ **then**
9: $t.view(x) \leftarrow e$
10: **end if**
11: **if** $e \in \mathscr{R}$ **then**
12: **if** $e \in reordered$ **then** ▷ read from any of the store operations
13: $b \leftarrow readGlobal(t, h)$
14: **if** $isSync(e, b)$ **then**
15: $t.view \leftarrow \bigsqcup_{s.mo}(t.view, b.bag)$
16: **else**
17: $t.view(x) \leftarrow \bigsqcup_{s.mo}(t.view(x), b.bag(x))$
18: **end if**
19: **else** ▷ read from the local thread view
20: $b \leftarrow readLocal(t)$
21: **end if**
22: **end if**
23: **if** $e \in F_{\sqsupseteq ACQ}$ **then**
24: $esw \leftarrow getSWSet(t, e)$
25: **for** $e' \in esw$ **do**
26: $t.view \leftarrow \bigsqcup_{s.mo}(t.view, e'.bag)$
27: **end for**
28: **end if**
29: **if** $e \in F_{REL}$ **then** ▷ no update to the current thread's view
30: **end if**
31: $e.bag \leftarrow t.view$
32: $s \leftarrow executes, t, b$
33: **end procedure**

$[d_1, \ldots, d_d]$. This delays the execution of e to execute after all program events except for the events $[d_{e+1}, \ldots, d_d]$. reordering its execution preserving the order of events identified by $d_1, \ldots, d_d$. Enforcing a particular order between the communication events provides the visibility of some $dom(com)$ events to some $codom(com)$ events. We provide an example test execution in Section 5.2.6 that enforces a certain ordering and weak memory behavior of the events forming communication relations.

The PCTWM algorithm extends PCT in a *memory model agnostic* by using the in a memory model agnostic procedures: (i) *isComEvent*, which is used to check if an event is *communication events* and potentially update the thread priorities only at such events (line 7) and (ii) *executeAndUpdateView*, which is used to update the local views of the threads using (line 14).

Procedure ***isComEvent*.** Following the definition of communication events above, a communication event is: (1) an *SC* event or (2) a read event which may read from other threads, or (3) a synchronization event, that can be a source or 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 every 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 its the source, 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 type of the communication relation.

On line 1 in Algorithm 3 we keep a reference b for a read or RMW event e to store the behavior of the write event e reads-from. Based on the type of e we update the view of thread t and the *bag* of e. On lines 3-4, if e is a non-communication event (i.e. a relaxed or non-atomic write) then it updates the view of t only at the location accessed by e. If e is a read or RMW from the *reordered* set, that is, one of the communication sinks, then On line 8 e reads from a visible write or RMW event w within history bound h, else on line 15 e reads-from local event to thread t by *readLocal*. If e reads-from a non-local event, it updates the view of t. We store the thread view propagated from w to t at b. Next, on line 10, if e is the sink of an *sw* relation then we update the view of t with the *bag* in b with the *mo*-maximal events per location. Otherwise, on line 12, we update the view of t only for the memory location accessed by event e.

On lines 16-17, if e is a write or RMW then the view of thread t is updated with event e at the location of e. On lines 18-19, if e is an acquire or stronger fence then it updates the view of thread t with the views of all events with which it synchronizes (returned by *getSWset*).

On line 4-7, if e is an *SC* event then it updates the view of thread t with the views of its *SC*-predecessors (returned by *getSC*).

Finally, we store the view of t in the *bag* of e on line 30 and update the execution state on line 32.

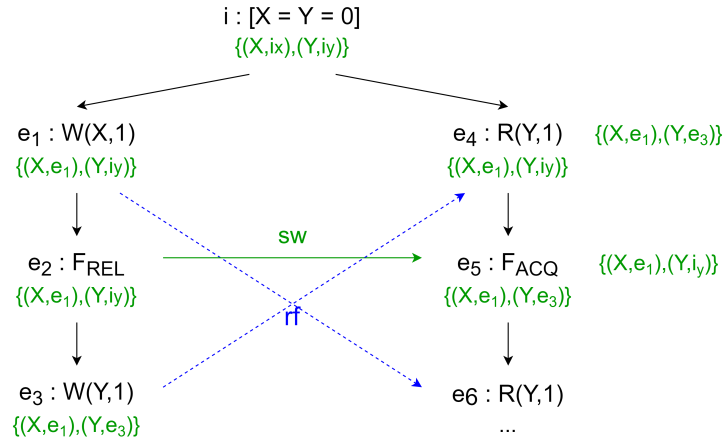


Figure 5.3: MP1: execution a = 1, b = 1 with views and bags.

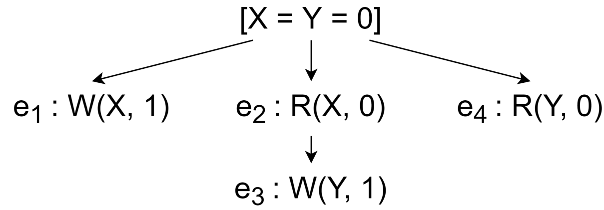
Example Consider the Program 5.3. In this program $a = 1, b = 0$ results in a bug.

The execution in the Program 5.3 shows that if $a = 1$ then also $b = 1$. In 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. Execution of e_1 updates the thread view to $\{(X, e_1), (Y, i_y)\}$ which remains same after e_2 following line 6 and 26 respectively in the Algorithm 3. Execution of e_3 updates the thread view on Y (line 6 in Algorithm 3). The read event e_4 reads-from e_3 and obtains T1's view in its *bag* (line 9 in Algorithm 3). It updates only the view on X, and the updated view is (X, e_1, Y, i_y) following line 16 in Algorithm 3. 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 21-25 in Algorithm 3. Finally, while executing e_6 the only write available in the thread-local view is e_1 which overwrites the initialization on X. As a result, e_6 must read value 1 following line 21 in Algorithm 3. In that case outcome $a = 1, b = 0$ is a bug.

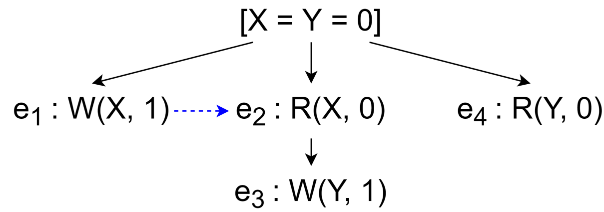
5.2.6 Example Test Executions Generated by PCTWM

We now discuss some example executions generated by PCTWM for testing Program 5.4, 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 T1, T2, and T3. The execution of a program that reads $Y == 1$ and $X == 0$ in T3 hits an assertion violation. While the 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.

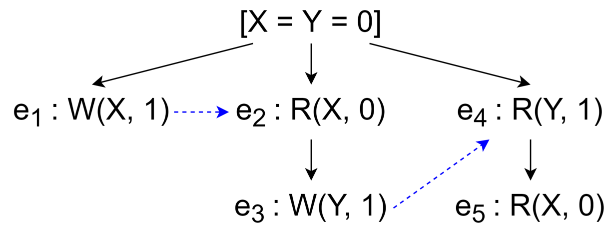
Generating the execution with $d = 0$. The $d = 0$ execution of Program 5.4 (see Figure 5.4(a)) does not have any communication relations. Therefore, it does not allow any external reads. Following Algorithm 2, PCTWM randomly assigns priorities to the threads



(a) The d = 0 execution



(b) A d = 1 execution



(c) A d = 2 execution

Figure 5.4: Three test executions for the Program MP2

and runs them serially in the order of their priorities. Since $d = 0$, it forbids any thread switches or communication relations.

Generating an execution with $d = 1$. PCTWM generates a $d = 1$ execution of the program by randomly sampling a communication relation in the execution. The algorithm randomly assigns priorities to the threads, (e.g., in the decreasing order to T1, T2, and T3, respectively for the execution in Figure 5.4(b)). Then, switches the threads at a randomly selected $d = 1$ event, allowing that event to read-from a value written in another thread (if it is a read event) or synchronize with an external event (e.g., a read-acquire even can synchronize with a write-release event). For this program, PCTWM selects the $d = 1$ event out of $k_{com} = \{e_2, e_4, e_5\}$. In the example execution in Figure 5.4(b), it selects $[e_2]$ as the sink of the *com* relation.

The test execution starts running the threads in the order of their priorities, reordering the execution of T2 at e_2 by reducing the priority of its thread (line 11 in Algorithm 2).

The test execution continues with T3, executing e_4. Since these events are not involved in a communication relation, they read from their thread local views, i.e., $X = 0$ and $Y = 0$ (corresponding to line 21 in Algorithm 3). After the completion of T3, PCTWM executes e_2. Since e_2 is reordered to form a communication relation with an earlier event, i.e., $e_2 \in$ *reordered*, it can read globally from any value within a bound of $h = 1$ (line 23). In this example, e_2 reads from e_1, forming the communication relation (e_1, e_2). Then, PCTWM runs the next event e_3 in T2 completing the test execution.

Generating an execution with $d = 2$. PCTWM generates a $d = 2$ execution of the program by randomly sampling two communication relations in the execution. Similar to the previous case, it randomly assigns priorities to the threads, (e.g., in the decreasing order to T1, T2, and T3 respectively). It switches the threads at a randomly selected $d = 2$ events, allowing only these events to read-from or synchronize with an externally written value. In this example, PCTWM selects the tuple $[e_2, e_4]$ of $d = 2$ events which can access thread external writes. The algorithm runs T1 executing e_1, delays the execution of e_2 by reducing the priority of T2. The execution continues with T3 but this time it also delays the execution of T3 at e_4 as well. The updated priorities of T2 and T3 preserve the order of the selected events $[e_2, e_4]$. Therefore, the execution moves to T2, executing $e_2 \in$ *reordered* by allowing it to read from an external value and form a communication relation. In this execution, e_2 reads from e_1 which forms the communication relation (e_1, e_2). Then, the PCTWM algorithm executes e_3 and moves to T3. Since $e_4 \in$ *reordered*, it forms a communication relation. In this execution, e_4 reads from e_3, which forms the communication relation (e_3, e_4). This execution with two communication relations hits the assertion violation.

This example highlights several insights of the algorithm. First, more complex executions with deeper concurrency bugs manifest in the existence of a higher number of communication relations. Second, the order of the events identified by $[d_1, \ldots, d_d]$ that are selected to form communication relations affect the generated test execution. For example, if the algorithm generates a test case by selecting $[e_4, e_2]$ instead of $[e_2, e_4]$, then e_4 can only read $X = 0$. The execution order of the selected d events affects the set of visible values to a read event. Finally, a communication relation updates the thread local views based on the semantics of the events in the formed communication relation. For example, the communication relation (e_3, e_4) in Figure 5.4(c) updates only the variable Y in the thread local view of T3 (following line 18 in Algorithm 3). 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 (corresponding to line 10 in Algorithm 3).

5.2.7 The Probability of Detecting Bugs using PCTWM

Given a program with k_{com} communication events, PCTWM samples an execution with d communication relations and a history bound of h with the probability of at least $\frac{1}{O((h.k_{com})^d)}$. It 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, it selects a communication event as the source 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 $O((h.k_{com})^d)$. Trivially, the probability of choosing an execution out of this set is at least $\frac{1}{O((h.k_{com})^d)}$, which is exponentially low only in the bug depth parameter d.

Chapter 6

Implementation

This chapter describes how the PCT and PCTWM algorithms are implemented in the C11Tester. We illustrate the PCT and PCTWM implementation based on the algorithm's procedure described in Chapter 5.

6.1 C11Tester Implementation and Plugins for Algorithms

The C11Tester replaces the pthread library functions to control the thread scheduling. C11Tester checks the consistency between its execution and the C/C++11 semantics when executing each action. It adopts the graph theory to maintain the correctness of the execution. The events and relations between events are separately abstracted as the nodes and edges in the graph. The C/C++11 standard's semantics are abstracted as the acyclic rule in the graph.

It also applies the full semantics in the C/C++ memory model to constrain the action selection. In other words, the C11Tester can offer the available threads for each step and randomly select the next executed thread and the next action on this thread. When the program needs to choose an event, e.g., a read-from value, C11Tester randomly selects from the available event set until the selected event does not violate the C/C++11 semantics. C11Tester utilizes the naive randomized thread scheduling, randomly picking one thread from current available(*non-sleep* and *non-blocked*) threads' set - rf_set. The implemented detector can report any data race or assertion violations it meets.

The workflow of C11Tester is simply abstracted in Figure 3.2. We can view the execution in C11Tester as an action-selection loop until the execution meets an end - no available thread or action to execute. First, by initializing each thread and recording their states, C11Tester collects the available threads' set every time it calls the thread scheduler. Second, C11Tester randomly picks an enabled thread from the set of available threads. If there exists at least one available thread, C11Tester processes the next event on this thread. Otherwise, this execution is finished. When processing an event, C11Tester checks whether it needs to read from other events and also collects available read-from events. Note that the correctness of possible behavior for an event is checked by the acyclic of the graph. Third, an event will be executed and added to the graph. Fourth, after executing this action, C11Tester finds out whether this event specifies the next event(e.g., the RMW action

is divided into serial actions - a read and a write action). If this is not the case, C11Tester repeats the first step - collecting currently available threads.

Property	Meaning
mo-graph	The graph abstracted by C11Tester to check the consistency between execution and C/C++11 semantics.
action_list	AA list, which is derived from the *mo-graph*, records all activities in one spot or on all threads. C11Tester backward traverses an action list on one location to collect possible behaviors for some events.
rf_set	The vector defined by C11Tester, which contains all available read-from events in the current state. Collected once every state.
check_current_action	The core function in interface *execution*. It checks and processes the chosen event based on its type.
select_next_thread	The core function in interface *scheduler* to realize the thread scheduling. It traverses the *threads* and collects available ones. In this function, we can implement different strategies to pick a thread to execute among currently available threads.
take_step	The thread selection action in *execution*, returns the next scheduled thread based on the current executed event.

Table 6.1: Properties of C11Tester and related to PCTWM implementation.

As we discussed before, C11Tester adopts the graph theory to keep track of all events and relations between them. The graph abstracted by C11Tester is called - mo-graph, i.e., modification order graph. Specifically speaking, it uses the delay decision in the action selection. It first randomly selects the required action from the allowed action set and then adds this action to the graph to check whether a cycle appears. If no cycle is created, the action is assumed as executable and moves to the next step. Otherwise, the selected action violates the C/C++ semantics and repeats the random selection until the appropriate action is chosen.

Random Testing in C11tester[51] The C11Tester adopts graph theory and associates every state transition taken by the thread with the dynamic operation that affected the transition. It uses a set to record the enabled sets and get the next transition on each thread.

As shown in figure6.1, C11Tester explores the execution space in two steps:(i) selecting a thread and executing the next event on this thread; (ii) selecting the behavior of the selected thread's next operation. The C11Tester employs a random technique by default for the thread and transition selection, which is the central component of the execution exploration. The pluggable framework for testing algorithms to intelligently choose the next thread and behavior is implemented by the C11Tester.

Collect Available Read From Events C11Tester collects a set of enabled read-from events for a read event. When collecting the enabled events, C11Tester first locates the

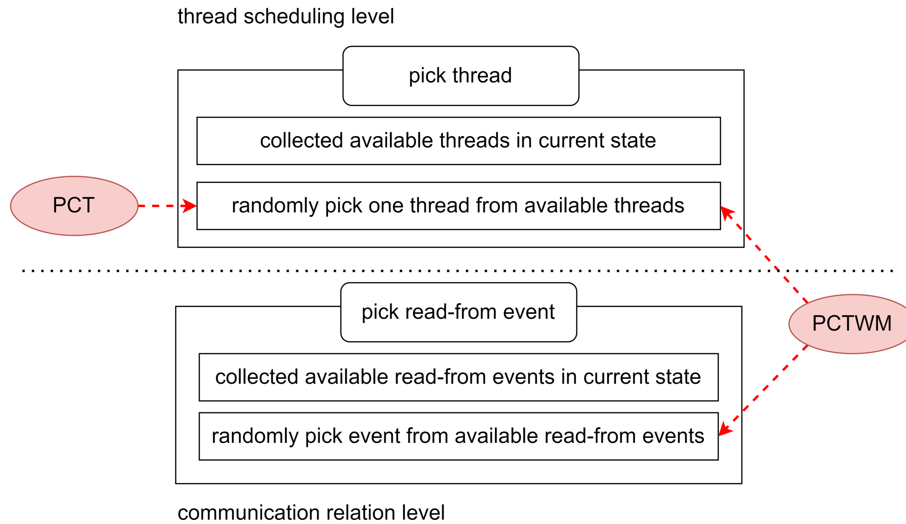


Figure 6.1: PCT and PCTWM implementation on C11Tester. Above the dotted line is the thread scheduling level. Below the dotted line is the communication relation(read-from value selection) value.

last action on each thread. And then it traverses back the action list until it meets a stop condition defined by C/C++11 semantics.

Interfaces of C11Tester The following are the key components of the C11Tester: 1) *cyclegraph* - to create and update the graph to model-check the correctness of the execution; 2) *execution* - to process the execution based on different actions and memory orders; 3) *scheduler* - the randomized scheduler to filter enabled threads and randomly select one thread to execute; 4) *threads* - to save the state of each thread including all parent and child threads; 5) *action* - to initiate and process each action by its action type, also to pass the information to the model checking for C/C++ semantics inside C11Tester; 6) *model* - control the whole C11Tester running; 6) *main*: to accept and set default parameters.

6.2 PCT and PCTWM: Parameters

This section specifies the flags we set for the parameters in PCT and PCTWM, through which we can pass the parameters to sample the scheduling. For example, in the command, we can use '-d2' to set the bug depth as 2 for PCTWM. The implementation of the PCT and PCTWM does not affect the old flags set by the C11Tester, e.g., controlling the execution time with flag 'n'.

Flag	Parameter	Range
d	bug depth	≥ 1
l	number of shared access events (i.e., instructions) ≥ 1	
s	seed	≥ 0

Table 6.2: Flags in PCT Implementation

Flag	Parameter	Range
d	bug depth	≥ 1
k	the estimated number of communication events	≥ 1
y	the history bound in collecting communication events for the view	≥ 1
s	seed	≥ 0

Table 6.3: Flags in PCTWM Implementation

6.3 PCT Implementation

PCT algorithm is implemented in the *scheduler* interface, replacing the naive random thread scheduling with priority-based scheduling. Based on the PCT implementation in the scheduler, PCTWM has been further implemented in interface *execution*.

6.3.1 Parameters

Algorithm Parameters As we discuss before, PCT has two parameters to control the scheduling - bug depth d and shared memory accesses events k. They are define in the header file of *params* and implemented in the *main* file. After the parameters are passed to PCT through the *main* function, the interface *scheduler* gets the parameters it needs. Second, we set a flag - 's' to change the random seed. The default seed is already set but this parameter can offer more options for sampling.

Additional Parameters These are some parameters used in implementation to sample the scheduling.

scheduler length This parameter is used to count how many times PCT schedules the threads, which relates to the priority change points and livelock number. The initial scheduler length is zero and it will increase by one every step.

livelock The model-checker approach has a common limitation - some scheduling samples may meet a 'livelock' in the search space, whose processing will be discussed in Chapter 6.3.3. It is more likely for PCT to create such schedules as it preempts threads only for a limited number of times. For this reason, we add another parameter - livelock in the C11Tester. It is automatically assigned a multiple of the number of shared memory accesses events k. Every time the *scheduler length* meets the times of *livelock*, it will adopt the random selection to jump out of the livelock.

6.3.2 Implementation based on Algorithm 1

Assign Initial Priorities Every time a new thread is created, PCT will give it a random initial priority(line 2 - 7 in Algorithm 1). If a program has n threads, then the initial priorities for these n threads vary from d to $d+n-1$. The priorities 1 to $d-1$ are used for the priority change points.

In the PCT implementation, an n-size vector is created to represent the initial priority for each thread. By inserting the thread' id to this vector randomly when C11Tester initializes a new thread, each thread's priority is given based on its index in this vector.

Pick the Highest-Priority Thread and Execute Next Event Different from randomly picking one thread among the available threads, PCT always chooses the current highest priority thread(line 8 in Algorithm 1). It executes the next event on this chosen thread(line 9 in Algorithm 1). C11Tester realizes the randomized thread scheduling in function *select_next_thread()*. PCT adds a function(*getHighestPriorityThread*) to pick the highest priority thread. It takes the set of all current available threads' ids as input and returns the id of the highest priority thread among these available ones. As for executing the next event on a chosen thread, it is the same as C11Tester.

Priority Change Points The list of priority change points, i.e., the thread switching points, plays a critical role in PCT implementation. Because this list $[d_1,\ldots,d_{d-1}]$ decides how the PCT updates threads' priorities and switches between them.

First, PCT picks $d-1$ different integers from the range $[1,k]$. In order to produce and preserve these numbers, we define a vector *chg_pts* - $[d_1,\ldots,d_{d-1}]$. To verify that these integers are distinct, PCT shall traverse the elements in this vector and keep generating new ones until the newly created one is distinct.

Second, PCT counts the length of the scheduler(processed shared access events). Based on lines 10 - 12 in Algorithm 1, the length of a scheduler is used to check whether it is the priority change point. The *schedulerlength* is initialized as zero and PCT increments the length of the scheduler in the function(*select_next_thread()*), which marks how many times PCT calls the thread scheduler. Every time PCT calls *select_next_thread* to choose one enabled thread, its length increments by one.

Third, to check whether meeting a thread switching point(priority change point), PCT compares the current scheduler length with values in the list - $[d_1,\ldots,d_{d-1}]$. If the scheduler length equals one integer in this list, it means that the scheduler meets a change priority point.

Fourth, if it is a priority change point, PCT gets the index of this priority point in $[d_1,\ldots,d_{d-1}]$ (line 12 in Algorithm 1). The function *find_chgidx()* takes the current scheduler length as input and return the index of this length in the list - $[d_1,\ldots,d_{d-1}]$. And PCT updates the priority of the current chosen thread - t, also the highest priority thread. The new priority assigned to this thread is the *Index* found by PCT. Function *movethread(index, threadid)* is implemented to realize line 13 in Algorithm 1, whose inputs are the index of change point and the id of current thread t.

6.3.3 Optimization

Processing Livelock The *livelock* refers to the case that the program gets stuck in one thread and cannot stop executing this thread or switch to another thread. This may result from the current highest-priority thread needing to read from values on other lower-priority threads. The *livelock* may happen in any model-checker that explores the executions' search space for a concurrent program on the weak memory model. For the naive randomized testing, this kind of case may not happen too often as every time the scheduler will randomly pick a thread to execute. However, in PCT, the scheduler will stay on one thread until it meets a priority change point, which makes it more likely to meet the livelock.

To prevent the livelock, we also add another parameter - *livelock* when implementing PCT. It is automatically assigned a value - a multiple of the number of the shared access events k by passing the parameter. PCT algorithm counts how many shared access events have been processed. And each time the number of processed shared access events is an integer multiple of parameter *livelock*, PCT makes a random selection to replace the highest priority selection so that PCT can jump out of the 'live lock'.

Input Protection Each parameter in PCT algorithm has a range. For example, the bug depth d in PCT algorithm should larger than or equal to zero but PCT picks $d-1$ priority change points. In the implementation, when the input d is lower than zero, PCT does not pick any priority change point.

The optimizations - processing livelock and input protection, in PCT implementation, are also added to PCTWM implementation.

6.4 PCTWM Implementation

The implementation of the PCTWM mainly covers the *execution*, *threads*, *action* and *scheduler*. The core realization of the PCTWM algorithm is to check and process action based on its type and memory order. How to process an event also decides how threads switch. In the *execution*, this implementation mainly relies on the API *check_current_action*. In the *threads* and *actions*, we add the feature of the visible synchronization vector to save the visible newest value for each shared variable. The priority-based thread scheduling is implemented in the *scheduler*.

6.4.1 Parameters

Algorithm Parameters Table 6.3 lists the parameters related to PCTWM implementation. First, the bug depth d in PCTWM represents the minimum communication relations it needs to trigger the bug. Second, PCTWM uses flag k to pass the number of communication events. Third, as we discuss in Chapter 5, PCTWM introduces a new parameter - history. Fourth, PCTWM still uses the flag s to represent the random seed.

Additional Parameters **priority change points** In interface *scheduler*, the list(*chg_pts*) saves d distinct integers, representing the d switching points.

Number of Processed Communication Events This parameter counts how many times PCTWM has met and processed a communication event, which relates to the priority change points and livelock number.

Livelock Similar to PCT, PCTWM also adds the parameter - *livelock*. It is automatically assigned a value - multiples of the communication events k_{com} by passing the parameter. PCTWM counts how many communication events have been processed. And every time the number of processed communication events is an integer multiple of parameter *livelock*, PCTWM makes a random selection to replace the highest priority selection so that it can jump out of the 'live lock'.

6.4.2 Features Added to Interface

Action According to the algorithm 2, an action may have one *bag* to save its *view*. So in the implementation, each action has a new vector to save its *view* for each variable.

Thread Similar to the action, the thread also has a vector as its local *view*. First, every time the C11Tester processes a write event on the thread, local *view* will be updated. Second, when a communication relation is set up, a read action reading globally, the local *view* will be updated as the values on other threads are 'visible' to this thread now.

Scheduler Moreover, for the operation *read*, we have two types - *global* and *local* as we discuss in the algorithm 5. So we set a bool-vector *external_read_thread* in the scheduler, which records whether currently, the thread needs to read *globally*.

6.4.3 Implementation based on Algorithm 2 3

We illustrate the implementation of PCTWM based on the procedure in Algorithm 2 and Algorithm 3.

Assign Initial Priorities On lines 2 - 4 in Algorithm 2, PCTWM assigns initial priority to a thread when initializing a new thread in the same way as PCT does. The difference between PCTWM is the range of initial priorities. When searching the bug with the depth of d, PCT picks $d-1$ change points while PCTWM picks d communication events to read globally. So the initial priorities assigned to n threads in PCTWM vary from $d+1$ to $d+n$.

Pick the Highest-Priority Thread and Execute Next Event Line 5 and 6 in Algorithm 2 is to pick the highest-priority thread and execute the next event on this thread. In interface *scheduler*, function *find_highest* returns current enabled thread with the highest priority by inputting current enabled threads' ids.

Generate Priority Change Points As we mentioned before, for a bug depth d, PCTWM needs to randomly pick d integers between $[1,k]$. n interface *scheduler*, these distinct integers are saved in a list - *chg_pts*. PCTWM also defines a function *set_chg_pts_byread* to generate priority change points, whose inputs are d, k and random seed - s.

Count Processed Communication Events PCTWM counts how many communication events it has processed. It first checks whether current event is a communication event(line 7 in Algorithm 2), which is implemented in interface *execution*. PCTWM processes an event(check_current_action), it first checks whether it is a communication event. If this is the case, PCTWM calls the function *IncInstrNum* to add one to the counter (line 8 in Algorithm 2).

Update Priority at Priority Change Point By comparing current processed events' number and integers in *chg_pts*, PCTWM knows whether it is a switching point with the help of function - *reach_chg_idx*. This function has two inputs - current processed events number and priority change points - $[d_1, \ldots, d_d]$. If it is the priority change point, it returns the index of this change point - an integer ≥ 0. If it is not, it returns -1. This implementation is consistent with lines 9 - 10 in Algorithm 2.

PCTWM lowers the priority of the current executed thread t. Using the function *movethread*, PCTWM updates the location of the current highest-priority thread to the related index of priority change point. This means it may no longer be the highest priority thread. Besides, if PCTWM updates the priority of the highest-priority thread when it processes a *read* action, it sets a read global job for this thread. Because PCTWM will switch to the 'new' highest-priority thread instead of processing the event on the 'old' highest-priority thread. But the *reodered* event needs to read globally when it is processed.

The difference in PCTWM is lying in the process that it delays the execution of its thread by reducing its priority (line 11) and adding the event to the set *reordered* (line 12). In the event processing function - *check_current_action*, it will mark the $change_flag$ as true to announce that PCTWM now meets a priority change point and does not process this event.

After updating the thread's priority and reordering the event, the last step of PCTWM when meeting a priority change point is to re-select a new thread with the highest priority in function - $take_step$.

Process the Event based on its Type In Algorithm 3, we use multiple if-branches to illustrate the different processing when PCTWM meets different types of events.

Line 3 - 5 in Algorithm 3 is implemented in function *process_write*. Line 6 - 17 is included in function *process_read*. Line 21 - 28 is how PCTWM process fences, which are realized in function *process_fence*.

Update View for Thread and Event View12, is a map from locations to a set of maximal-*mo* events. The implementation of *view* for threads and events is in the form of a list - saves the latest visible event for each variable. Though the definition and implementation of *view* for thread and event are similar, their roles are slightly different. When reading locally, the read action can get the latest visible local read-from value directly from the current *view* on the thread. The *View* on event is used for updating the *view* on a thread faster. Algorithm 3 uses different names to distinguish the *view* for events and threads. For the thread view,

Algorithm 3 uses *view* - *t.view*. And for events' views, Algorithm 3 names it *bag* - *e.bag*, as shown in line 10, 12, 24, and 31.

View Update for Event The *bag(view)* of an event represents the visible latest value for each variable when the thread emits this event in the schedule. The *bag(view)* of an event can help save time when we compute the view for a thread. Because the view for a thread needs to be computed when an event on it communicates with a global event on another thread.

PCTWM computes the *bag(view)* for all communication events - *write_RELEASE*, *read*, and *fence* events.

So when PCTWM updates the *view* for an event, it adopts the backward traverse too, which means it will stop searching values for its view when confronting an event with *view*(named *bag* in implementation). Because the *bag* of an 'older' event is out-of-date.

View Update for Thread The view for a thread - *t.view*, is a new feature for each thread. In the Algorithm 3, line 4, 9 - 13, 19 , 24, and 31 are about updating the thread view. And the implementation is mostly in the functions - *computeUpdate and computeUpdate_fence*.

First, as the main thread is always first initialized, every new child thread is assigned with the main thread's *view* when a child thread is created. Second, every time PCTWM meets a write action in one thread, we will update(*process_write*) the view of it. Third, when PCTWM meets a communication event, PCTWM also updates its thread's local view(*updatelocvec*).

Chapter 7

Experiment and Evaluation

In this chapter, we discuss our evaluation of PCTWM on several benchmarks and then compare the results with the state-of-the-art weak memory testing tool C11Tester.

Evaluation Method We repeat the testing on one benchmark 1000 times and count how many runs it detects the bug(assertion or data race). We compare the bug detection capability of the algorithms by comparing their bug hitting rates.

7.1 Benchmarks

7.1.1 Benchmarks with Assertion Violation

These two benchmarks are injected with one assertion as a bug.

Seqlock This benchmark is taken from the seqlock implementation from Figure 5 of Hans Boehm's MSPC 12 paper[16], made the writer correctly use release atomics for the data field stores, and injected a bug by weakening atomics that initially increment the counter to relaxed memory ordering.

rwlock The author of the C11Tester also designs a flawed reader-writer lock in which the write-lock procedure makes the mistake of wrongly using relaxed atomics. This benchmark uses double protections - the read-lock to protect read actions from loading atomic variables and the write-lock to protect write actions from storing to the atomic variables.

7.1.2 Benchmarks with Data Race

The data structure benchmarks to compare the ability of the C11Tester and the algorithms to discover data races are gathered from https://github.com/mc-imperial/tsan11.

barrier: The spinning barrier has one writer and some readers in which the number of the readers can be adjusted. The barrier has an initial value representing the number of threads it needs to synchronize and a variable recording the number of synchronizations completed so far. The barrier will stop spinning until the number of completed synchronizations reach the set value. The injected bug is to use the wrong relaxed atomics for the variable which stores the number of spinning threads.

dekker: This benchmark uses two variables: flag0/1 and turn. A flag0 value of true indicates that process 0 wants to enter the critical section. Entrance to the critical section is granted for process P0 if P1 does not want to enter its critical section or if P1 has given priority to P0 by setting turn to 0. The injected bug is to make the mistake of wrong relaxed atomics to change the flag0/1 value.

cldeque: This benchmark is a realization of a double ended queue from the paper[44], using relaxed operations (for efficiency) and fences and release/acquire synchronization to establish order. While the study verifies the accuracy of an ARM implementation, its C11 implementation is not validated. The benchmark's test driver employs two threads; the thread that owns the deque pushes three work items and receives two work items, while the other thread steals one work item. The previous model-checker identified a flaw in the published implementation. When a steal and push action occur simultaneously and the push operation resizes the deque, the issue arises. The bug manifests as a load from a possibly uninitialized memory address.

mpmcqueue: This multiple-producer, multiple-consumer queue[24] allows many readers and writers to access it simultaneously. The test driver will run two threads that are the same. Each thread will enqueue an item before dequeuing an item later on in the process.

linuxrwlocks: This linux reader-writer lock[61] allows readers or writers to hold the lock at once, but not both. The test driver uses two identical threads and a single rwlock t to test the Linux reader-writer lock. Each thread reads under a reader lock and writes under a writer lock.

mcslock: This benchmark is taken from the implementation of contention-free lock from Mellor-Crummey and Scott[23][54]. The lock functions like a concurrent queue, where threads are queued in a first-in, first-out fashion. Each thread alternates between reading and writing the same shared variable, releasing the lock-in between operations. The injected bug is to load the value of the variable - gate with relaxed atomics during locking.

msqueue: This benchmark is a C/C++ memory model adaptation of the Michael and Scott[55] lock-free queue. The injected bug is to make as much use as possible of relaxed atomics. The test driver operates two mirrored threads while each thread enqueues and then dequeues a single item.

7.1.3 Real Application Benchmarks

The three real applications that used for the performance(execution time or throughput) evaluation are Iris[90],a low-latency C++ logging library; Mabain[29], a key-value store library; Silo[77][80], a multi-core in-memory storage engine.

Mabain Mabain is a lightweight key-value store library. Mabain contains a few test drivers that insert key-value pairs concurrently into the Mabain system. Same as the previous work in the C11Tester, the assertions are turned off in the test driver for performance measurement. The measurement metric is the execution time of inserting 100,000 key-value pairs into the Mabain system.

According to the bug analysis in the C11Tester paper[51], the test driver has one asynchronous writer and a few workers. Workers and writers share a locked queue. The writer consumes jobs (database inserts) in the queue, while workers submit jobs. When all jobs

are submitted, the writer stops. The bug lies in the lack of checking to make sure all jobs are finished before stopping the writer. As a result, after the writer is stopped, some values may not be inserted into Mabain, producing assertion failures.

Iris Iris is a low latency asynchronous C++ logging library that buffers data using lock-free circular queues. The *test_usepackage_lfringbuffer.cpp* is selected as the test driver to measure performance, including one producer and one consumer. For the evaluation of all three tools, the number of iterations in the test driver was set to 1 million. All tools reported data races in Iris.

Silo Silo is an in-memory database that is designed for performance and scalability for modern multicore machines. For this application, *dbtest.cc* is chosen as the measurement, the running time of this test driver is set to 30 seconds and each run has 5 threads in parallel. Different from the other two applications, throughput is the metric of measuring performance for Silo.

7.1.4 Test Parameter for each Benchmark

For PCT and PCTWM, they both have the bug depth - d. To repeat, the definition and implementation of bug depth in PCT and PCTWM as we discussed in Chapter 5. In PCT, the bug depth d represents the minimum number of ordering(scheduling) constraints that are required to trigger the concurrency bug. And in the implementation, a bug with the depth of d requires the PCT to pick $d-1$ priority change points. In PCTWM, the bug depth d means that at least d communication relations are required to trigger the concurrency bug and we need to pick d events as the destinations for d communication relations.

Shared access events, the total amount of which is k, are designed for PCT algorithm, including all write, read, and fence actions. Communication events k contains all sequential consistent(SC), read, fence, and write release operations. We estimate the events in the while loop for once but multiple it with the for a loop. We also do not count the shared variable initializations as these accesses never execute concurrently.

Benchmark	d	LOC	PCT k	PCTWM k_{com}
dekker	0	50	20	14
msqueue	0	232	49	31
barrier	1	38	15	10
cldeque	1	122	82	56
mcslock	2	75	21	14
mpmcqueue	2	108	22	17
linuxrwlocks	2	90	31	20
rwlock	2	98	79	74
seqlock	3	50	17	14

Table 7.1: Parameters and Estimated Values for Benchmarks.

Research Questions To evaluate the effectiveness of PCTWM we address the following research questions:

RQ1. With what frequency does PCTWM detect the bugs when we provide the theoretical bug depth for the parameter*d*?

RQ2. How does the bug detection ability of PCTWM change when varying parameters - bug depth *d* or history *h*?

RQ3. Does PCTWM algorithm improve the bug hitting rate compared to C11Tester random testing?

RQ4. Does the PCTWM improve for the programs with a higher amount of weak memory accesses?

RQ5. Does PCTWM cause the overhead in C11Tester?

RQ6. How does the bug hitting rate change when changing the parameter(bug depth/instruction number) in the PCT and PCTWM algorithms?

7.2 RQ1. Bug Detection Ability with Estimated Parameters

Our first research question is to test whether we can detect the bugs with the estimated parameter values for each benchmark. Table 7.2 lists the bug hitting rate of the PCT and PCTWM with predicted value in Table 7.1 for each parameter.

7.2.1 PCT

For all benchmarks, PCT with the estimated parameters in Table 7.1 finds the bug successfully. However, the bug hitting rates of benchmarks - *dekker, cldeque, seqlock, and rwlock*, are lower than those of C11Tester. The results in Table 7.2 are obtained with the theoretical bug depth *d*, whose bug hitting rates may be lower. Because switching the threads precisely with theoretical bug depth has lower probability. We are going to further enhance the bug hitting rate of PCT by increasing *d* in the following section.

7.2.2 PCTWM

PCTWM with the theoretical bug depth and estimated communication events successfully trigger the bug in all nine benchmarks. And we can see in Table 7.2 that PCTWM with the test parameters always has a higher bug hitting rate compared with PCT.

7.3 RQ2. Bug Detection Ability Varying Bug Depth and History

In this question, we bound the number of shared access events *k* and communication events k_{com} in these benchmarks and change the value of *bug depth* and *history* to see how the bug hitting rate changes.

Benchmark	d	PCT		PCTWM		
		k	rate(%)	k_{com}	rate(%)	history
dekker	0	20	22.2	14	100	1
msqueue	0	49	100	31	100	1
barrier	1	15	73.9	10	77.8	2
cldeque	1	86	51.6	56	48.9	1
mcslock	2	21	100	14	100	1
mpmcqueue	2	22	99.8	17	100	1
linuxrwlocks	2	31	100	19	100	1
rwlock	2	79	19.7	74	76.9	4
seqlock	3	17	23.1	14	24.3	3

Table 7.2: Data Structure Benchmark Bug Hitting Rate with Estimated Parameters' Values.

Varying Bug Depth In the former research question, we test the bug hitting rate of PCTWM and PCT algorithm with the estimated bug depth, which is the minimum number of communication relations we need to trigger the bug. We bound the number of shared access events, and then increase the bug depth d in PCT and see the changes in the bug hitting rate for each benchmark.

Table 7.3 and Table 7.4 gives the bug hitting rate for each benchmark when the bug depth is set as the theoretical depth d, $d+1$, $d+2$ and $d+3$.

We can see that when we increase the bug depth in PCT algorithm, the bug hitting rate of benchmarks *cldeque* increase a lot. For benchmark *rwlock*, even there is an increment, the bug hitting rate decreases when the bug depth is added to $d+3$. So does for the benchmarks *barrier* and *seqlock*.

And for PCTWM algorithm, the bug hitting rate of benchmarks *cldeque* increases a lot. The bug hitting rate of benchmark *dekker* even decreases. Because for the benchmark *dekker*, the theoretical bug depth is 0 and it requires the read event to read locally. If we increase the bug depth, it allows the read event to read globally, which decrease the probability to hit the bug.

Table 7.3 and Table 7.4 show that, higher bug depth can increase the bug detection ability of PCT and PCTWM algorithms, but it is not a given.

Varying History In this paragraph, we continue testing the PCTWM bug detection ability by changing history - h. Table 7.5 gives the bug hitting rate over 1000 runs from history - h - 1 to 4.

For benchmarks the hitting rate can reach 100 percent, changing history does not affect the bug hitting rate. Our first analysis is for benchmark *dekker and msqueue*, their bug depths are zero, representing that bug can be hit when no communication relation is formed. In PCTWM algorithm, we can control events to read locally. Our second analysis is for benchmarks with bug depth higher than zero - *cldeque, mcslock, mpmcqueue, and linuxrwlock*. By analyzing their event trace, we can see that the selected communication events are perfect in these benchmarks. And sometimes when a read action needs to read

Benchmark	d	Rate(d)	Rate(d+1)	Rate(d+2)	Rate(d+3)
dekker	0	22.2	21.7	22.7	23.3
msqueue	0	100	100	100	100
barrier	0	73.9	76.8	72.4	73.5
cldeque	1	51.6	100	100	100
mcslock	2	100	100	100	100
mpmcqueue	2	99.8	100	100	100
linuxrwlocks	2	100	0	100	100
rwlock	2	19.8	49.6	73.7	61.4
seqlock	3	23.1	26.2	28	24.5

Table 7.3: Data Structure Benchmark Bug Hitting Rate in 1000 rounds(%) - Varying Bug Depth in PCT. k is the same as we list in Table 7.1

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	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	24.3(h:4)	24.7(h:3)	25.6(h:2)

Table 7.4: Data Structure Benchmark Bug Hitting Rate in 1000 rounds(%) - Varying Bug Depth in PCTWM. k_{com} is the same as we list in Table 7.1

globally, it may only have one write value read from in this scheduling, which guarantees the 100% hitting rate. In these two cases, changing history will not affect a lot, as the scheduling decided by PCTWM is enough to hit the bug.

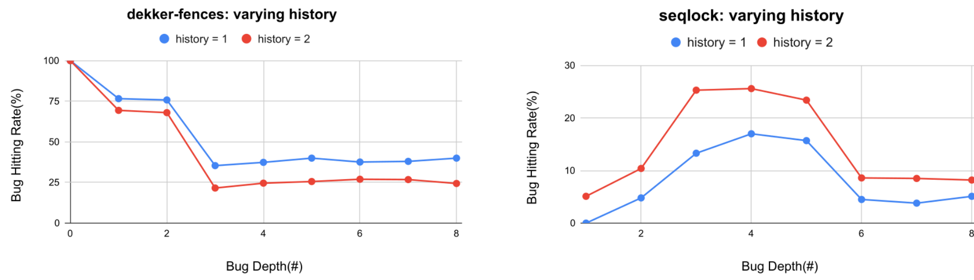
Also, we visualize how the bug hitting rate change for the same d and k_{com} in the Figure 7.5. We cannot say whether increasing or decreasing the value of history is good or not. A high history value extends the search space, which can find more bugs with different bug depths but also may let the PCTWM miss the critical behavior selection.

7.4 RQ3. Bug Detection Ability Comparison: C11Tester v.s. PCT v.s. PCTWM

C11Tester uses the bug hitting rate [51] to compare its effectiveness with the old tools. Similar to it, we demonstrate how PCTWM algorithms not only improve C11Tester's ability to discover bugs but also give the weak-memory guarantee. In Figure 7.2, we visualize their

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	14	5	13.9	20	21.9	24.5

Table 7.5: Data Structure Benchmark Bug Hitting Rate in 1000 rounds(%) - Varying History



(a) dekker (b) ms_queue

Figure 7.1: Bug Hitting Rate - Varying h in PCTWM

rate of hitting the same bug in one benchmark.

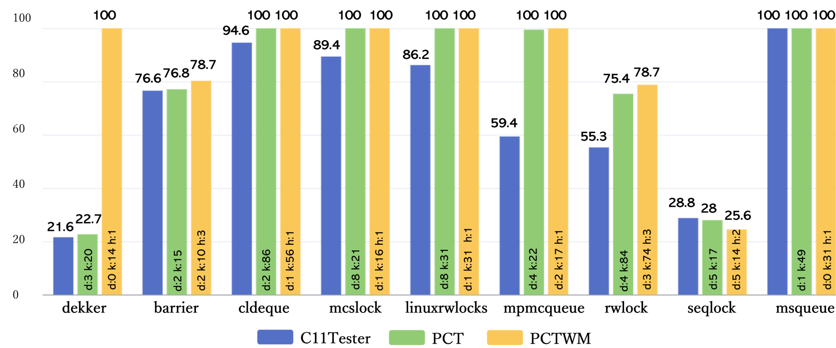


Figure 7.2: Bug Hitting Rate for All Nine Benchmarks

```
main thread:   flag1,flag2=false;
               turn=0;

thread 1                          || thread 2
flag1 = true;                     || flag2 = true;
                                  ||
while(flag2==true&&turn==0){      || while(flag1==true&&turn==1){
//busy wait                       || //busy wait
}                                 || }
                                  ||
// critical section               || // critical section
                                  ||
turn=1;                           || turn=0;
flag1=false;                      || flag2=false;
```

Figure 7.3: Benchmark:Dekkfer-fences

7.4.1 PCT vs C11Tester: Bug Detection Performance

PCT vs C11Tester We first compare the bug detection ability of the C11Tester with PCT scheduling with the default random scheduling. We can see that the PCT implementation gets a higher bug hitting rate in seven benchmarks than the default, five of which achieve obvious improvement. In this case, we define the *'obvious'* as a hitting rate of an improvement of more than 35% or approaching 100%. For benchmark *msqueue*, C11Tester, PCT, and PCTWM always perform 100% bug hitting because the injected bug is too easy to trigger.

7.4.2 PCTWM vs PCT: Bug Detection Performance

PCTWM vs PCT When we use PCTWM, seven benchmarks present a higher bug hitting rate while six of them are obvious. Moreover, a huge increment appears in the bug hitting rate of benchmark *dekker*. PCTWM can always find the bug in *dekker* and here we use the Figure 7.3 to show its logic and reason for this increase. Benchmark *dekker* is written based on the classic concurrency example - 'Peterson's Algorithm'[67]. It controls the execution with two boolean flags($flag1$ and $flag2$) and one integer variable($turn$) to make sure that only one thread can enter the critical section at one moment. The variables '$flag1flag2$' and '$turn$' can be viewed as the gate of critical section. They keep checking the condition in a while loop. In a SC model, the concurrency bug - two threads entering the critical section at the same time, will not happen as the variables '$flag1flag2$' and '$turn$' will only read from the latest write action, which is the write actions - writing true to $flag1flag2$ at the beginning of thread 1 and 2. However, in the weak memory model, if the memory order of read action is relaxed, the variable $flag1flag2$ may read from the initial write value on the main thread. To be more specific, the depth for this bug is zero, the possible read-from value - $flag1 = false$ or $flag2 = false$ is written at the beginning of the main thread. For PCT, as this algorithm does not control read-from value for read action, even we set the bug depth as zero. But for PCTWM, by setting d as zero, we can control the read actions to read locally. And the local-thread view for the flag on another thread($flag2$ for thread 1, $flag1$ for thread 2) is the initial global write action(writing false to $flag1flag2$). So, PCTWM can always hit this bug in *dekker*.

Also, the average bug hitting rate of these nine benchmarks are 67.9%, 78.1%, and 87.5%. The PCT and PCTWM algorithms improve the average bug hitting rate of the nine benchmarks by 16% and 29%, respectively.

In conclusion, both the PCT algorithm and the PCTWM algorithm improve the C11Tester's bug detection ability. The PCTWM algorithm provides a more significant improvement than the PCT algorithm. Though PCT improves the bug hitting rate, i.e., the bug detection ability, its theoretical guarantees do not apply to weak memory programs.

Benchmark	C11tester	PCT			PCTWM			
	rate(%)	*d*	*k*	rate(%)	*d*	*k*	*h*	rate(%)
dekker	21.6%	3	20	22.7	0	14	1	100
msqueue	100	1	49	100	0	31	1	100
barrier	76.6	2	15	77.1	2	10	3	78.7
cldeque	94.6	2	86	100	2	56	1	100
mcslock	89.4	8	26	100	1	16	1	100
linuxrwlocks	86.2	8	20	100	1	19	1	100
mpmcqueue	59.4	4	19	100	2	17	1	100
rwlock	55.3	4	84	75.4	3	74	3	78.7
seqlock	28.8	5	17	28.0	5	14	2	25.6
Average	67.9%	78.2%			87%			

Table 7.6: Hitting Rate of each benchmark(hitting data races in 1000 rounds). Higher data-race hitting rate and lower execution time are better.

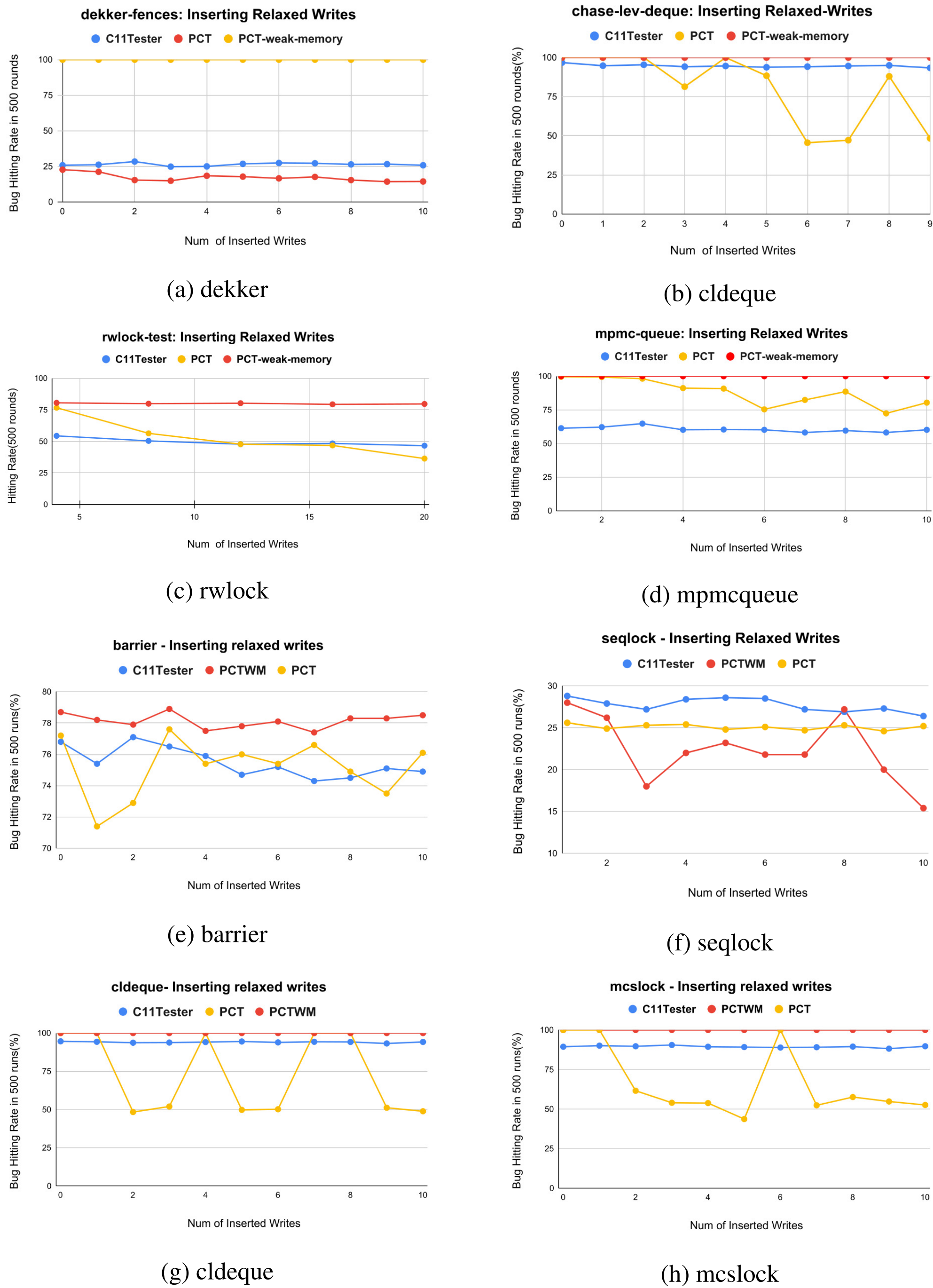
7.5 RQ4: PCTWM vs PCT: Bug Detection Performance

How does the amount of weak memory accesses affect the performance of PCTWM? In RQ4, 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 programs which does not affect the program behavior. Essentially it increases the number of visible writes to read-from for the read or RMW accesses.

In the Figure 7.4, we observe significant differences in the bug detection rate by 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. While 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.



(a) dekker (b) cldeque

(c) rwlock (d) mpmcqueue

(e) barrier (f) seqlock

(g) cldeque (h) mcslock

Figure 7.4: Bug Hitting Rate - Inserting Relaxed Writes

As the PCTWM algorithm also revises the definition of the shared access events to the communication events, we design this experiment to show the resilience of PCTWM's bug detection ability. In the Figure 7.4 we drew the trend of bug detection rate after inserting

more relaxed write actions to four benchmarks, which compared the robustness of the PCT and PCTWM algorithms considering that the concept of communication events is the key difference between the two. Here we only list inserting relaxed writes in four benchmarks because the hitting rates of some benchmarks do not differ a lot.

The bug detection capability of the C11Tester does not decrease a lot lies in the fact that the bug depth in all nine benchmarks is low, eight of them less than or equal to 2. In other words, the interleavings we need to trigger the bug are not very complex and the randomized testing can easily switch at these one or two critical points.

By incorporating the concept of communication events, we not only increase the worst-case probability of triggering the bug, but we also improve the algorithm's robustness. All read, write, and fence operations are counted as shared access events by the PCT method. The PCTWM also suggests defining communication events that can synchronize with other threads by filtering on their memory order type.

7.6 RQ5: Does or why PCTWM cause the overhead in the C11Tester?

7.6.1 Data Structure Benchmarks

To figure out the overhead introduced by the PCT and PCT algorithm, we record the average execution time of each run for the nine benchmarks over 1000 runs. The execution time for all nine data structure benchmarks increases after the implementation of the PCT and PCT algorithm. For the seven cds_modified_checker benchmarks, their average execution time is separately 3ms, 5.7ms, and 6.1ms in C11Tester, PCT and PCTWM algorithm. For the two benchmarks - *seqlock* and *rwlock*, after the PCT and PCTWM algorithm implementation, the execution time respectively increase about 19% and 29% compared to that of the C11Tester. So we conclude that the PCT and PCTWM do introduce the overhead in execution time when detecting the bug.

time/ms	C11tester	PCT	PCTWM
dekker	2	5	6
msqueue	4	6	6
barrier	4	7	7
cldeque	2	4	5
mcslock	3	5	6
mpmcqueue	4	7	7
linuxrwlocks	2	6	6
rwlock	10310	12780	13300
seqlock	10230	12530	12570

Table 7.7: Execution time of each Benchmark(average over 1000 rounds). The average execution time for running one benchmark over 1000 runs. Lower average execution time is better.

7.6.2 Real Applications Performance

Table 7.8 lists the performance assessment result for each application.

First, the C11Tester also implements the data race detection in these three applications and the PCTWM detected data race in all of them no matter single or multiple core-configuration.

Second, there is no obvious difference in the throughput result in Silo. For the other two using execution time as an assessment metric, the PCT and PCTWM algorithm costs more time. For the Mabain, the PCTWM takes around 10% more time and the PCTWM generates about 15% overhead in the execution time compared to the C11Tester. The overhead brought by PCTWM contains the following reasons: 1) PCTWM searching the highest-priority thread while C11Tester randomly generates a number in the enabled vector size; 2) PCTWM traverses backward for external read action; 3) PCTWM updating the visible vector for shared variables for each thread and communication events.

Furthermore, performance of three real applications does not vary between the single-core and multiple-core configurations. This result is due to the fact that all three versions - C11Tester, PCT, and PCTWM - can only run one thread at a time.

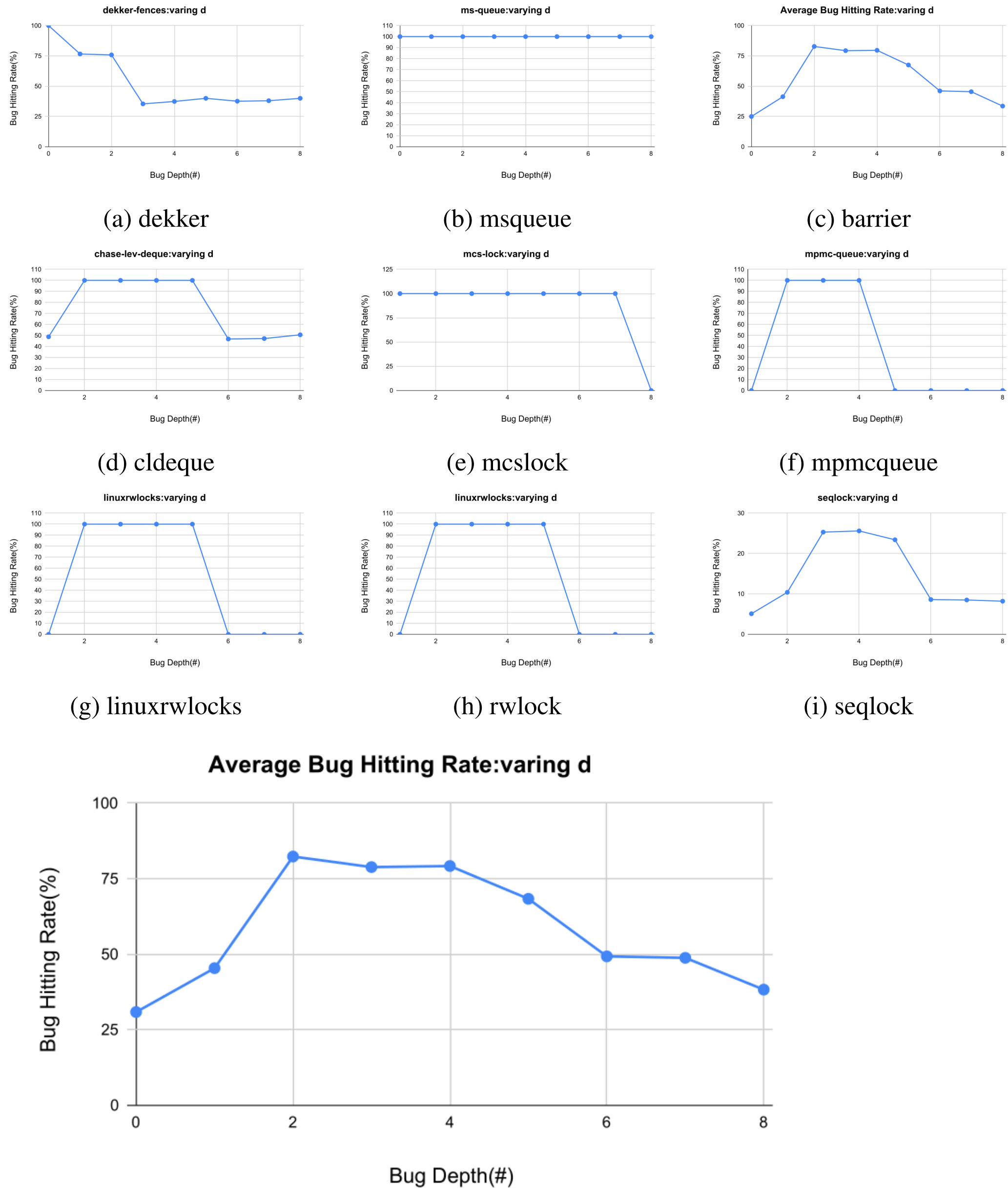
App	core	C11tester	PCT			PCTWM			
		result	d	k	result	d	k_{com}	h	result
Silo (ops/sec)	single	12428(0.58%)	40	200	12763(1.94%)	20	50	10	11039(7.38%)
	multiple	12760(0.61%)			12847(1.79%)				11387(6.92%)
Mabain (time/s)	single	7.73(1.56%)	30	300	8.92(2.51%)	2	50	2	8.43(4.11%)
	multiple	7.65(2.48%)			8.88(3.76%)				8.40(3.62%)
Iris (time/s)	single	10.98(2.02%)	30	100	12.86(3.65%)	25	60	15	12.79(4.78%)
	multiple	10.83(1.88%)			12.8938(4.57%)				12.43(6.59%)

Table 7.8: Real applications performance on single-core and multiple-core configuration. Performance results for application benchmarks in the single-core and multiple-core configurations(averaged over 10 runs). Parentheses include the relative standard deviation. Larger throughout is better for throughput-based measurements. Shorter execution time is better for time-based measurements.

7.7 RQ 6. Parameter's effect on PCTWM algorithm

In the RQ2, the bug hitting rate of some benchmarks increase after enhancing the bug depth. And in this section, we explores how the bug hitting rate changes if we further increase the bug depth.

According to the empirical experimental results in the PCT paper[21], the bug detection ability will first strengthen and then weaken when increasing the bug depth. This empirical observation is consistent with the experimental results in Table 7.4 as the 'turning point of bug depth' in each benchmark is different.



(a) dekker (b) msqueue (c) barrier

(d) cldeque (e) mcslock (f) mpmcqueue

(g) linuxrwlocks (h) rwlock (i) seqlock

(j) Average Bug Hitting Rate

Figure 7.5: Bug Hitting Rate - Varying d in PCTWM

Chapter 8

Related Work

In this chapter, we discuss the related technologies to our research project. The related work is divided into concurrency and consistency, concurrency bug types, concurrency testing, and techniques for detecting concurrency bugs in the weak memory model.

8.1 Concurrency and consistency

Memory consistency models play a crucial role in concurrent systems. Architectures [63, 4, 69, 5] exhibit weak memory concurrency behaviors due to various architectural features such as memory hierarchy, interconnect, and so on for performance reasons. 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 [52, 17, 8, 81, 68, 10, 11, 37, 42, 38, 25, 14, 28, 45, 53, 41]. In this paper we follow the C/C++ concurrency semantics adopted by C11Tester [49]. However, due to the subtle semantics of these primitives, writing weak memory concurrent programs are often difficult and error-prone. Therefore weak memory concurrency poses a significant challenge to testing and verification.

8.2 Concurrency Bugs Types

The bugs in multi-thread programs may result from i) unexpected inputs to the program; ii) the unexpected thread interleavings[85]. But only the latter type is called the concurrency bugs. For these two causes, the researchers design different kinds of automated testing tools, one for generating various random inputs[34, 74, 75, 22, 30] and the other for thread scheduling[84, 83]. Some testing tools combine the inputs and thread interleaving coverage together[87], and adopt effective methods to reduce the search space.

The concurrency bugs are hard to understand as they may result from complex and unexpected interactions of different components in the program. Lu et.al[50] classify them into deadlock and non-deadlock, deadlock bugs including atomicity violation, order violation, and the other bugs. Lu et. al[50] do not consider data races as data races might be benign. However, many researchers tend to focus on detecting data races in concur-

rent programs[76, 60, 7, 59]. Considering the severeness of data races in some cases, Wu et.al[85] proposes four types of concurrency bugs - data race, atomicity violation, order violation, and deadlock. The benchmarks in this project cover three types of concurrency bugs - data race, atomicity violations, and order violation.

8.3 Concurrency testing

When testing concurrent programs, the primary challenge is proving the validity of the realization as there are numerous cases, beyond the programmers' expectations. 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 [70], non-preemptive contexts [56], scheduler delays [31], and phases [18].

Randomized testing aims at detecting bugs by randomly generated test executions, and they are shown [79] to be effective in practice. The randomized partial order sampling algorithm [73] is designed to cover execution traces more uniformly than a pure random walk. The probabilistic concurrency testing (PCT) algorithm [21] improved the state of the art by providing a theoretical guarantee on the randomized algorithm. The parallel version of the PCT algorithm (PPCT) [58] allows the parallel execution of many threads instead of serializing them. The PCT algorithm designed for multi-thread programs with a set of totally ordered events is later extended to distributed systems [39, 65], to capture a more general partially ordered set of events. The partial order sampling (POS) algorithm [88] also provides theoretical probability bounds on the generated tests. PCT differs from the other probabilistic approaches as it guarantees a probability is not exponentially low in the number of program events, but only the bug depth, d. PCT achieves this by characterizing the depth of concurrency bugs and by bounding the sample set of executions to the set of executions with depth d.

8.4 Techniques for Detecting Concurrency Bugs in the Weak Memory Model

This project is based on C11Tester[51], an automatic testing tool. Besides testing tools, this thesis briefly introduces three techniques for guaranteeing the correctness of the program's execution in the weak memory model.

8.4.1 Model Checker

Model checker[26, 71], i.e., verification by state-space exploration[33], is the currently most often used technique to check the correctness of weak memory applications. This method

explores the set of possible executions that are reachable from the initial state and checks whether it meets any 'bad' state (i.e., one violating the semantics or specifications). The model-checking tools are typically expressed either in an axiomatic or an operational style.

The operational model-checker is an abstraction of actual machines, composed of idealized hardware components such as buffers and queues[4]. Susmit Sarkar[72] points out that the operational model is more intuitive than typical axiomatic models. Because the operational model abstracts from the actual hardware more directly as they usually have the notion of global time. Different from the operational model, the axiomatic model[62][19] distinguishes permitted behaviors from prohibited behaviors, typically by confining various memory access relations.

Most of the model-checkers are specific to finding the missed bugs in one weak memory model[6, 62, 63]. And some verification tools offer a generalized method[3], simulating the non-SC behaviors by transforming different inputs.

This method faces the difficulty of state-space explosion. That is to say, the concurrent program's numerous instructions and threads would generate massive executions, squandering time and resources. To tackle this problem, some researchers adopt various ways to reduce the search space[6, 1].

8.4.2 State Space Reduction

This technique can be viewed as an improvement for the model checker, aimed at addressing its biggest obstacle - the state space explosion problem.

Some researchers utilize the relaxation analysis[6][20][89][15] to reduce the size of state-space. Partial order reduction[66][32][27] is the most prominent. Some researchers noticed that many state transition graph-based model checkers are built on interleaving semantics, which can lead to rapid growth in the graph size. Researchers tried different ways to control the graph size, among which the Petri Nets[86][78][82] is the most prevalent due to its unfolding technique.

8.4.3 Fence-insertion Tool

Some researchers develop an automatic fence-insertion tool to ensure the correctness of concurrent systems under a weak memory model, considering that fences can ensure the correctness of many algorithms in the relaxed memory model, like non-blocking. However, the cost of fences is the high performance brought by relaxed architecture. In other words, programmers should use fences as few as possible. The double-edged sword characteristic makes memory fence placement difficult and error-prone because it needs nuanced reasoning about the underlying memory model.

Some scalable tools[40][48] place the fences by taking the memory model description, safety description, and a program as inputs, and then computing the constraints. Another kind of fence-insertion[47][2] tool is designed for a certain relaxed memory model.

Chapter 9

Conclusions and Future Work

This chapter gives an overview of the project's contributions. Then we draw some conclusions from the results and discuss some ideas for future work. Finally, I reflect on myself through this project.

9.1 Summary

In this thesis, we presented the Probabilistic Concurrency Testing for Weak Memory (PCTWM) algorithm for testing weak memory concurrency programs and provide 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. Similar to PCT, PCTWM achieves its strong theoretical guarantees on the lower bound on the probability of detecting bugs based on bounding the sample set of executions by the bug depth d.

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 weak memory programs with more relaxed write operations. We also show that the implementation of PCTWM does not cause significant overhead.

This thesis project is completed by the following procedure. We begin by reading and reviewing the literature on the C/C++11 weak memory model and the PCT algorithm. The PCT is then implemented on the most recent C/C++11 weak memory concurrency testing tool - C11Tester - and its effectiveness is compared to naive randomized testing. Based on this, we consider the PCTWM algorithm to further broaden and improve the PCT algorithm's bug detection ability following the characteristics of the C/C++11 weak memory model. Then, we put these two novel algorithms - PCT and PCTWM - to the test on data structure benchmarks and real-world applications to demonstrate their efficiency. Finally,

in this thesis, we theoretically propose the notion of PCTWM and demonstrate the effectiveness of this algorithm with experimental results.

9.2 Future work

Due to the limitation of time and resources, my thesis project still has some points that can be further improved in the future. First, the PCT and PCTWM algorithm relies on the parameters - k and k_{com}. However, manually estimating these events amount, no matter shared memory access or communication events, is time-wasted, especially for the large benchmarks. We will build an automatic tool to estimate the number of these events in each program. Second, *livelock*, we now set a bound for the events (scheduling) that have been processed and perform some naive randomized testing when the bound is met to leap out of the livelock, which requires more creative development. Third, we may further refactor the code and explore the cause of the overhead by perfecting the code as much as possible. Third, as we discussed in Chapter 8, concurrency bugs have different types and researchers are still exploring their causes. Our benchmarks cover three types of concurrency bugs in this thesis - data race, order violations, and assertion violations. In the future, we may test the effectiveness of PCTWM on more different types of concurrency bugs.

9.3 Self-Reflection

This 10-month project also taught me a lot on a personal level. First and foremost, this is the first time I have used English to do such a project, which significantly improves my reading and academic writing skills in English. Second, understanding open-source software and applying new techniques to it can be difficult at first as an implementer. However, this experience has greatly improved my coding skills. Third, this project provides a forum for me to collaborate with others, thereby improving my critical thinking and academic communication skills. To sum up, this project gives me great improvement in using English for academic writing, contributing to an open source repo, and research skills.

The coding always needs improvement. So I will be maintaining this project on GitHub on an ongoing basis. Considering my level of programming, I also think there are some areas of my code that are worth refactoring.

Through this project, I also have some reflections on the academic project. In the beginning, I was shy and could not well summarize my daily work in weekly meetings, which decrease my efficiency. Then I gradually understand how to proceed with the project and pick the key points to discuss in meetings with my supervisors. Academia is a path that requires strong willpower and patience. There may be unexpected situations, such as experiments that do not go as expected or problems that cannot be solved. But it is important to have strong willpower and an uncompromising spirit to overcome these challenges. Last but not the least, this thesis serves as an influential foundation for my future work.

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *ACM SIGPLAN Notices*, 49(1):373–384, 2014.

[2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under tso. In *Proceedings of the 19th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'13, page 530–536, Berlin, Heidelberg, 2013. Springer-Verlag. ISBN 9783642367410. doi: 10.1007/978-3-642-36742-7_37. URL `https://doi-org.tudelft.idm.oclc.org/10.1007/978-3-642-36742-7_37`.

[3] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *ESOP*, pages 512–532, 2013.

[4] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: modelling, simulation, testing, and data-mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. doi: 10.1145/2627752.

[5] Jade Alglave, Will Deacon, Richard Grisenthwaite, Antoine Hacquard, and Luc Maranget. Armed cats: Formal concurrency modelling at arm. *ACM Trans. Program. Lang. Syst.*, 43(2), 2021. doi: 10.1145/3458926.

[6] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting rid of store-buffers in tso analysis. In *International Conference on Computer Aided Verification*, pages 99–115. Springer, 2011.

[7] Utpal Banerjee, Brian Bliss, Zhiqiang Ma, and Paul Petersen. A theory of data race detection. In *Proceedings of the 2006 workshop on Parallel and distributed systems: testing and debugging*, pages 69–78, 2006.

[8] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In *POPL'11*, pages 55–66. ACM, 2011. doi: 10.1145/1926385.1926394.

[9] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *ACM SIGPLAN Notices*, 46(1):55–66, 2011.

[10] Mark Batty, Kayvan Memarian, Kyndylan Nienhuis, Jean Pichon-Pharabod, and Peter Sewell. The problem of programming language concurrency semantics. In *ESOP'15*, pages 283–307, 2015. doi: 10.1007/978-3-662-46669-8_12.

[11] Mark Batty, Alastair F. Donaldson, and John Wickerson. Overhauling SC atomics in C11 and OpenCL. In *POPL '16*, pages 634–648. ACM, 2016. doi: 10.1145/2837614.2837637.

[12] Mark Batty, Alastair F Donaldson, and John Wickerson. Overhauling sc atomics in c11 and opencl. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 634–648, 2016.

[13] Pete Becker et al. Iso/iec 14882: 2011: Programming languages–c++(final draft international standard). Technical report, Technical Report, 2011.

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

[15] Swarnendu Biswas, Minjia Zhang, Michael D. Bond, and Brandon Lucia. Valor: Efficient, software-only region conflict exceptions. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, page 241–259, New York, NY, USA, 2015. Association for Computing Machinery. ISBN 9781450336895. doi: 10.1145/2814270.2814292. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/2814270.2814292`.

[16] Hans-J. Boehm. Can seqlocks get along with programming language memory models? In *Proceedings of the 2012 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '12, page 12–20, New York, NY, USA, 2012. Association for Computing Machinery. ISBN 9781450312196. doi: 10.1145/2247684.2247688. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/2247684.2247688`.

[17] Hans-J. Boehm and Sarita V. Adve. Foundations of the C++ concurrency memory model. In *PLDI'08*, 2008. doi: 10.1145/1375581.1375591.

[18] Ahmed Bouajjani and Michael Emmi. 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*, pages 451–465, 2012.

[19] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *International Colloquium on Automata, Languages, and Programming*, pages 428–440. Springer, 2011.

[20] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *International Conference on Computer Aided Verification*, pages 107–120. Springer, 2008.

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

[22] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[23] cbloom rants. Mcs list-based lock. `http://cbloomrants.blogspot.com/2011/07/07-18-11-mcs-list-based-lock_18.html`, 2012.

[24] cbloom rants. A look at some bounded queues. `http://cbloomrants.blogspot.com/2011/07/07-30-11-look-at-some-bounded-queues.html`, 2012.

[25] Soham Chakraborty and Viktor Vafeiadis. Grounding thin-air reads with event structures. 3(POPL), 2019. doi: 10.1145/3290383.

[26] Edmund M. Clarke, E Allen Emerson, and A Prasad Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 8(2):244–263, 1986.

[27] Edmund M Clarke, Orna Grumberg, Marius Minea, and Doron Peled. State space reduction using partial order techniques. *International Journal on Software Tools for Technology Transfer*, 2(3):279–287, 1999.

[28] Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proceedings of the ACM on Programming Languages*, 4: 1–29, 12 2019. doi: 10.1145/3371102.

[29] Changxue Deng. Mabain: A fast and light-weighted key-value store library. `https://github.com/chxdeng/mabain`, 2018.

[30] Dongdong Deng, Wei Zhang, and Shan Lu. Efficient concurrency-bug detection across inputs. *Acm Sigplan Notices*, 48(10):785–802, 2013.

[31] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamaric. 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*, pages 411–422, 2011.

[32] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Springer, 1996.

[33] Patrice Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 174–186, New York, NY, USA, 1997. Association for Computing Machinery. ISBN 0897918533. doi: 10.1145/263699.263717. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/263699.263717`.

[34] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[35] ISO/IEC 14882. Programming language C++, 2011.

[36] ISO/IEC 9899. Programming language C, 2011.

[37] Alan Jeffrey and James Riely. On thin air reads: Towards an event structures model of relaxed memory. In *LICS'16*. ACM/IEEE, 2016. doi: 10.1145/2933575.2934536.

[38] Jeehoon Kang, Chung-Kil Hur, Ori Lahav, Viktor Vafeiadis, and Derek Dreyer. A promising semantics for relaxed-memory concurrency. In *POPL'17*, POPL 2017, page 175–189, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450346603. doi: 10.1145/3009837.3009850. URL `https://doi.org/10.1145/3009837.3009850`.

[39] Burcu Kulahcioglu Ozkan, Rupak Majumdar, Filip Niksic, Mitra Tabaei Befrouei, and Georg Weissenbacher. Randomized testing of distributed systems with probabilistic guarantees. *PACMPL*, 2(OOPSLA):160:1–160:28, 2018.

[40] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, jun 2012. ISSN 0163-5700. doi: 10.1145/2261417.2261438. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/2261417.2261438`.

[41] Ori Lahav and Udi Boker. What's decidable about causally consistent shared memory? *ACM Trans. Program. Lang. Syst.*, 44(2), apr 2022. doi: 10.1145/3505273.

[42] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. In *PLDI 2017*, pages 618–632, 2017. doi: 10.1145/3062341.3062352. Technical Appendix Available at `https://plv.mpi-sws.org/scfix/full.pdf`.

[43] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Transactions on Computers c-28*, 9:690–691, 1979.

[44] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. *ACM SIGPLAN Notices*, 48(8): 69–80, 2013.

[45] Sung-Hwan Lee, Minki Cho, Anton Podkopaev, Soham Chakraborty, Chung-Kil Hur, Ori Lahav, and Viktor Vafeiadis. Promising 2.0: Global optimizations in relaxed memory concurrency. In *PLDI 2020*, page 362–376, 2020. doi: 10.1145/3385412.3386010.

[46] Christopher Lidbury and Alastair F. Donaldson. Dynamic race detection for C++11. In *POPL 2017*, pages 443–457, 2017. doi: 10.1145/3009837.3009857.

[47] Alexander Linden and Pierre Wolper. A verification-based approach to memory fence insertion in relaxed memory systems. In *International SPIN Workshop on Model Checking of Software*, pages 144–160. Springer, 2011.

[48] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. *SIGPLAN Not.*, 47(6):429–440, jun 2012. ISSN 0362-1340. doi: 10.1145/2345156.2254115. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/2345156.2254115`.

[49] Nian Liu, Binyu Zang, and Haibo Chen. No barrier in the road: A comprehensive study and optimization of arm barriers. In *PPOPP'20*, page 348–361, 2020. doi: 10.1145/3332466.3374535.

[50] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: a comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, pages 329–339, 2008.

[51] Weiyu Luo and Brian Demsky. C11tester: a race detector for c/c++ atomics. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 630–646, 2021.

[52] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In *POPL '05*. ACM, 2005.

[53] Roy Margalit and Ori Lahav. Verifying observational robustness against a c11-style memory model. *Proc. ACM Program. Lang.*, 5(POPL), 2021. doi: 10.1145/3434285.

[54] John M Mellor-Crummey and Michael L Scott. Synchronization without contention. *ACM SIGPLAN Notices*, 26(4):269–278, 1991.

[55] Maged M Michael and Michael L Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 267–275, 1996.

[56] Madanlal Musuvathi and Shaz Qadeer. 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*, pages 446–455, 2007.

[57] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. *ACM Sigplan Notices*, 42(6):446–455, 2007.

[58] Santosh Nagarakatte, Sebastian Burckhardt, Milo M. K. Martin, and Madanlal Musuvathi. 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*, pages 543–554, 2012. doi: 10.1145/2254064.2254128. URL `https://doi.org/10.1145/2254064.2254128`.

[59] Robert HB Netzer and Barton P Miller. Improving the accuracy of data race detection. In *Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 133–144, 1991.

[60] Robert HB Netzer and Barton P Miller. What are race conditions? some issues and formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, 1 (1):74–88, 1992.

[61] Brian Norris and Brian Demsky. 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*, pages 131–150, 2013.

[62] Peizhao Ou and Brian Demsky. Checking concurrent data structures under the c/c++11 memory model. *SIGPLAN Not.*, 52(8):45–59, jan 2017. ISSN 0362-1340. doi: 10.1145/3155284.3018749. URL `https://doi-org.tudelft.idm.oclc.org/10.1145/3155284.3018749`.

[63] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In *ECOOP*, pages 478–503, 2010.

[64] Scott Owens, Susmit Sarkar, and Peter Sewell. A better x86 memory model: x86-tso. In *International Conference on Theorem Proving in Higher Order Logics*, pages 391–407. Springer, 2009.

[65] Burcu Kulahcioglu Ozkan, Rupak Majumdar, and Simin Oraee. Trace aware random testing for distributed systems. *Proc. ACM Program. Lang.*, 3(OOPSLA):180:1–180:29, 2019.

[66] Doron Peled. All from one, one for all: on model checking using representatives. In *International Conference on Computer Aided Verification*, pages 409–423. Springer, 1993.

[67] Gary L. Peterson. Myths about the mutual exclusion problem. 1981.

[68] Jean Pichon-Pharabod and Peter Sewell. A concurrency semantics for relaxed atomics that permits optimisation and avoids thin-air executions. In *POPL 2016*, pages 622–633, 2016. doi: 10.1145/2837614.2837616.

[69] Christopher Pulte, Shaked Flur, Will Deacon, Jon French, Susmit Sarkar, and Peter Sewell. Simplifying ARM concurrency: multicopy-atomic axiomatic and operational models for ARMv8. *PACMPL*, 2(POPL):19:1–19:29, 2018. doi: 10.1145/3158107.

[70] Shaz Qadeer and Jakob Rehof. 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*, pages 93–107, 2005.

[71] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *International Symposium on programming*, pages 337–351. Springer, 1982.

[72] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding power multiprocessors. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, pages 175–186, 2011.

[73] Koushik Sen. 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*, pages 323–332, 2007. doi: 10.1145/1321631.1321679. URL `https://doi.org/10.1145/1321631.1321679`.

[74] Koushik Sen and Gul Agha. Automated systematic testing of open distributed programs. In *International Conference on Fundamental Approaches to Software Engineering*, pages 339–356. Springer, 2006.

[75] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5):263–272, 2005.

[76] Konstantin Serebryany and Timur Iskhodzhanov. Threadsanitizer: data race detection in practice. In *Proceedings of the workshop on binary instrumentation and applications*, pages 62–71, 2009.

[77] and Eddie Kohler Stephen Tu, Wenting Zheng. Silo: Multicore in-memorystorage engine. `https://github.com/stephentu/silo`, 2013.

[78] Jiaquan Sun, Guanjun Liu, Dongming Xiang, and Changjun Jiang. A petri-net-based method for detecting bugs in multiple threads. In *2019 IEEE 16th International Conference on Networking, Sensing and Control (ICNSC)*, pages 150–156, 2019. doi: 10.1109/ICNSC.2019.8743177.

[79] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: an empirical study. In José E. Moreira and James R. Larus, editors, *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP '14, Orlando, FL, USA, February 15-19, 2014*, pages 15–28. ACM, 2014.

[80] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. Speedy transactions in multicore in-memory databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 18–32, 2013.

[81] Viktor Vafeiadis. Formal reasoning about the c11 weak memory model. In *Proceedings of the 2015 Conference on Certified Programs and Proofs*, pages 1–2, 2015.

[82] Antti Valmari. Stubborn sets for reduced state space generation. In *International Conference on Application and Theory of Petri Nets*, pages 491–515. Springer, 1989.

[83] Zan Wang, Dongdi Zhang, Shuang Liu, Jun Sun, and Yingquan Zhao. Adaptive randomized scheduling for concurrency bug detection. In *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 124–133. IEEE, 2019.

[84] Zhendong Wu, Kai Lu, Xiaoping Wang, and Xu Zhou. Collaborative technique for concurrency bug detection. *International Journal of Parallel Programming*, 43(2): 260–285, 2015.

[85] Zhendong Wu, Kai Lu, and Xiaoping Wang. Surveying concurrency bug detectors based on types of detected bugs. *Science China Information Sciences*, 60(3):1–27, 2017.

[86] Dongming Xiang, Guanjun Liu, Chungang Yan, and Changjun Jiang. Detecting data inconsistency based on the unfolding technique of petri nets. *IEEE Transactions on Industrial Informatics*, 13(6):2995–3005, 2017. doi: 10.1109/TII.2017.2698640.

[87] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications*, pages 485–502, 2012.

[88] Xinhao Yuan, Junfeng Yang, and Ronghui Gu. 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*, pages 317–335, 2018.

[89] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. *SIGPLAN Not.*, 50(6):250–259, jun 2015. ISSN 0362-1340. doi: 10.1145/2813885.2737956. URL https://doi-org.tudelft.idm.oclc.org/10.1145/2813885.2737956.

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

Appendix A

Glossary

A.1 Experiment Set Up

A.1.1 Set Up Environment

The experiments in this thesis are from three tools - C11Tester, C11Tester with PCT and C11Tester with PCTWM.

- Download and install the vagrantBox.
- Git clone the C11Tester in the varagntBox.
- Git clone the PCT-version C11Tester and PCTWM-version C11Tester in the vagrantBox. From the GitHub link `https://github.com/GMYMingyu/C11_PCT_PCTWM.git`.

A.1.2 Run the Script

The detailed instruction to run the script will be kept updating in the Readme file of the GitHub Repo `https://github.com/GMYMingyu/C11_PCT_PCTWM.git`.

Run the Data Structure Benchmark

- Go to the directory *c11tester-benchmarks/cds_check_modified*. Run the run_all.sh file. *c11tester-benchmarks/tasn11_missingbug*. Run the run_all.sh file.

Run the Real Application Benchmark

- Go to the directory *c11tester-benchmarks*. Run the run_all.sh file.

Appendix B

Requirements and Guidelines

This chapter details some requirements and guidelines for MSc theses submitted to the Software Engineering Research Group.

B.1 Requirements

B.1.1 Layout

- Your thesis should contain the formal title pages included in this document (the page with the TU Delft logo and the one that contains the abstract, student id and thesis committee). Usually there is also a cover page containing the thesis title and the author (this document has one) but this can be omitted if desired.
- The final thesis and drafts submitted for reviewing should be printed double-sided on A4 paper.

B.1.2 Content

- The thesis should contain the following chapters:
  - Introduction.
    Describes project context, goals and your research question(s). In addition it contains an overview of how (the remainder of) your thesis is structured.
  - One or (usually) more "main" chapters.
    Present your work, the experiments conducted, tool(s) developed, case study performed, etc.
  - Overview of Related Work
    Discusses scientific literature related to your work and describes how those approaches differ from what you did.
  - Discussion/Evaluation/Reflection
    What went well, what went less well, what can be improved?

  - Conclusions, Contributions, and (Recommendations for) Future Work
  - Bibliography

B.1.3 Bibliography

- Make sure you've included all required data such as journal, conference, publisher, editor and page-numbers. When you're using BIBTEX, this means that it won't complain when running `bibtex your-main-tex-file`.
- Make sure you use proper bibliographic references. This especially means that you should avoid references that **only** point at a website and not at a printed publication.

  For example, it's OK to add a URL with the entry for an article describing a tool to point at its homepage, but it's not OK to just use the URL and not mention the article.

B.2 Guidelines

- The main chapters of a typical thesis contain approximately 50 pages.
- A typical thesis contains approximately 50 bibliographic references.
- Make sure your thesis structure is balanced (check this in the table of contents).

  Typically the main chapters should be of equal length. If they aren't, you might want to revise your structure by merging or splitting some chapters/sections.

  In addition, the (sub)section hierarchies with the chapters should typically be balanced and of similar depth. If one or more are much deeper nested than others in the same chapter this generally signals structuring problems.
- Whenever you submit a second draft, include a short text which describes the changes w.r.t. the previous version.