Imperial College London
Department of Computing

Practical systematic concurrency testing for concurrent and distributed software

Paul Thomson

October 2016

Supervised by Alastair F. Donaldson

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Computing of Imperial College London
and the Diploma of Imperial College London

Declaration

This thesis and the work it presents are my own except where otherwise acknowledged.

Paul Thomson

Abstract

Systematic concurrency testing (SCT) is a promising solution to finding and reproducing concurrency bugs. The program under test is repeatedly executed such that a particular schedule is explored on each execution. Numerous techniques have been proposed to make SCT scalable. Despite this, we have identified the following open problems: (1) There is a major lack of comparison and empirical evaluation of SCT techniques; (2) There is a need for better reduction techniques that go beyond the current theoretical limits; (3) The feasibility of applying SCT in practice is unclear, particularly for distributed systems. This thesis makes the following contributions to the field of SCT:

- An independent, reproducible empirical study of existing SCT techniques over 49 buggy concurrent software benchmarks. Surprisingly, we found that the "naïve" controlled random scheduler performs well, finding more bugs than preemption bounding. We report the results for all techniques. We discuss the benchmarks and challenges faced in applying SCT.

- The *lazy happens-before relation* (lazy HBR), which provides reduction *beyond* partial-order reduction for programs that use mutexes. Our evaluation over 79 publicly available benchmarks shows both a large *potential* and large *practical* improvement from exploiting the lazy HBR.

- A description of how to create an SCT tool in practice, with a focus on subtle-yet-important details that are typically not discussed in prior work.

- A case study where we apply SCT in the context of distributed systems written for *Azure Service Fabric* (Fabric). We introduce our Adara actors framework for writing portable, *statically-typed* actors. We describe our model of Fabric and evaluate it on a system containing 15 bugs, showing that our Fabric model includes enough behaviours/asynchrony to expose these subtle pitfalls.

Acknowledgements

First and foremost, I would like to thank my supervisor, Alastair F. Donaldson, for his constant support and enthusiasm over the years. He introduced to me to software verification and testing. He has taught me so much. His pet peeves have become my pet peeves! I consider him an excellent role model, mentor, and friend.

I thank Cristian Cadar and Paul H. J. Kelly for their support and kindness, particularly towards the beginning of my PhD. I thank Shaz Qadeer and Madan Musuvathi for their positive comments and feedback over the years; I would probably not even know about systematic concurrency testing if it wasn't for their seminal work in this field. I thank Akash Lal and Shaz Qadeer (again) for hosting my MSR internships; the former internship laid the foundations of Chapter 6.

From the Multicore Programming Group, I thank Pantazis Deligiannis for being a good friend and for his work on P#; our collaborations have led to many further opportunities for me. I thank Adam Betts and Nathan Chong (the other "founding" members) for being good friends, and, of course, the later members too: Jeroen Ketema, Daniel Liew, Ethel Bardsley, John Wickerson, Tyler Sorensen, Andrei Lascu, and Christopher Lidbury.

I thank the EPSRC for funding my PhD via an EPSRC DTA studentship. I relied on the Imperial College High Performance Computing service;[1] the HPC cluster was critical to performing studies on this scale.

I thank my parents for everything they have given me, and Tor for her love and support.

[1] `https://www.imperial.ac.uk/admin-services/ict/self-service/research-support/hpc/hpc-service-support/service/`

Contents

List of Tables

List of Figures

1 Introduction

The age of ever increasing clock speeds is over and we have entered the multicore revolution [SL05]. In order to benefit from the growing importance of multicore and distributed systems, programs must be concurrent. Unfortunately, concurrent programming is hard; unexpected interactions between concurrent threads can lead to *concurrency bugs*—bugs that may or may not occur depending on the thread schedule. Traditional testing techniques are ineffective at finding and reproducing concurrency bugs due to this nondeterminism [KLVU10].

One successful approach for finding and reproducing concurrency bugs is *systematic concurrency testing* (SCT) [God97, MQB+08, YCG08, EQR11] which is the focus of this thesis. The technique involves repeatedly executing a concurrent program with fixed inputs, forcing a particular schedule to be explored on each execution. The approach is appealing as the analysis is highly automatic, has no false-positives, and supports reproduction of bugs by replaying the bug-inducing schedule. The straightforward approach is to use a depth-first search (DFS) to exhaustively explore *all* schedules. However, this does not scale to large programs as the number of schedules is exponential in the number of execution steps, and a partial exploration does not provide any useful coverage guarantees. Thus, researchers have proposed a number of alternative approaches to reduce the number of schedules that need to be explored during SCT, such as *schedule bounding* [MQ07b, EQR11], *partial-order reduction* (POR) techniques [God96, FG05, MQ07a, AAJS14], and *heuristic/randomisation-based* techniques [BKMN10, NBMM12, YNPP12].

Despite the successful work in this area, we have identified three important open problems which we address in this thesis:

1. There is a major lack of comparison and empirical evaluation of current SCT techniques, particularly in terms of how the techniques compare with each other at finding the same concurrency bugs. Baseline techniques used in prior work [MQ07b, BKMN10] include a straightforward depth-first search and (non-systematic) perturbing of the OS scheduler, which fails to address how other SCT techniques compare. Much work in this area also uses a set of proprietary concurrent benchmarks that

are not publicly available; thus, despite the apparent significance of previous findings, the claims made have not been independently validated on a different set of benchmarks. In Chapter 3, we address this by presenting a large empirical study of existing SCT techniques on a set of 49 buggy concurrent software benchmarks drawn from public code bases.

2. There is a real need for better reduction techniques that reduce the schedule-space by going *beyond* what is possible when using POR [God96]; although POR has clear benefits, there are still certain sets of equivalent schedules that it cannot reduce. In Chapter 4, we introduce the *lazy happens-before relation* that achieves significant reduction (beyond POR) for programs that use mutexes.

3. The challenges of applying SCT in practice are not entirely clear as they are not discussed in detail in prior work. In particular, distributed systems typically remain out-of-reach. The fundamental steps to create an SCT tool are also not discussed in depth. We attempt to address this as follows. In Chapter 3, we describe the issues encountered when trying to apply SCT during our empirical study. In Chapter 5, we describe the implementation of a novel Java SCT tool, including both high-level details that are widely-applicable to any such tool and low-level details that are specific to Java. In Chapter 6, we describe a large case study where we apply SCT in the context of distributed systems—arguably one of the most challenging settings in which to apply SCT.

1.1 Contribution

In Chapter 2, we introduce systematic concurrency testing and then formally define our concurrent program model. We continue with the four main contributions of this thesis:

- Chapter 3 describes our empirical study of existing SCT techniques. We gathered 49 buggy concurrent software benchmarks, drawn from public code bases, which we call SCTBench. We applied a modified version of an existing concurrency testing tool to SCTBench, comparing five SCT techniques in terms of their bug finding ability: depth-first search, preemption bounding, delay bounding, a controlled random scheduler, and probabilistic concurrency testing (PCT). Surprisingly, we found that the "naïve" controlled random scheduler, which randomly chooses one thread to execute at each scheduling point, performs well, finding more bugs than preemption bounding. We report the results for all techniques. We discuss the benchmarks and

the challenges we faced in applying SCT. We have made SCTBench and our tools publicly available for reproducibility and use in future work.

- Chapter 4 describes our *lazy happens-before relation*, which provides reduction *beyond* partial-order reduction for programs that use mutexes. We prove that schedules with identical lazy HBRs reach the same state and present two reduction techniques backed by the approach: *lazy HBR caching* and *lazy dynamic partial-order reduction*. We implemented these methods in JESS, our new SCT tool for Java programs, and present an evaluation over 79 publicly available benchmarks. Our evaluation shows both a large *potential* and large *practical* improvement from exploiting the lazy HBR.

- Chapter 5 describes how to create an SCT tool in practice based on our experience building JESS, including both high-level details that are widely-applicable to any such tool and low-level details that are specific to our setting (Java programs), with a focus on subtle-yet-important details that are not discussed in prior work. As a part of this, we present our race detection algorithm that we believe is more efficient than prior work.

- Chapter 6 describes our case study where we apply SCT in the context of distributed systems, arguably the most challenging scenario for SCT. We target systems written for *Azure Service Fabric* (Fabric) [Fam15] by creating a model of Fabric. We introduce *Adara actors*, our framework for writing portable, *statically-typed* actors, which we use in our model. We evaluate our model on a system containing 11 real bugs, plus 4 injected bugs that we believe are representative of subtle mistakes that developers are likely to make when using Fabric. We found 14 of the 15 bugs using SCT, including all of the injected bugs, showing that our Fabric model includes enough behaviours/asynchrony to expose these subtle pitfalls.

1.2 Work published during the PhD

Chapter 3 was originally published as a PPoPP paper [TDB14] that won best student paper award and was then invited to a special issue of the ACM Transactions on Parallel Computing journal [TDB16]. The lazy happens-before relation, included in Chapter 4, was described in a PPoPP short paper [TD15]. Aspects of the actor-based case study of Chapter 6 were described in a FAST paper [DMT+16]. The author contributed to earlier SCT experiments for the P# actor-based framework (used initially in Chapter 6) in a

PLDI paper [DDK+15]. The author also contributed to two further papers on data race analysis for GPU kernels [BCD+12, BCD+15]; this work helped inform the ideas in this thesis, but is not included here.

2 Background

In this chapter we introduce systematic concurrency testing (§2.1), including common terminology and techniques, before describing our model of a concurrent program (§2.2) which abstracts away programming language specifics and is assumed in the remaining chapters. We finally describe some common operations (§2.3) as examples to give context to the model.

2.1 Systematic concurrency testing

Systematic concurrency testing (SCT) [God97, MQB+08, YCG08, EQR11], also known as *stateless model checking* [God97], is a technique for finding and reproducing concurrency bugs. It tests a target program (or procedure) that, typically, must terminate in finite time and be deterministic modulo scheduling nondeterminism.[1] The program is executed repeatedly and a precise schedule (interleaving of operations) is forced each time. This process continues until all schedules have been explored, some resource budget (typically time or number of schedules) is reached, or some coverage requirement has been met. Unlike traditional stateful model checking [CE81, QS82], the system under test does not need to be modelled in a modelling language; instead, the original program is executed. Furthermore, the states of the program do not need to be captured (capturing the state of an unmodified program can be nontrivial). The approach is appealing as the analysis is highly automatic, has no false-positives, and supports reproduction of bugs by replaying the bug-inducing schedule. Other techniques, such as perturbing the OS scheduler by inserting calls to `sleep` (possibly with randomisation) or other similar functions [EFN+02, PLZ09, YNPP12] do not provide precise control over the schedule that is executed, bug reproducibility, nor coverage guarantees.

[1] In this thesis, we frequently simplify the presentation by assuming the target program terminates in finite time, has no inputs (e.g. by choosing fixed values for all inputs), and is sequentially consistent [Lam79]. These assumptions are not required in general for SCT and typically do not prevent its application in practice. We revisit these assumptions throughout this section.

History SCT was pioneered by Godefroid's VeriSoft tool [God97]. Musuvathi et al. later created the CHESS tool [MQ07b, MQ08, MQB+08] which supported SCT of C/C++ and C# programs on Microsoft Windows. The work on CHESS introduced several new SCT techniques such as preemption bounding [MQ07b] and delay bounding [EQR11] which we describe below. Numerous SCT research has been conducted since [YCG08, CBM10, WSG11, YNPP12, CMM13, AAJS14].

Depth-first search SCT traditionally performs a depth-first search (DFS) of the schedule-space and SCT tools like CHESS still support this baseline approach. More advanced techniques like preemption bounding and dynamic partial-order reduction are also still based on a DFS. The key advantage of a DFS is that it efficiently ensures that a different schedule is explored on each execution, resulting in an eventual exhaustive search of the considered schedule-space; unexplored schedule prefixes can be efficiently stored in a stack data structure. Exploring schedules exhaustively without using a DFS is challenging due to the space required for storing unexplored parts of the schedule-tree [CBM10]. As is common in prior work, in this thesis we assume that the state-space is acyclic and thus all schedules are finite. This can be enforced (in a pragmatic but unsound manner) using a per-schedule time limit. Prior work has shown that cyclic state-spaces can be explored exhaustively using SCT in practice [MQ08], as long as the programmer calls a recognised *yield* function to indicate when a thread is not making progress; we use a similar technique in this thesis (see §3.3 and §6.6.2).

Controlling execution via instrumentation Controlling the schedule can be achieved by instrumenting the program so that threads are blocked by the SCT tool before each operation; execution is serialised so that only one thread executes at a time and the choice of which thread to execute next is controlled by the SCT tool. Thus, SCT is typically applied without making changes to the OS scheduler. We cover details of implementing an SCT tool in Chapter 5.

Common terminology SCT tools interleave threads at their *visible operations* [God97]—such as reading from or writing to a global variable or locking a mutex. We formalise this in §2.2. Researchers refer to a thread executing a single visible operation as a *step*, a *transition*, an *event*, or simply an *operation*. Threads are typically blocked immediately before each visible operation by the SCT tool at which point a thread is chosen to be released; these points are called *scheduling points* [MQ07b]. The simplest way of representing a *schedule* is as a list of thread identifiers (thread ids), such that each thread is executed

in sequence from the start of the program/procedure for one step. Thus, the schedule $\langle T1, T1, T2 \rangle$ represents executing thread 1 for two steps, followed by thread 2 for one step. Since the target program is deterministic (modulo schedule nondeterminism), the same schedule will always reach the same program state. We formalise schedules in §2.2.

Concurrency bugs In this thesis (and as is often the case in SCT research), we consider *bugs* to be safety property violations, such as deadlocks, assertion failures, crashes, and uncaught exceptions. In particular, we rely on assertions already present in the target program or we may add assertions to check specific properties. *Concurrency bugs* are bugs that may or may not occur depending on the schedule; if a particular bug occurs on *every* possible schedule then we do not consider it a concurrency bug. In Chapter 6, we encounter some simple liveness bugs where the target program enters an infinite loop; we detect these simply by enforcing a step limit. We do not encounter more complex liveness bugs in this thesis.

SCT vs. controlled scheduling We note that the meaning of *systematic* concurrency testing is not well-defined; systematic can be synonymous with never executing a schedule more than once, leading to an exhaustive exploration (or, at least, an exhaustive exploration of all *considered* schedules), or it can simply mean controlling precisely which schedules will be executed resulting in a deterministic exploration that might repeat schedules and does not necessarily "complete". In both cases, the program must be deterministic and the scheduling decisions are controlled. In this thesis, we assume the latter definition which means more techniques are regarded as SCT. For example, using a random scheduler that randomly picks a thread and allows it to execute for one step can be still be regarded as SCT. Perturbing the OS scheduler via random calls to `sleep` is *not* SCT as the process is nondeterministic.

Nondeterminism due to inputs and data races In this thesis, we fix the inputs (e.g. command line arguments, input parameters) of all programs or procedures that we test. However, it is possible to consider different inputs or even enumerate all inputs, although it is likely infeasible to explore all inputs in a reasonable time limit. As in prior work [God97, FG05, YNPP12], in this thesis we assume sequential consistency [Lam79] by only considering interleavings of operations from different threads as if executing on a single-core processor, where a write by a thread is immediately visible to all other threads.

Thus, we do not consider the effects of data races[2] when executing under relaxed memory models (e.g. [SSO+10]). This assumption is not always as limiting as it may seem; developers often write their programs to be data race free (and free from explicit weak memory operations in languages like C++11), relying on synchronisation primitives that ensure sequentially consistent behaviour. In these cases, it is only necessary to consider interleavings of synchronising operations, like locking a mutex,[3] to be able to to find all safety property violations [God97]. Furthermore, it is possible to use a data race detector [FF09] during SCT, ensuring that the approach is sound for programs that contain data races, as long as detected data races are regarded as bugs; the resulting behaviours from racy executions are not explored, but these executions are all regarded as buggy anyway since they contain data races. We discuss how we handle data races in our empirical study in §3.6 and discuss some work that handles data races under relaxed memory models in §7.2.

SCT techniques Exploring all schedules of a program is typically infeasible as the number of schedules is exponential in the number of steps in an execution. Thus, researchers have proposed several different SCT techniques to try to minimise the number of schedules executed before a bug is found. We describe several key techniques that we consider in this thesis:

- **Schedule bounding techniques** [MQ07b, EQR11] perform a bounded DFS: preemption bounding [MQ07b] restricts the DFS to only schedules with fewer than c preemptive context switches; delay bounding considers only schedules with fewer than c deviations (delays) from an otherwise deterministic scheduler. The intuition is that many concurrency bugs only require a few preemptive context switches at the right places in order to manifest and so will be exposed within reasonable time. In contrast, an unbounded DFS will require an infeasible amount of time to complete and will mostly explore context switches at deep locations (due to the depth-first search order) before timing out. Of course, bugs that require more than c preemptions (or delays) in order to manifest will be missed, so full coverage is sacrificed. Schedule bounding has two additional benefits, regardless of bug finding ability. First, it produces simple counterexamples; a schedule with a small number of preemptions is likely to be easy to understand. This property has been used in trace simplification [JS10, HZ11]. Secondly, schedule bounding provides a *bounded cover-*

[2] We define a data race as two threads accessing the same shared memory location concurrently (i.e. without intervening synchronisation between the two threads), where at least one of the accesses is a write.

[3] Of course, an operation like locking a mutex is often implemented using weak memory operations, but we abstract these low-level details and instead treat such operations as atomic.

age guarantee; given a preemption bound of c, if all schedules within the preemption bound are explored and free from bugs, then any remaining bugs must require at least $c + 1$ preemptions. A guarantee of this kind provides some indication of the necessary complexity and probability of occurrence of any bugs that might remain. The bound can be increased iteratively so that a superset of schedules is explored on each iteration and all schedules will be explored in the limit. Thus, *iterative* schedule bounding determines an order in which to explore *all* schedules, with the hope that many concurrency bugs will be exposed more quickly than when using an unbounded depth-first search. We formalise and evaluate preemption bounding and delay bounding in our empirical study in Chapter 3.

- **Partial-order reduction (POR) techniques** [God96, FG05, MQ07a, AAJS14] avoid execution of a certain class of *provably redundant* schedules that reach the same state. In POR [God96], the total-order of a schedule (i.e. a list of operations) is weakened to become a partial-order, yielding an equivalence class of schedules that all reach the same state. Only schedules that have *different* partial-orders need to be executed. *Dynamic partial-order reduction* (DPOR) [FG05] uses POR to perform a DFS that skips many schedules but still ensures full coverage—all bugs will be found if the search completes. Despite the increased efficiency of DPOR over a straightforward DFS, schedule explosion can still occur and so the search may not complete within a reasonable time or resource limit. Furthermore, DPOR provides no useful coverage guarantees if it does not complete. Thus, for an incomplete search, DPOR inherits the problems of an unbounded DFS (it will be biased towards deep context switches), albeit much more efficient. We introduce POR and DPOR in more detail, as well as our new reduction technique that goes beyond POR for programs with mutexes, in Chapter 4.

- **Randomisation- and heuristic-based techniques** [BKMN10, NBMM12, YNPP12] explore schedules using specially designed heuristics or randomisation. Typically, they do not record which schedules have been explored (they do not use a depth-first search) and so cannot aim to explore the entire schedule-space. *Probabilistic concurrency testing* (PCT) [BKMN10] is a well-known randomisation-based technique that uses randomisation and a priority-based scheduler that cannot guarantee exploration of all schedules; however, it is claimed to find bugs quickly and even provides a *probabilistic guarantee* of finding bugs. We describe and evaluate PCT in our empirical study in Chapter 3.

2.2 Concurrent program model

We now introduce our concurrent program model P, a labelled state transition system that abstracts away from programming language specifics, but note that the techniques described in this thesis are applicable to real concurrent programs written in languages such as C++ and Java. We use our model to describe SCT algorithms in an unambiguous manner. Our model is based on Flanagan and Godefroid's model [FG05].

2.2.1 States

Let State be the set of all states. A state $s = (ss, tss)$ of the system is a tuple where $ss \in \mathsf{SharedState}$ is the shared state and $tss : \mathsf{Tid} \to \mathsf{ThreadState}$ is the thread state of every thread (a mapping from each thread id to a thread state). Thus, we assume the following disjoint finite sets (types):

- Tid: The set of all thread identifiers (thread ids). We assume the program consists of a finite number of threads, where each thread has a unique thread id.
- $\mathsf{SharedState}$: The set of all shared states. The shared state part of a state represents the global variables, heap, mutexes, condition variables, and any other state that can be accessed by multiple threads.
- $\mathsf{ThreadState}$: The set of all thread states. The thread state of each thread represents the thread's private data, such as its instruction pointer, local variables, stack, registers, etc. In languages like C++, local variables can be shared between threads; thus, in this case, such variables are typically considered to be part of the shared state, unless it can be shown that their addresses are never passed to other threads.

Given a function such as tss, let $tss[a \mapsto b]$ yield a function that is identical to tss except that $tss(a) = b$.

2.2.2 Transitions

With each thread id $tid \in \mathsf{Tid}$ and thread state $ts \in \mathsf{ThreadState}$, we associate a unique transition $t_{tid,ts} : \mathsf{SharedState} \rightharpoonup \mathsf{SharedState} \times \mathsf{ThreadState}$. Thus, a transition $t_{tid,ts}$ is a partial function that defines how thread tid in thread state ts mutates its thread state and the shared state when executed. A transition corresponds to a thread executing one *visible operation* [God97] (an operation that accesses the shared state) followed by a finite number of *invisible* operations (that access only the thread state) up until immediately before the

next visible operation. We clarify this with an example below. Considering interleavings of invisible operations is unnecessary when checking safety property violations, such as deadlocks and assertion failures [God97]. A transition that is not defined for a particular shared state corresponds to a thread that is blocked e.g. because it is waiting for a mutex to be released. Let $\mathsf{Transition}$ be the set of all transitions. The state transition relation $\delta \subseteq \mathsf{State} \times \mathsf{Transition} \times \mathsf{State}$ defines the labelled transitions between states. We write $s \xrightarrow{t} s'$ to indicate that $(s, t, s') \in \delta$. The transition relation δ is defined by the following rule:

$$\frac{tss(tid) = ts \qquad t_{tid,ts}(ss) \text{ is defined} \qquad t_{tid,ts}(ss) = (ss', ts') \qquad tss' = tss[tid \mapsto ts']}{(ss, tss) \xrightarrow{t_{tid,ts}} (ss', tss')}$$

Thus, given a state $s = (ss, tss)$, every thread id is mapped to a thread state in tss, and for every (tid, ts) entry in tss, there is a corresponding transition $t_{tid,ts}$ which, *if defined* for ss, yields a state transition. Also note that a state transition only updates the thread state of the corresponding thread as well as the shared state; all other thread states are unchanged (and thus, so are the corresponding transitions for other threads). For convenience, let $next(s)$ denote the corresponding transitions for state s (based on individual thread states—there will be one for every thread id, even if the thread is blocked) and let $enabled(s)$ denote the subset of these that are actually defined for the state (based on the shared state component). Formally:

$$next((ss, tss)) = \{t_{tid,ts} \mid tss(tid) = ts\}$$

$$enabled((ss, tss)) = \{t_{tid,ts} \mid tss(tid) = ts \wedge t_{tid,ts}(ss) \text{ is defined}\}$$

Given a transition $t_{tid,ts}$ let $thread(t_{tid,ts}) = tid$ yield the thread id of the transition. We say that a transition t (and thread $thread(t)$) is *enabled* in state s iff $t \in enabled(s)$. If $enabled(s) = \emptyset$ then there are no transitions from this state and we say that s is a deadlock state or *terminal* state.

For convenience, let $enabledThreads(s)$ denote the enabled threads in s. Formally:

$$enabledThreads(s) = \{tid \mid \exists t.\ t \in enabled(s) \wedge thread(t) = tid\}$$

We will occasionally use object-oriented style dot-notation to apply functions that take

one argument. For example, we could write the previous definition as:

$$s.enabledThreads = \{tid \mid \exists t.\ t \in s.enabled \wedge t.thread = tid\}$$

We will also use dot-notation to access tuple components via the symbols from the original definitions. For example, given a state s, we can access the shared state via $s.ss$ and the thread states via $s.tss$.

Transition example. As described above, a transition corresponds to a thread executing one visible operation (an operation that accesses the shared state) followed by a finite number of invisible operations (that access only the thread state), up until immediately before the next visible operation. Say thread *tid* is in thread state *ts* such that it is about to execute the following statements:

```
1 i = g1;
2 if(i == 0) {
3   j = 1;
4 } else {
5   j = 2;
6   g1 = 1;
7 }
8 g2 = 3;
9 // ... [more statements]
```

Assume that $g1$ and $g2$ are global variables (part of the shared state) while i and j are local variables (part of the thread state). The transition $t_{tid,ts}$ represents all possible behaviours for *tid* from thread state *ts* until the next visible operation. The transition also yields the next transition for this thread (and there may be several possible next transitions depending on the shared state parameter that the transition is applied to). Thus, the transition $t_{tid,ts}$ captures the fact that *tid*'s instruction pointer is at line 1 as well as the values of i and j (as these are represented in the thread state *ts*). Reading the value of $g1$ from the shared state (into local variable i) is the visible operation of transition $t_{tid,ts}$. Which invisible operations follow depends on the value read from $g1$. Similarly, the next transition will either be one that first executes (as its visible operation) line 6 or line 8, depending on the the value read from $g1$. As explained, the execution of one visible operation and multiple invisible operations is collapsed into a single transition, as this is sufficient for finding all safety property violations, such as deadlocks and assertion failures [God97].

Labelled transition system The overall semantics of a concurrent program can be defined as a labelled transition system, $P = (\mathsf{State}, \mathsf{Transition}, \delta, s_0)$, where State is the set of all states, $\mathsf{Transition}$ is the set of all thread transition functions, δ is the set of labelled state transitions and s_0 is the initial state of the system.

2.2.3 Schedules

A *schedule* can be represented as a list of transitions $S = \langle t_1, t_2, \ldots, t_k \rangle$ such that there exist states $s_0, s_1, \ldots, s_k$, where s_0 is the initial state and $s_0 \xrightarrow{t_1} s_1 \ldots \xrightarrow{t_k} s_k$. In other words, a schedule is a path through the transition system. In our model, we assume that all schedules are finite. This is a reasonable assumption because we consider target programs that test some concurrency scenario and then terminate; we do not attempt to reason about infinite schedules. We also assume that a transition cannot appear multiple times in a schedule, which implies that thread states are not repeated. This simplifies the presentation and can be enforced by assuming that each thread state stores the number of operations it has executed so far in a local variable. Let $\mathit{state}(S)$ denote s_k, the state reached by executing S from the initial state. If $\mathit{state}(S)$ is a terminal state, then S is a *terminal schedule*.

We use the following definitions for lists. Let $S = \langle t_1, \ldots, t_k \rangle$ be a list. We define:

- $\mathit{dom}(S) = \{1, \ldots, k\}$.
- $\mathit{last}(S) = t_k$ (the last element of the list).
- $|S| = k$ (the length of the list).
- $S(i) = t_i$, for $i \in \mathit{dom}(S)$ (the ith element of the list).
- $S[i : j]$ to be the sub-list of S from the ith to the jth element (inclusive): $S[i : j] = \langle S(i), S(i+1), \ldots, S(j-1), S(j) \rangle$, where $S[i : j]$ is defined to be $\langle\rangle$ if $j < i$. Any elements in the resulting list that are not defined are omitted; e.g. $S[-1 : 0] = \langle\rangle$.
- $S \cdot S'$ to be the concatenation of S with a second list $S' = \langle t'_1, \ldots, t'_l \rangle$. Thus, $S \cdot S' = \langle t_1, \ldots, t_k, t'_1, \ldots, t'_l \rangle$.

Given a list $S = w \cdot \langle a, b \rangle \cdot u$, we say that the elements a and b are *adjacent* in S.

2.2.4 Shared objects

Our model currently keeps the shared state abstract. However, it can be useful and intuitive to view the shared state as a map from shared objects to values, so that a shared

state is a function: $ss : \mathsf{Object} \to \mathsf{Value}$. Thus, we refine our model by introducing:

- Object: The set of all shared objects. A shared object represents an individual global variable (or perhaps the address of an individual byte in the heap), mutex, condition variable, or any other shared data structure that is accessed by a visible operation.
- Value: The set of all values that shared objects can have, such as all 8-bit bit vectors for shared bytes, and boolean values true and false for boolean variables.

We assume that each transition $t_{tid,ts}$ only accesses a single shared object o; let $obj(t_{tid,ts})$ denote this shared object. A transition becomes a partial function that takes a single value and yields a value and thread state: $t_{tid,ts} \in \mathsf{Value} \rightharpoonup \mathsf{Value} \times \mathsf{ThreadState}$. The transition relation δ is then defined by the following rule:

$$\frac{\begin{array}{c} tss(tid) = ts \qquad o = obj(t_{tid,ts}) \qquad v = ss(o) \\ t_{tid,ts}(v) \text{ is defined} \qquad t_{tid,ts}(v) = (v', ts') \qquad ss' = ss[o \mapsto v'] \qquad tss' = tss[tid \mapsto ts'] \end{array}}{(ss, tss) \xrightarrow{t_{tid,ts}} (ss', tss')}$$

Thus, a transition is defined depending on the value of its accessed shared object and the transition can only update the value of its accessed shared object as well as the thread state of the corresponding thread. Assuming a *single* shared object that represents the entire shared state is equivalent to our original model. Furthermore, in Chapter 4, we introduce the notion of *independent* transitions; note that transitions can still be independent even if they access the same shared object. Thus, we henceforth assume this refined model without loss of generality.

2.3 Common visible operations

Our model abstracts away the different types of visible operations, such as locking a mutex, writing to a shared variable, etc. However, it is useful to consider some common visible operations that are typically assumed in practice and the shared objects that they access. Furthermore, we demonstrate how to simulate thread creation and termination in our model. We henceforth use operation to mean visible operation, unless otherwise stated. Some common operations are:

- `start`: We typically assume that every thread's first visible operation is a `start` operation that accesses a shared object that represents the thread itself. We do not require thread creation in our model; instead, we assume that a thread *tid* is

initially blocked on its `start` operation until some other thread "creates" thread *tid* by updating the thread object of *tid* so that thread *tid* is now enabled and can execute its `start` operation. Thus, the `start` operation does not have any effect, other than providing an initial visible operation for every thread. We typically assume that there is one initially enabled thread and, thus, its `start` operation does not block.

- `create`: When executed by a thread, a `create` operation accesses a shared object that represents some other thread *tid*. The operation represents "creating" thread *tid*, as described above, such that thread *tid*'s initial `start` operation is no longer blocked.

- `end`: We typically assume that threads can "terminate". When a thread *tid* terminates, it executes the `end` operation that accesses a shared object that represents the thread *tid* itself. We then assume the thread blocks forever on a second `end` operation that again accesses the thread itself. Thus, the thread cannot execute any further operations.

- `join`: When executed by a thread, a `join` operation accesses a shared object that represents some other thread *tid*. The operation blocks iff thread *tid* has not yet terminated by executing its first `end` operation. Thus, we assume that the first `end` operation updates the value of the shared object for thread *tid* such that `join` operations that access thread *tid* no longer block.

- `read`: A `read` operation reads the value of a shared variable. Thus, the shared variable's value is not changed.

- `write`: A `write` operation updates the value of a shared variable.

- `lock`: A `lock` operation accesses a mutex. We can let the value of a mutex be $\perp$ iff no thread owns the mutex or thread id *tid* $\in$ Tid iff thread *tid* owns the mutex. A `lock` executed by thread *tid* blocks if the mutex value is not $\perp$. Otherwise, the value is $\perp$ and so the operation is enabled and updates the mutex state to *tid*.

- `unlock`: An `unlock` operation unlocks a mutex. We assume that a thread *tid* must only ever try to `unlock` a mutex that it owns (i.e. that has state *tid*). Doing otherwise could be treated as an invalid program or could be assumed to be a bug in the program such that further exploration is unnecessary. Thus, we assume that an `unlock` operation executed by thread *tid* is enabled iff the mutex state is *tid* and updates the state of the mutex to $\perp$.

3 Empirical Study

In this chapter, we present an independent empirical study of SCT techniques. This is motivated by the lack of comparison and independent empirical evaluation of SCT techniques prior to our work. We gathered 49 buggy concurrent software benchmarks from public code bases which we call SCTBench. We applied a modified version of an existing concurrency testing tool, called Maple, to SCTBench, testing five SCT techniques: depth-first search (DFS), iterative preemption bounding (IPB), iterative delay bounding (IDB), a controlled random scheduler, and probabilistic concurrency testing (PCT).

We attempted to answer several research questions, including:

- Which technique performs the best in terms of bug finding ability?
- Does PCT beat the other techniques as in previous work?
- How effective are the two main schedule bounding techniques, preemption bounding and delay bounding, at finding bugs?
- What challenges are associated with applying concurrency testing techniques to existing code?
- Can we classify certain benchmarks as trivial or non-trivial?

Our main findings are:

- PCT (with parameter d=3) was the most effective technique in terms of bug finding; it found all the bugs found by the other techniques, plus an additional three, and it missed only one bug.
- Surprisingly, the "naïve" controlled random scheduler, which randomly chooses one thread to execute at each scheduling point, performed well, finding more bugs than preemption bounding and just two fewer bugs than delay bounding. In particular, random scheduling performed better than preemption bounding and delay bounding on the work stealing queue benchmark which was originally used to evaluate preemption bounding and delay bounding.

- Delay bounding was superior to preemption bounding and schedule bounding was superior to an unbounded DFS, as in prior work.
- The majority of bugs in SCTBench can be exposed using a small schedule bound (1-2), supporting previous claims, although one benchmark requires 5 preemptions.
- The need to remove nondeterminism and control all synchronisation (as is required for SCT) can be nontrivial. There were 8 distinct programs that could not easily be included in out study, such as those that perform network and inter-process communication.
- Some of the benchmarks used in prior work are arguably trivial. We report various properties about the benchmarks tested, such as the fact that the bugs in 18 benchmarks were exposed 50% of the time when using random scheduling. We note that future work should not use the benchmarks that we classify as trivial when presenting new techniques, other than as a minimum baseline.

To make our study reproducible, we provide the 49 benchmarks (SCTBench), our scripts, and the modified version of Maple used in our experiments, online:

`https://github.com/mc-imperial/sctbench`

We believe SCTBench will be valuable for future work on concurrency testing in general and SCT in particular. Our results are given in terms of number of schedules, not time, which allows them to be easily compared with other work and tools.

Relation to published work The core material of this chapter was published in our conference paper [TDB14] and journal paper [TDB16].

3.1 Motivation

Prior work suggests that schedule bounding techniques (introduced in §2.1), like preemption bounding [MQ07b] and delay bounding [EQR11], are effective techniques for finding concurrency bugs [MQ07b, EQR11]. The evaluation of these techniques has focused on a particular set of C# and C++ programs that target the Microsoft Windows operating system, most of which are not publicly available. Furthermore, this prior work typically uses an unbounded DFS as a baseline but does not consider other straightforward SCT techniques, such as a controlled random scheduler that randomly chooses a thread to

execute after each step.[1] The PCT algorithm [BKMN10] is another technique that has been shown to find bugs in large applications such as Mozilla Firefox and Internet Explorer [BKMN10]; these applications were not made to be deterministic but we note that PCT can be used as an SCT technique when applied to deterministic programs. However, a thorough comparison of PCT with other SCT techniques has not been conducted.[2] We believe that these exciting and important claims about the effectiveness of SCT techniques would benefit from further scrutiny using a wider range of publicly available benchmarks.

To this end, we present an independent, reproducible empirical study of SCT techniques. We have put together SCTBench, a set of 49 publicly available benchmarks gathered from a combination of stand-alone multithreaded test cases and test cases drawn from 13 distinct applications and libraries. These are benchmarks that have been used in previous work to evaluate concurrency testing tools (although mostly not in the context of SCT), with a few additions, which we have made amenable to SCT. We use an extended version of Maple [YNPP12], an open source concurrency testing tool, to test the benchmarks.

3.2 The techniques

In this section, we describe each technique that we evaluate: an unbounded DFS, iterative preemption bounding, iterative delay bounding, PCT and controlled random scheduling. We also discuss upper bounds of the DFS-based techniques in terms of the number of terminal schedules and describe the probabilistic guarantee given by the PCT algorithm.

3.2.1 Unbounded depth-first search (DFS)

SCT is typically implemented using an unbounded DFS so that the remaining schedules that need to be explored can be efficiently stored using a stack data structure, where the maximum height of the stack is equal to the number of steps (transitions) in the longest schedule.

The DFS algorithm can be represented as the recursive procedure in Algorithm 1. Thus, note that the schedule-space can conceptually be represented as a prefix-tree, where each node is a schedule and the branches of a node are the enabled transitions at the scheduling point. Recall that we are performing a dynamic analysis and so the schedule tree is not known a priori; it is discovered on-the-fly. In other words, the use of $state(S)$ corresponds

[1] We note that [MQ07b] plots the state (partial-order) coverage of preemption bounding against a technique called "random" on a single benchmark, but the details of this and the bug finding ability are not mentioned.

[2] We note that [BKMN10] compares PCT against the use of random sleeps, but not against controlled random scheduling. PCT is also compared against preemption bounding, but only on two benchmarks.

Algorithm 1 Unbounded DFS algorithm (DFS).

```
1: procedure EXPLORE(S)
2:   for each t ∈ enabled(state(S))
3:     Explore(S · ⟨t⟩)
4: end procedure
```

to forcing the schedule S on the program under test so that the enabled threads can be inspected. After exploring the first terminal schedule, the search then backtracks to the most recent scheduling point; the next schedule is explored by executing the program *from the start*, replaying the previous schedule up to the most recent scheduling point, scheduling the next enabled thread and then continuing to schedule threads until termination once again. Algorithm 1 does not specify the order in which enabled transitions should be explored from each state. In our implementation in Maple §3.3, we explore enabled transitions in thread creation order, starting with the most recently executing thread and wrapping in a round-robin fashion. For example, if the last transition of a schedule S was from thread 3, then the order in which threads will be explored (if enabled) from $state(S)$ is $[3, 4, \ldots, n, 1, 2]$ (assuming n threads). We cover the implementation of an SCT tool in more detail in Chapter 5.

When the search completes, *all* terminal schedules (and all terminate states) have been explored. A key downside to this baseline approach is that the number of schedules increases exponentially with the length of the program (the number of transitions in a terminal schedule). Thus, exploring all schedules is usually infeasible. If the search does not complete, there is no coverage guarantee. Furthermore, since the search order is depth-first, an incomplete search is likely to favour exploring many different preemptions at deep scheduling points (i.e. towards the end of the execution). This can mean that schedules with earlier preemptions are not considered. Thus, bugs that require early preemptions will be missed.

3.2.2 Iterative preemption bounding

Preemption bounding [MQ07b] uses a DFS but bounds the number of preemptive context switches in a schedule. A *context switch* occurs in a schedule when control switches from one thread to another. Formally, given a schedule S, transition $S(i)$ is a context switch if and only if:

$$i > 1 \text{ and } S(i).thread \neq S(i-1).thread$$

Algorithm 2 Preemption bounding algorithm.

```
1: procedure EXPLORE(S)
2:   for each t ∈ enabled(state(S))
3:     if PC(S · ⟨t⟩) ≤ c
4:       Explore(S · ⟨t⟩)
5: end procedure
```

A *preemptive* context switch (a *preemption*) is a context switch away from a thread that was enabled (and thus was preempted from continuing). Consider a context switch $S(i)$. Let $s = state(S[1 : i-1])$ be the state that is reached immediately *after* transition $S(i-1)$ and before transition $S(i)$. Transition $S(i)$ is a preemption iff $S(i-1).thread$ is enabled in s. In other words, the schedule could have continued with $S(i-1).thread$, but $S(i).thread$ was executed instead.

We define the *preemption count* PC of a schedule recursively. A schedule of length zero or one has no preemptions. Otherwise:

$$PC(S \cdot \langle t \rangle) = \begin{cases} PC(S) + 1 & \text{if } last(S).thread \neq t.thread \\ & \quad \wedge\ last(S).thread \in state(S).enabledThreads \\ PC(S) & \text{otherwise} \end{cases}$$

With a preemption bound of c, any schedule S with $PC(S) > c$ will not be explored. Algorithm 2 shows the preemption bounding algorithm as a recursive procedure; the recursive call is only made if the schedule has a preemption count that is less than or equal to the preemption bound c. Note that for given any schedule S, there must exist a transition t in $state(S).enabled$ that can be explored without increasing the preemption count, unless $state(S)$ is a terminal state.

The idea behind preemption bounding is that it greatly reduces the number of schedules, but still allows many bugs to be found [MQ07b, MQB+08, EQR11]. The intuition is that many bugs only require a *few* preemptions at at the *right places* in order to manifest. In contrast, an unbounded search is unlikely to complete within feasible time, as described above. Thus, using a low preemption bound increases the chance of exploring all schedules within the preemption bound, without exceeding the time or schedule limit, which will include exploring preemptions at various depths.

Example 1. *Consider Figure 3.1, which shows a simple multithreaded program. Thread T0 launches three threads concurrently and is then disabled. All variables are initially zero and threads execute until there are no statements left. We refer to the visible operations of*

T0	T1	T2	T3
a) `create(T1,T2,T3)`	b) `x=1`	d) `z=1`	e) `assert(x==y)`
	c) `y=1`		

Figure 3.1: Simple multithreaded program.

Algorithm 3 Iterative preemption bounding algorithm.

```
1: procedure IPB
2:     c = 0
3:     repeat
4:         Explore(⟨⟩)                ▷ Preemption bounding procedure from Algorithm 2
5:         c = c + 1
6:     until EXPLORE did not skip any schedules
7: end procedure
```

each thread via the statement labels (a, b, c, etc.) and we (temporarily) represent schedules as a list of labels.

An example of a schedule with zero preemptions is $\langle a, b, c, e, d\rangle$. Note that, for example, e is not a preemption in this particular schedule because T1 has no more statements and so is considered disabled after c. A schedule that causes the assertion e to be violated is $\langle a, b, e\rangle$; this schedule has one preemption at operation e. The bug will not be found with a preemption bound of zero, but will be found with any greater bound.

Instead of picking a preemption bound, it is possible to perform *iterative preemption bounding*, where the preemption bound is initially set to zero and incremented after each search completes. Iterative preemption bounding is shown in Algorithm 3, which calls EXPLORE from Algorithm 2. The process repeats until the preemption bound was increased enough to allow all schedules to be explored or until the time limit is reached. Therefore, iterative preemption bounding essentially defines a partial-order in which to explore schedules: schedule S will be explored before schedule S' if $PC(S) < PC(S')$. Thus, iterative preemption bounding is a heuristic that prioritises schedules with a low preemption count, aiming to expose buggy schedules before the time or schedule limit is reached. Note that iterative preemption bounding will repeat schedules after the first call to EXPLORE because each call will search for all schedules with *at most c* preemptions. Thus, the first call will explore schedules with 0 preemptions, the second will explore schedules with 0–1 preemptions, the third will explore schedules with 0–2 preemptions, etc.

3.2.3 Iterative delay bounding

Delay bounding bounds the number of *delays* (deviations from a deterministic scheduler) in a schedule. A *delay* conceptually corresponds to blocking the thread that would be chosen by the scheduler at a scheduling point, which forces the next thread to be chosen instead. The blocked thread is then immediately re-enabled. As such, delay bounding requires an underlying deterministic scheduler. In the remainder of this thesis we assume that delay bounding is applied in the context of a non-preemptive round-robin scheduler that considers threads in thread creation order, starting with the most recently executing thread. We assume this instantiation of delay bounding because it has been used in previous work [EQR11] and is straightforward to explain and implement.

The following is a definition of the delay count of a schedule assuming the non-preemptive round-robin scheduler. Assume that each thread id is a non-negative integer, numbered in order of creation; the initial thread has id 0, and the last thread created has id $N-1$. For two thread ids $x, y \in \{0, \ldots, N-1\}$, let $\mathit{distance}(x, y)$ be the unique integer $d \in \{0, \ldots, N-1\}$ such that $(x + d) \bmod N = y$. Intuitively, this is the "round-robin distance" from x to y. For example, given four thread ids $\{0, 1, 2, 3\}$, $\mathit{distance}(1, 0)$ is 3. For a schedule S and thread id tid, let $\mathit{delays}(S, \mathit{tid})$ yield the number of delays required to schedule thread tid at the state reached by S:

$$\begin{aligned}\mathit{delays}(S, \mathit{tid}) = \quad & |\{\ x : 0 \leq x < \mathit{distance}(\mathit{last}(S).\mathit{thread}, \mathit{tid})) \ \wedge \\ & (\mathit{last}(S).\mathit{thread} + x) \bmod N \in \mathit{state}(S).\mathit{enabledThreads}\ \}|\end{aligned}$$

This is the number of enabled threads that are skipped when moving from $\mathit{last}(S).\mathit{thread}$ to tid. For example, let $\mathit{last}(S).\mathit{thread} = 3$, $\mathit{state}(S).\mathit{enabledThreads} = \{0, 2, 3, 4\}$ and $N = 5$. Then, $\mathit{delays}(S, 2) = 3$ because in order to execute thread 2, threads 3, 4 and 0 are skipped (but not thread 1, because it is disabled).

We define the delay count DC of a schedule recursively. A schedule of length zero or one has no delays. Otherwise:

$$DC(S \cdot \langle t \rangle) \quad = \quad DC(S) + \mathit{delays}(S, t.\mathit{thread})$$

With a delay bound of c, any schedule S with $DC(S) > c$ will not be explored. The algorithm for delay bounding is identical to algorithm for preemption bounding (Algorithm 2), except that the preemption count PC is replaced with the delay count DC. As in preemption bounding, note that for given any schedule S, there must exist a transition t in $\mathit{state}(S).\mathit{enabled}$ that can be explored without increasing the delay count, unless $\mathit{state}(S)$

T0	T1	T2	T3
a) `create(T1,T2,T3)`	b) `x=1`	f) `x=1`	e) `assert(x==y)`
	c) `y=1`	g) `y=1`	

Figure 3.2: Adversarial delay-bounding example.

is a terminal state.

The intuition behind delay bounding is similar to that of preemption bounding; that is, many bugs can be found with only a few preemptions or delays [MQ07b, MQB+08, EQR11]. The extra idea behind delay bounding is that it often also does not matter *which* thread is switched to after a preemption; thus, allowing only the next enabled thread without spending additional delays reduces the number of schedules more than preemption bounding, while still allowing many bugs to be found. Indeed, given all schedules of a program, the subset with at most c delays is a subset of the schedules with at most c preemptions. Thus, delay bounding always reduces the number of schedules by at least as much as preemption bounding.

Example 2. *Consider Figure 3.1 once more. Assume thread creation order* $\langle T0, T1, T2, T3\rangle$. *The assertion can also fail via:* $\langle a, b, d, e\rangle$, *with one delay/preemption at* d. *However, a preemption bound of one yields 11 terminal schedules, while a delay bound of one yields only 4 (assume that an assertion failure is a terminal state).*

Now consider Figure 3.2, which is a modified version of the program where the statements of T2 have been replaced with the same statements as T1, which we label as f) and g). Now, the assertion cannot fail with a delay bound of one because two delays must occur so that T1 and T2 do not execute all their statements. For example, $\langle a, b, e\rangle$ *exposes the bug, but executing* e *uses two delays. However, this schedule only has one preemption, so the assertion can still fail under a preemption bound of one.*

Adding an additional n *threads between T1 and T3 (in the creation order) with the same statements as T1 will require* n *additional delays to expose the bug, while still only one preemption will be needed. Empirical evidence [EQR11] suggests that adversarial examples like this are not common in practice. Our results (§3.7) also support this.*

As with preemption bounding, it is possible to perform *iterative delay bounding*, where the delay bound is initially set to zero and incremented after each search completes. The algorithm for iterative delay bounding is identical to Algorithm 3, except Explore must be a delay bounded search. As with iterative preemption bounding, iterative delay bounding will repeat schedules after the first call to Explore.

3.2.4 Controlled random scheduling

A controlled random scheduler uses randomisation to determine the schedule that is explored. At each scheduling point, one transition is randomly chosen from the set of enabled transitions using a uniform distribution. Unlike schedule fuzzing, where random sleeps are used to perturb the OS scheduler [BAEFU06], the random scheduler fully controls scheduling nondeterminism. As with any SCT technique, the executed schedule can easily be recorded and replayed (because schedule nondeterminism is controlled). However, no information is saved *for subsequent executions*. Thus, it is possible that the same schedule will be explored multiple times. The search cannot "complete", even for programs with a small number of schedules. Additionally, a random scheduler *can* be used on programs that exhibit nondeterminism *beyond* scheduler nondeterminism, although schedule replay would be unreliable. In this thesis, we consider only deterministic programs.

3.2.5 Probabilistic Concurrency Testing

The PCT algorithm [BKMN10] uses a randomised priority-based scheduler such that the highest priority enabled thread is scheduled at each scheduling point. A bounded number of *priority change points* are inserted at random depths in the execution which change the currently executing thread's priority to a low value. Importantly, the random depths of the change points are chosen in advance, *uniformly* over the estimated number of steps (transitions) of a schedule. This is in contrast to random scheduling, where a random choice is made at every step.

More formally, the algorithm is described in [BKMN10] as follows. Given a program with at most n threads and at most k steps (in a single terminal schedule), choose a bound d. Note that it is necessary to have estimates for n and k; these can be obtained by performing several profiling runs, which we discuss further in §3.6. The algorithm then performs the following for each execution of a single schedule:

1. Randomly assign each of the n threads a distinct *initial* priority value from $\{d, d+1, \ldots, d+n\}$. The lower priority values $\{1, \ldots, d-1\}$ are reserved for priority change points.

2. Randomly pick integers $k_1, \ldots, k_{d-1}$ from $\{1, \ldots, k\}$. These will be the priority change points.

3. Schedule threads strictly according to their priorities; never schedule a thread if a higher priority thread is enabled. In other words, from a state s, execute the

transition t from $s.enabled$ iff there does not exist t' in $s.enabled$ such that $t.thread$ has a lower priority than $t'.thread$. After executing the k_i-th step ($1 \leq i < d$), change the priority of the thread that executed the step to i.

Example 3. *Once again, consider the program in Figure 3.2. For this program, the number of threads is $n = 4$ and the number of steps is $k = 6$. One way for the bug to occur is for statement e to occur after b but before c. This is possible with one priority change point, so let $d = 2$. Assume the initial random thread priorities chosen are:*

$$\{T0 \mapsto 5, T1 \mapsto 4, T3 \mapsto 3, T2 \mapsto 2\}.$$

Assume the random priority change point chosen is $k_1 = 2$. Thus, the schedule that will be explored is: $\langle a, b, e \rangle$, which causes the assertion to fail. Statement a is executed because $T0$ has the highest priority. $T0$ then becomes disabled, so $T1$ becomes the highest priority thread that is enabled and b is executed. At this point, step 2 was just executed; thus, the priority change point is triggered and $T1$'s priority is lowered to 1. $T3$ becomes the highest priority thread that is enabled and so e is executed.

The work on PCT also introduces the idea of a *bug depth* metric—not to be confused with the depth (number of steps) of a schedule. The *bug depth* is defined as the minimum set of ordering constraints between instructions from different threads that are sufficient to trigger the bug [BKMN10]. Assuming a bug with bug depth d, the probability of the PCT algorithm detecting the bug on a single execution is $1/nk^{d-1}$ (inverse exponential in d).

As with random scheduling (and unlike DFS-based approaches), no information is saved for subsequent executions, so the search cannot "complete" and the technique *can* be used on programs with nondeterminism. Similar to schedule bounding, the intuition behind PCT is that many concurrency bugs typically require certain orderings between only a few instructions in order to manifest [MQ07b, MQB+08, EQR11, LPSZ08].

3.2.6 Upper bounds on number of terminal schedules and probabilistic guarantees

Upper-bounds for the number of terminal schedules produced by the above DFS techniques are described in [MQ07b, EQR11]. In summary, assume at most n threads and at most k steps in each thread. Of those k, at most b steps block (cause the executing thread to become disabled) and i steps do not block. The upper bound for an unbounded DFS is exponential in n and k, and thus infeasible for programs with a large number of steps.

With a scheduling bound of c, the upper bound for preemption bounding is exponential in c (a small value), n (often, but not necessarily, a small value) and b (usually much smaller than k). Crucially, it is no longer exponential in k. The upper bound for delay bounding is exponential only in c (a small value). Thus, delay bounding performs well (in terms of number of terminal schedules) even when programs create a large number of threads.

As explained above, PCT gives a probabilistic guarantee: assuming a bug with bug depth d, the probability of finding the bug with PCT on a single execution is $1/nk^{d-1}$ (inverse exponential in d).

3.3 Maple

We chose to use a modified version of the Maple tool [YNPP12] to conduct our study. Maple is a concurrency testing tool framework for pthread [LB98] programs. It uses the dynamic instrumentation library, PIN [L+05], to test binaries without the need for recompilation. One of the modules, *systematic*, is a re-implementation of the CHESS [MQB+08] algorithm for preemption bounding. The main reason for using Maple, instead of CHESS, is that Maple targets pthread programs. This allows us to test a wide variety of open source multithreaded benchmarks and programs. Previous evaluations [MQ07b, MQB+08, EQR11] focus on C# programs and C++ programs that target the Microsoft Windows operating system, most of which are not publicly available. In addition, CHESS requires re-linking the program with a test function that can be executed repeatedly; creating this type of test harness requires resetting the global state (e.g. resetting the value of global variables) and joining any remaining threads, which can be non-trivial. In contrast, Maple can test native binaries out-of-the-box, by restarting the program for each terminal schedule that is explored. A downside of this approach is that it is slower. Checking for data races is also supported by Maple; as discussed later in §3.6, this is important for identifying visible operations. The public version of CHESS can only interleave memory accesses in native code if the user adds special function calls before each access.[3] We now discuss further implementation details.

Depth-first search As explained in §2.1, SCT techniques often use a DFS in order to efficiently store the remaining unexplored schedules using a stack data structure. The type of DFS determines the order in which schedules are explored—recall from §3.2.1 that the schedule space can be represented as a prefix-tree, where each node is a schedule and the

[3]See "Why does wchess not support /detectraces?" at `http://social.msdn.microsoft.com/Forums/en-us/home?forum=chess`

branches of a node are the enabled transitions at the state reached by the schedule. In our study, we use a left-recursive DFS where child branches (transitions) are ordered by thread ids in thread creation order, starting with the most recently executing thread and wrapping in a round-robin fashion. Thus, the initial execution explores the non-preemptive round-robin schedule; this is the same for all techniques that use a DFS: unbounded DFS, iterative preemption bounding and iterative delay bounding. We discuss the impact of using a DFS on our study in §3.6.

Preemption bounding Maple already included support for preemption bounding, using the underlying DFS approach.

Delay bounding We modified Maple to add support for delay bounding, following a similar design to preemption bounding. At each scheduling point, Maple conceptually constructs several schedules consisting of the current schedule concatenated with an enabled transition t. If executing t will cause the delay bound to be exceeded (as explained in §3.2.3), the schedule is not considered.

Controlled random scheduling Maple already included a controlled random scheduler (although this was not used in prior work [YNPP12]). As explained in §3.2.4, at each scheduling point, one transition is randomly chosen from the set of enabled transitions using a uniform distribution; that transition is then scheduled for one step.

PCT algorithm Prior to our modifications, Maple already included a version of PCT implemented using Linux scheduler priorities [YNPP12]. By changing settings of the Linux scheduler, it is apparently possible to implement strict priorities, as required for PCT. However, in order to ensure that we are using an implementation that is identical to the one described in the original PCT paper [BKMN10], we re-implemented PCT within the SCT framework of Maple; as such, our implementation is very similar to the pseudocode from the PCT paper. This also makes the comparison fair, as all techniques are implemented on the same framework (except for the Maple algorithm). Another reason this was necessary was so that we could run the experiments on our cluster (see §3.7), where it is not possible to change the settings of the Linux scheduler.

Modelling of blocking operations All the techniques used in our study are implemented in Maple's SCT framework (except the Maple algorithm). Immediately before each visible operation (e.g. pthread function or shared memory access), the set of enabled

transitions is determined; if the operation that a thread is about to execute will block, then the thread is considered to be disabled. In this case, control is returned to Maple's scheduler and the thread is marked as disabled. Similarly, when an operation enables other threads, the threads must be marked as enabled at the next scheduling point. We did not add heuristics for automatically detecting when threads become enabled/disabled (e.g. [NBMM12]). Thus, all potentially blocking operations must be implemented/modelled in Maple.

Maple algorithm The Maple tool uses a non-SCT technique by default, which we refer to as the *Maple algorithm* [YNPP12]. This algorithm performs several profiling runs, where the schedule is not controlled, recording patterns of inter-thread dependencies through shared-memory accesses. From the recorded patterns, it predicts possible alternative interleavings that may be feasible, which are referred to as interleaving idioms. It then performs *active* runs, influencing thread scheduling to attempt to force untested interleaving idioms, until none remain or they are all deemed infeasible (using heuristics). Unlike SCT, Maple does not serialise execution. Though non-SCT techniques are generally beyond the scope of this work, we test the Maple algorithm in our study since it is readily available in the tool.

Busy-wait loops A busy-wait loop (or spin loop) is a loop that repeatedly checks whether another thread has written to a shared variable before exiting the loop. These must be handled specially in SCT because the presence of such a loop means there is an infinite length schedule where the looping thread is never preempted. To handle this, we manually inserted a call to *yield* in every busy-wait loop. We also modified Maple so that, during a DFS, a preemption was forced at every yield operation, without increasing the preemption or delay count. This is unsound, as such operations do not guarantee a preemption to another thread in practice and certain bugs may require a yield to *not* be preempted. Prior work provides a sound solution using thread priorities [MQ08], as long as yield statements are added appropriately. However, due to its simplicity and efficiency, and the fact that we are already testing multiple different scheduling algorithms, we used the simpler unsound approach in this study. Furthermore, for benchmarks that use busy-wait loops, forcing a preemption at yield always allows the bug to manifest and guarantees termination (based on our understanding of the benchmarks). All such bugs were indeed found by schedule bounding in our study (except the bug in `misc.safestack`, which was not found by any technique).

Busy-wait loops must also be handled specially in PCT. In the original PCT pa-

Benchmark set	Benchmark types	# used	# skipped
CB	Test cases for real applications	3	17 networked applications.
CHESS	Test cases for several versions of a work stealing queue	4	0
CS	Small test cases and some small programs	29	24 were non-buggy.
Inspect	Small test cases and some small programs	1	28 were non-buggy.
Miscellaneous	Test case for lock-free stack and a debugging library test case	2	0
PARSEC	Parallel workloads	4	29 were non-buggy.
RADBenchmark	Tests cases for real applications	3	5 Chromium browser; 4 networking; 3 (see text).
SPLASH-2	Parallel workloads	3	9 (see text).

Table 3.1: An overview of the benchmark suites used in the study.

per [BKMN10], the authors state that their implementation uses heuristics to identify threads that are not making progress and lowers their priorities with a small probability. In our implementation, we change the priority of the current thread to the lowest possible priority immediately after it executes a yield operation.

3.4 Benchmark Collection

We have collected a wide range of pthread benchmarks from previous work and other sources. We have ensured that all benchmarks are deterministic (modulo scheduling non-determinism) and that all potentially blocking functions are modelled in Maple (or replaced with simpler primitives that are modelled in Maple). Thus, our benchmarks are amenable to SCT and work with Maple's SCT framework. As a result, some benchmarks that use network communication, inter-process communication, less common synchronisation etc., were skipped, as getting these benchmark to work would require significant engineering effort.

Table 3.1 summarises the benchmark suites (with duplicates removed), indicating where it was necessary to skip benchmarks. "Non-buggy" means there were no existing bugs documented and we did not find any during our examination of the benchmark. We now provide details of the benchmark suites (§3.4.1) and challenges of the application of SCT identified through our benchmark gathering exercise (§3.4.2).

3.4.1 Details of benchmark suites

Concurrency Bugs (CB) Benchmarks [YN09] Includes buggy versions of programs such as `aget` (a file downloader) and `pbzip2` (a file compression tool). We modified `aget`, modelling certain network functions to return data from a file and to call its interrupt handler asynchronously. Many benchmarks were skipped due to the use of networking, multiple processes and signals (`apache`, `memcached`, `MySQL`).

CHESS [MQB+08] A set of test cases for a work stealing queue, originally implemented for the Cilk multithreaded programming system [FLR98] under Windows. The `WorkStealQueue` (`WSQ`) benchmark has been used frequently to evaluate concurrency testing tools [MQ08, MQB+08, MQ07b, MQ07a, BKMN10, NBMM12]. We manually translated the benchmarks to use pthreads and C++11 atomics; a heap corruption error occurred when running two of the tests natively (without Maple). We fixed this issue and SCT revealed a bug that is much rarer, which we use in the study.

Concurrency Software (CS) Benchmarks [CF11] Examples used to evaluate the ESBMC tool [CF11], including small multithreaded algorithm test cases (e.g. bank account transfer, circular buffer, dining philosophers, queue, stack), a file system benchmark and a test case for a Bluetooth driver. These tests included unconstrained inputs. None of the bugs are input dependent, so we selected reasonable concrete values. We had to remove or define various ESBMC-specific functions to get the benchmarks to compile.

Inspect Benchmarks [YCG08] Used to evaluate the INSPECT concurrency testing tool. We skipped the `swarm_isort64` benchmark, which did not terminate after five minutes when performing data race detection (see §3.6). There were no documented bugs, and testing all benchmarks revealed a bug in only one benchmark, `qsort_mt`, which we include in the study.

Miscellaneous We encountered two individual test cases, which we include in the study. The `safestack` test case, which was posted to the CHESS forums[4] by Dmitry Vyukov, is a lock-free stack designed to work on weak-memory models. The bug exposed by the test case also manifests under sequential consistency, so it should be detectable by existing SCT tools. Vyukov states that the bug requires at least three threads and at least five

[4]See "Bug with a context switch bound 5" at `http://social.msdn.microsoft.com/Forums/en-US/home?forum=chess`

preemptions. Previous work reported a bug that requires three preemptions [EQR11], which was the first bug found by CHESS that required that many preemptions.

The `ctrace` test case, obtained from the authors of [KZC12], exposes a bug in the ctrace multithreaded debugging library.

PARSEC 2.0 Benchmarks [Bie11] A collection of multithreaded programs from many different areas. We used `ferret` (content similarity search) and `streamcluster` (online clustering of an input stream), as these contain known bugs. We created three versions of `streamcluster`, each containing a distinct bug. One of these is from an older version of the benchmark and another was a previously unknown bug which we discovered during our study (see *Memory safety* in §3.4.2). We configured the `streamcluster` benchmarks to use non-spinning synchronisation and added a check for incorrect output. All benchmarks use the "test" input values (the smallest) with two threads, except for `streamcluster2`, where the bug requires three threads.

RADBenchmark [JPPS11] Consists of 15 tests that expose bugs in several applications. The 3 benchmarks we use test parts of Mozilla SpiderMonkey (the Firefox JavaScript engine) and the Mozilla Netscape Portable Runtime Thread Package, which are suitable for SCT. We skipped 9 benchmarks due to use of networking and multiple processes. Several tested the Chromium browser; the use of a GUI leads to nondeterminism that cannot be controlled or modelled by any SCT tools we know of. We skipped 3 benchmarks which behave unexpectedly when running under Maple's SCT framework. We reduced the thread counts and parameter values of stress tests, as is appropriate for SCT (see *Stress tests* in §3.4.2). Compared to the original version of this study [TDB14], we compiled the RADBench benchmarks with different compiler flags so that certain provided libraries are statically linked; Maple works with both dynamically and statically linked libraries, but we wanted to ensure that the same libraries are used when we uploaded the benchmarks to machines in our cluster.

SPLASH-2 [W+95] Three of these benchmarks have been used in previous work [PLZ09, BKMN10]. SPLASH-2 requires a set of macros to be provided; the bugs are caused by a set that fail to include the "wait for threads to terminate" macro. Thus, all the bugs are similar. For this reason, we just use the three benchmarks from previous work, even though the macros are likely to cause issues in the other benchmarks. We added assertions to check that all threads have terminated as expected. We reduced the values of input parameters, such as the number of particles in `barnes` and the size of the matrix in `lu`, so

the tests could run without exhausting memory (due to Maple's data race detector). Reducing parameters as much as possible is appropriate for SCT (see *Stress tests* in §3.4.2); we discuss this further in §3.7.

3.4.2 Effort required to apply SCT

We restrict our benchmarks to those where we can apply SCT so that we can apply all techniques to all benchmarks (recall that random scheduling and PCT can be applied to nondeterministic programs). We encountered a range of issues when trying to apply SCT to the benchmarks, which we now discuss.

Environment modelling When applying SCT, system calls that interact with the environment, and hence can give nondeterministic results, must be modelled or fixed to return deterministic values. Similarly, depending on the framework being used, functions that can cause threads to become enabled or disabled must be handled specially, as they affect scheduling decisions. This includes the forking of additional processes, which requires both modelling and engineering effort to make the testing tool work across different processes. For the above reasons, a large number of benchmarks in the CB and RADBenchmark suites had to be skipped because they involve testing servers, using several processes and network communication. Modelling network communication and testing multiple processes are both non-trivial tasks. We believe the difficultly of controlling nondeterminism and synchronisation is a key issue in applying SCT to existing code bases. However, note that non-SCT techniques can handle programs with nondeterminism and unmodelled synchronisation, depending on how the techniques are implemented; for example, controlled random scheduling does not require deterministic programs (although bug replay will be unreliable) and blocking synchronisation functions can be detected approximately (on-the-fly) using heuristics [NBMM12].

Isolated concurrency testing An alternative to creating an environment model that can be reused is to create isolated tests that test one component against one-off "mock" versions of any dependent components. In this way, any nondeterminism in the mock components can be fixed. This is similar to unit testing but with multiple threads. Unfortunately, we found that many programs are not designed in a way that makes this easy. An example is the Apache httpd web server. The server module that we inspected had many dependencies on other parts of the server and called system functions directly, making it difficult to create an isolated test case. Apache developers test the server as a

whole; network packets are sent to the server by a script running in a separate process. Note that it is also difficult to apply (sequential) unit testing to such software.

Many applications in the CB benchmarks use global variables and function-static variables that are scattered throughout several source files. These would need to be handled carefully with SCT tools that require a repeatable function to test, such as CHESS, in which the state must be reset when the function returns. This is not a problem for Maple, which restarts the test program for every schedule explored.

Stress tests Some of the benchmarks we obtained were stress tests, such as those in RADBench. These benchmarks create a large number of threads that undertake a significant amount of computation to increase the chance of exploring an unlikely interleaving under the OS scheduler. Increasing the amount of work is often achieved by increasing the size of the inputs or making threads execute work in a loop. In the context of SCT, this is extremely inefficient and unnecessary; instead, the number of threads and other parameters should be reduced as much as possible, as SCT ensures that many interleavings will be explored. Artificially increasing the thread count and parameters to make the benchmark "harder" is not representative of how one should use SCT for bug-finding. Thus, we chose the minimum thread counts and parameters when converting stress tests and CPU performance benchmarks; for example, the PARSEC benchmarks accept "number of threads" (t) and "input size" parameters, which we set to "two threads" and "test size" (the smallest input size option), respectively. However, note that many benchmarks still create more than t threads; for example, the `ferret` benchmark creates a pipeline of threads where each stage in the pipeline contains t threads. In practice, one may also increase the thread count and other parameters iteratively, in case there exist bugs that depend on higher thread counts or parameter values. However, prior work suggests that most concurrency bugs only require certain orderings between a small number of threads (typically two) [LPSZ08]. There was one instance where we knew we had to increase the thread count above the minimum for the bug to manifest; for the `streamcluster2` benchmark from the PARSEC benchmark suite, we changed the "number of threads" parameter, t, from two to three.

Memory safety We found that certain concurrency bugs manifest as out-of-bounds memory accesses, which do not always cause a crash. We implemented an out-of-bounds memory access detector on top of Maple, which allowed us to detect a previously unknown bug in the PARSEC `streamcluster3` benchmark. Unfortunately, detecting out-of-bound memory accesses is a non-trivial problem and our implementation had many

false-positives where memory allocation was missed or where libraries access bookkeeping information that lies outside of `malloc`ed regions. Furthermore, the extra instrumentation code caused a slow-down of up to 8x; Maple's existing information on allocated memory was not designed to be speed-efficient. We disabled the out-of-bound access detector in our experiments, but we note that a production quality concurrency testing tool would require an efficient method for detecting out-of-bound accesses to automatically identify this important class of bug. We manually added assertions to detect the (previously unknown) out-of-bounds access in `streamcluster3` and the (previously known) out-of-bounds access in `fsbench_bad` in the CS benchmarks. Out-of-bounds accesses to synchronisation objects, such as mutexes, are still automatically detected; this was used to detect the bug in `pbzip2` from the CS benchmarks.

Data races We found that 30 of the 49 benchmarks contained data races. There are many compelling arguments against the tolerance of data races [Boe11], and according to the C++11 standard, if it is possible for a program execution to lead to a data race, the behaviour of the program for this execution is undefined. Nevertheless, at the level of program binaries, data races do not result in undefined behaviour and many data races are not regarded as bugs by software developers. Treating data races as errors would be too easy for benchmarking purposes, as they hide the more challenging bugs that the benchmarks capture. A particular pattern we noticed was that data races often occur on flags used in ad-hoc busy-wait synchronisation, where one thread keeps reading a variable until the value changes. At the C++ level, the "benign" races could be rectified through the use of C++11 relaxed atomics, the "busy-waits" could be formalised using C++11 acquire/release atomics, and synchronisation operations could be added to eliminate the buggy cases. However, telling the difference between benign and buggy data races is non-trivial in practice [KZC12, N$^+$07]. We explain how we treat data races in our study in §3.6.

Output checking The bugs in the benchmarks `CB.aget` and `parsec.streamcluster2`, lead to incorrect output, as documented in the bug descriptions. Thus, we added extra code to read the output file and trigger an assertion failure when incorrect; the output checking code for the `CB.aget` was provided as a separate program, which we added to the benchmark. Several of the PARSEC and SPLASH benchmarks do not verify their output, greatly limiting their utility as test cases.

Busy-wait loops As explained in §3.3, several benchmarks use busy-wait loops. As explained, we added a call to *yield* in every busy-wait loop and modified Maple to react appropriately.

3.5 Research questions

Our aim was to use SCTBench to empirically compare the following techniques: an unbounded depth-first search, iterative preemption bounding, iterative delay bounding, PCT, and controlled random scheduling, to answer the following research questions (RQs):

RQ1 Which technique performs the best in terms of bug finding ability?

RQ2 Does PCT beat the other techniques as in prior work [BKMN10]?

RQ3 How effective is the controlled random scheduler (a naïve technique that is rarely tested in prior work) in comparison to the other techniques?

RQ4 Does delay bounding beat preemption bounding, as in prior work [EQR11], and do both schedule bounding techniques beat a straightforward unbounded depth-first search, as in prior work [MQ07b, EQR11]?

RQ5 How many bugs can be found with a small number of preemptions/delays, and can we find non-synthetic examples of concurrency bugs that require more than three preemptions (the largest number of preemptions required to expose a bug in previous work [EQR11])?

RQ6 How easy is it to apply SCT to various existing code bases in practice?

RQ7 Can we classify certain benchmarks exhibiting defects as highly trivial or non-trivial, based on the ease or difficulty with which the techniques we study are able to expose defects?

We answer **RQ1**–**RQ5** quantitatively by investigating the number of bugs found by each technique within a schedule limit, showing how these numbers vary as the schedule limit is increased. We answer **RQ6** qualitatively, based on our experience collecting and modifying benchmarks during the construction of SCTBench. To answer **RQ7**, we identify a number of properties of benchmarks that indicate when bug-finding is trivial, and report on the extent to which SCTBench examples exhibit these trivial properties. We also report on benchmarks that appear to present a challenge for SCT techniques, based on the fact

that associated defects were missed by several (or, in one case, all) of the techniques we study.

3.6 Experimental Method

Our experimental evaluation aims to compare unbounded depth-first search (DFS), iterative preemption bounding (IPB), iterative delay bounding (IDB), controlled random scheduling (Rand) and probabilistic concurrency testing (PCT). We also test the default Maple algorithm (MapleAlg). Bugs are deadlocks, crashes, assertion failures and incorrect output. Each benchmark contains a single concurrency bug.[5]

For each SCT technique, we use a limit of 100,000 terminal schedules to enable a full experimental run over our large set of benchmarks to complete on a cluster within one month. There are two exceptions: for the `chess.IWSQWS` and `chess.SWSQ` benchmarks (see Table 3.3 and 3.4) we use a terminal schedule limit of 10,000; for these longer-running benchmarks, evaluation with the higher schedule limit exceeded our one month time restriction. We henceforth assume use "schedule" to refer to "terminal schedule" for brevity.

We chose to use a schedule limit instead of a time limit because there are many factors and potential optimisation opportunities that can affect the time needed for a benchmark to complete; we believe that the time variance for the different techniques (for execution of a single schedule of a given benchmark) is negligible, assuming reasonably optimised implementations. Furthermore, the cluster we have access to shares its machines with other jobs, making accurate time measurement difficult. In contrast, the number of schedules explored cannot be improved upon, without changing key aspects of the search algorithms themselves. By measuring the number of schedules, our results can potentially be compared with other algorithms and future work that use different implementations with different overheads.

Each benchmark goes through the following phases:

Data Race Detection Phase When detecting safety property violations using a technique that ensures full coverage (like an unbounded DFS), it is sound to only consider scheduling points before each synchronisation operation, such as locking a mutex, and not memory accesses, as long as execution aborts with an error as soon as a data race is detected [MQB+08]. Thus, if there are data races, an error will be reported; if there

[5] In fact, we *assume* each benchmark contains a single concurrency bug; a schedule that finds a deadlock, crash, assertion failure or incorrect output is deemed to have found the underlying bug in the benchmark, even though, for example, a different schedule may trigger a different assertion to fail.

are no data races and the search completes, then the program is free from safety property violations (such as assertion failures and deadlocks). This greatly reduces the number of schedules that need to be considered as memory accesses are not regarded as scheduling points. However, treating data races as errors is not practical for this study (see §3.4.2).

Thus, as in previous work [YNPP12], we circumvent this issue by performing dynamic data race detection to identify a subset of load and store instructions that are known to participate in data races. We treat these instructions as visible operations during concurrency testing by inserting scheduling points before them. For each benchmark, we execute Maple in its data race detection mode ten times, without controlling the schedule. Note that data race detection is nondeterministic, since the schedule is not controlled. Each racy instruction (stored as an offset in the binary) is treated as a visible operation in subsequent phases. We also tried detecting data races during concurrency testing, but this caused an additional slow-down of up to 8x, as Maple's data race detector is not optimised for this scenario.

Thus, the techniques explore the sequentially consistent outcomes of a subset of the possible data races for a concurrent program. Bugs found by this method are real (there are no false-positives), but bugs that depend on relaxed memory effects or data races not identified during the dynamic data race detection phase will be missed. We do not believe these missed bugs threaten the validity of our comparison, since the same information about data races is used by all of the techniques (excluding the Maple algorithm); the set of racy instructions could be considered as part of the benchmark.

An alternative to under-approximation would be to use static analysis to over-approximate the set of racy instructions. We did not try this, but speculate that imprecision of static analysis would lead to many instructions being promoted to visible operations, causing schedule explosion.

Note that the data races detected and used in our experiments are different from those in our original study [TDB14] because the data race detection phase is nondeterministic.

Depth-First Search (DFS) Phase We next perform SCT using a DFS, with no schedule bounding and a limit of 100,000 terminal schedules.

Iterative Preemption Bounding (IPB) Phase We next perform SCT on the benchmark using iterative preemption bounding. By repeatedly executing the program, restarting after each execution, we first explore all schedules that have zero preemptions, followed by all schedules that have precisely one preemption, etc., until either the limit of 100,000 schedules is reached, all schedules have been explored or a bug is found. If a bug is found,

the search does not terminate immediately; the remaining schedules within the current preemption bound are explored (for our set of benchmarks, it was always possible to complete this exploration without exceeding the schedule limit). This allows us to check whether non-buggy schedules could exceed the schedule limit when an underlying search strategy other than our DFS approach is used (see §3.3).

In practice, all SCT tools that we are aware of do not perform iterative preemption bounding in this manner. Instead, with a preemption bound of c, it is necessary to explore all schedules with c *or fewer* preemptions due to the use of a DFS. Thus, iterative preemption bounding will explore all schedules with 0 preemptions, followed by all schedules with 0–1 preemptions (redundantly re-exploring schedules with 0 preemptions), followed by all schedules with 0–2 preemptions (redundantly re-exploring schedules with 0–1 preemptions), etc. In our study, we simulate an optimised version of preemption bounding that does *not* redundantly re-explore schedules with fewer than c preemptions. We achieve this simply by ignoring previously explored, and thus redundant, schedules when processing our log files. We chose to do this because it might be possible to implement such an algorithm in practice and we did not want to unfairly penalise the technique due to the specific implementation that we used. In particular, we note work that uses compressed schedules to store the unexplored schedule-tree [CBM10].

Iterative Delay Bounding (IDB) Phase This phase is identical to the previous, except delay bounding is used instead of preemption bounding.

Random scheduler (Rand) Phase We run each benchmark 100,000 times using Maple's controlled random scheduler mode. This allows us to compare the other techniques against a naïve controlled scheduler. Recall that the random scheduler may re-explore schedules.

Probabilistic Concurrency Testing (PCT) Phase Recall that PCT requires parameters n (maximum number of threads), k (maximum number of execution steps) and d (the "bug depth", which controls the number of priority change points that will be chosen). In order to experiment with PCT using varying values for d, it was necessary to obtain reasonable estimates for n and k. We obtained these estimates for each benchmark as follows. First, we used results related to SCTBench obtained in prior work to provide initial estimates for n and k—see Table 3, column "# threads" and "# max scheduling points" in [TDB14]. Using these initial estimates we executed 1,000 schedules of the benchmark using PCT with d=3. We chose d=3 as we believed that this would increase the amount

of interleaving, potentially increasing the chance of observing different execution lengths. During these executions we recorded the maximum observed number of threads and the maximum observed number of steps; we start counting steps from when the initial thread first launches a second thread. We used these values for n and k, respectively, in our experiments[6].

Unlike the other bounded techniques, there is no obvious way to perform iterative PCT. In order to provide a thorough evaluation of PCT, we experimented with each d in $\{1, 2, 3\}$, using PCT to run each benchmark for 100,000 executions for each value of d. We present each version of PCT (parameterised with a value for d) as a separate technique.

Maple Algorithm (MapleAlg) Phase We test each benchmark using the Maple algorithm. This algorithm terminates based on its own heuristics; we enforced a time limit of 24 hours per benchmark, although execution only took this long due to a livelock bug in the Maple tool.

Notes on DFS and POR As discussed in §4.5, the SCT techniques we evaluate are built on top of Maple's default DFS strategy. Although DFS is just one possible search strategy, and different strategies could give different results, we argue that this is not important in our study. First, if the DFS biases the search for certain benchmarks, then all DFS techniques are likely to benefit or suffer equally from this. Second, iterative schedule bounding explores *all* schedules with c preemptions/delays before *any* schedule with $c+1$ preemptions/delays. This means that when the first schedule with $c+1$ preemptions/delays is considered, exactly the same set of schedules, regardless of search strategy, will have been explored so far. Thus, if a bug is revealed at bound c then, by exploring *all* schedules with bound c (as described above), we can determine the worst case number of schedules that might have to be explored to find a bug, accounting for an adversarial search strategy.

We do not attempt to study the various POR techniques [MQB+08, MQ07a, FG05, AAJS14] in this study. This is because (a) our principle aim was to validate the findings of prior works on schedule bounding, most of which do not incorporate full POR (and indeed, the relationship between POR and schedule bounding is complex [CMM13, MQ07a, HF11]), (b) we already include a large number of techniques, and (c) as noted above, Maple's data race detector is not well-optimised and thus infeasible for use during SCT; the information stored for data race detection is similar to that needed for POR tech-

[6] We note that, in hindsight, this may be an unrealistic approach to obtaining the parameters. A better approach would be as follows: (1) Choose any values for n and k. (2) Execute PCT for e.g. 1,000 schedules, recording the maximum observed number of threads and steps. (3) Update n and k based on what was observed. (4) Repeat the process to refine the values for n and k.

niques such as DPOR, and so performing POR was deemed infeasible without significant engineering effort. We consider POR in Chapter 4.

Notes on randomisation The controlled random scheduler and PCT techniques both use a random number generator. Given one of these techniques, a seed (used to initialise the random number generator) and a benchmark, the (single) schedule executed by the technique for the benchmark is deterministic. Unlike the DFS techniques, the random techniques have no implied order between schedules: two different seeds result in two independent schedules that can be tested in parallel.

Our method for testing the controlled random scheduler and PCT techniques was as follows. We used a fixed initial seed to generate a single list of 100,000 seeds using a random number generator; we used these same seeds for all benchmarks and for all randomised techniques to produce 100,000 schedules in each case.

For a given benchmark, we can use the number of buggy schedules out of 100,000 (i.e. the proportion of buggy schedules) to compare the random-based techniques; although this is dependent on the initial seed, as the schedule limit is increased, we would expect this to become stable. We can also use the number of schedules before the bug is found. However, this is very dependent on the initial seed and a technique may "get lucky" for some benchmarks. Thus, we can instead consider the "average number of schedules needed to expose a bug", calculated using: 100,000 / "number of buggy schedules"; this shows how many schedules are likely to be needed *on average* before a bug is found. As the schedule limit is increased, we would expect this number to become stable and, thus, be independent of the initial seed.

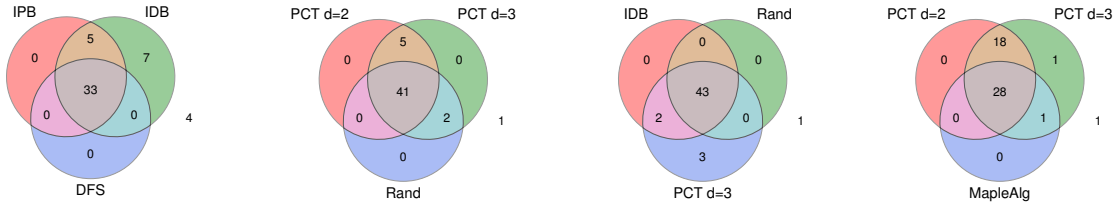
3.7 Experimental Results

We conducted our experiments on a Linux cluster, with Red Hat Enterprise Linux Server release 6.4, an x86_64 architecture and gcc 4.7.2. Our modified version of Maple is based on the last commit from 2012.[7] The benchmarks, scripts and the modified version of Maple used in our experiments can be obtained from:

`https://github.com/mc-imperial/sctbench`.

Throughout this section, we use **RQ1**–**RQ7** to indicate that an observation relates to one of the research questions posed in §3.5. When we refer to x buggy schedules, we mean the x schedules executed by a particular technique that found the bug in a given

[7] `http://github.com/jieyu/maple` commit at Sept 24, 2012



(a) DFS-based. (b) Randomisation. (c) DFS-based vs. randomisation. (d) MapleAlg.

Figure 3.3: Venn diagrams showing number of benchmarks in which the bugs were found with the various techniques.

benchmark. When we refer to x bugs being found by a technique, we mean that the technique found a bug in x of the benchmarks.

For **RQ6**, we refer the reader to §3.4.2, where we discuss the difficultly of applying SCT to the benchmarks.

3.7.1 Venn diagrams

The Venn diagrams in Figure 3.3 give a concise summary of the bug-finding ability of the techniques in terms of number of bugs found in SCTBench within the schedule limit.

Figure 3.3a summarises the bugs found by the DFS-based techniques. In relation to **RQ4**, the figure shows that IPB was superior to DFS, finding all 33 bugs found by DFS, plus an additional 5. The figure also shows, also in relation to **RQ4**, that IDB found all 38 bugs found by IPB, plus an additional 7. The bugs in 4 benchmarks were missed by all DFS-based techniques; we discuss this further below.

Figure 3.3b shows the bugs found by the randomisation techniques, PCT and Rand. We show the results for PCT with d=2 and d=3 because PCT found the most bugs when using these values for d. The results show that PCT d=3 performed the best in terms of number of bugs found within the schedule limit, finding 48 bugs, including all those found by the other techniques (see Figure 3.3c and 3.3d also). Thus, in answer to **RQ1** and **RQ2**, the results show that PCT d=3 is the most capable technique at finding bugs in SCTBench; this concurs with findings of prior work in which PCT found bugs faster than IPB [BKMN10].

Figure 3.3c shows the bugs found by the superior schedule bounding technique (IDB), the random scheduler (Rand) and PCT with d=3 (the most successful configuration of PCT). Note that the bugs in 43 benchmarks were found by both IDB and Rand, and IDB found just 2 additional bugs that were missed by Rand. Although not shown in these diagrams, Rand also found *all* the bugs found by IPB, plus an additional 5. Thus, in

answer to **RQ3**, Rand performed better than IPB in terms of number of bugs found and was not far behind IDB. Furthermore, Rand found the bugs in fewer schedules than IDB for 21 of the benchmarks. A similar observation can be made about IPB and Rand. Thus, Rand was often faster at finding bugs than schedule bounding. We discuss the surprising results for Rand below.

Figure 3.3d shows the bugs found by MapleAlg vs. PCT d=2 and PCT d=3. Maple found 29 of the 49 bugs (all of which were also found by PCT d=3) and missed 19 bugs that were found by PCT d=3.

The bug in `misc.safestack` was missed by *all* techniques; we discuss this in more detail below.

3.7.2 Cumulative plots

The graphs in Figures 3.4 and 3.5 give an alternative summary of the techniques.

Figure 3.4 is a *cumulative* plot showing the number of bugs found (y-axis) after x schedules (x-axis) for each technique over all the benchmarks. Each line represents a technique and is labelled by the name of the technique and the number of bugs found by the technique within the schedule limit. If a given technique has a point at coordinate (x, y) then there were y benchmarks for which the technique was able to expose a bug using x schedules or fewer, i.e. for which "number of schedules to first bug" is less than or equal to x. This plot shows the number of bugs that would be found by the techniques using schedule limits lower than 100,000. For example, with our schedule limit of 100,000, IDB and Rand found 45 and 43 bugs, respectively; with a schedule limit of 1,000, they would have found 40 and 42 bugs, respectively.

As explained in §3.6, the Rand and PCT results are specific to the random seeds used during our experiments. Thus, in Figure 3.5, we present results using the average number of schedules needed to expose a bug, which is given by: 100,000 / "number of buggy schedules". Figure 3.5 is similar to Figure 3.4, but includes only PCT d=3 and Rand (otherwise, the graph is overcrowded). The additional dashed lines show the *average* behaviour of the techniques.

Observe that, in Figure 3.4, the ordering of the techniques by number of bugs found remains fairly consistent for schedule limits above 1,000, the exception being IDB and Rand, with IDB overtaking Rand in terms of bug-finding ability at 2990 schedules. In Figure 3.5, the same is true when considering the average behaviour of the techniques. Thus, the number of bugs found by the techniques within our schedule limit is, for the most part, an accurate reflection of the bug finding ability of the techniques on our benchmarks.

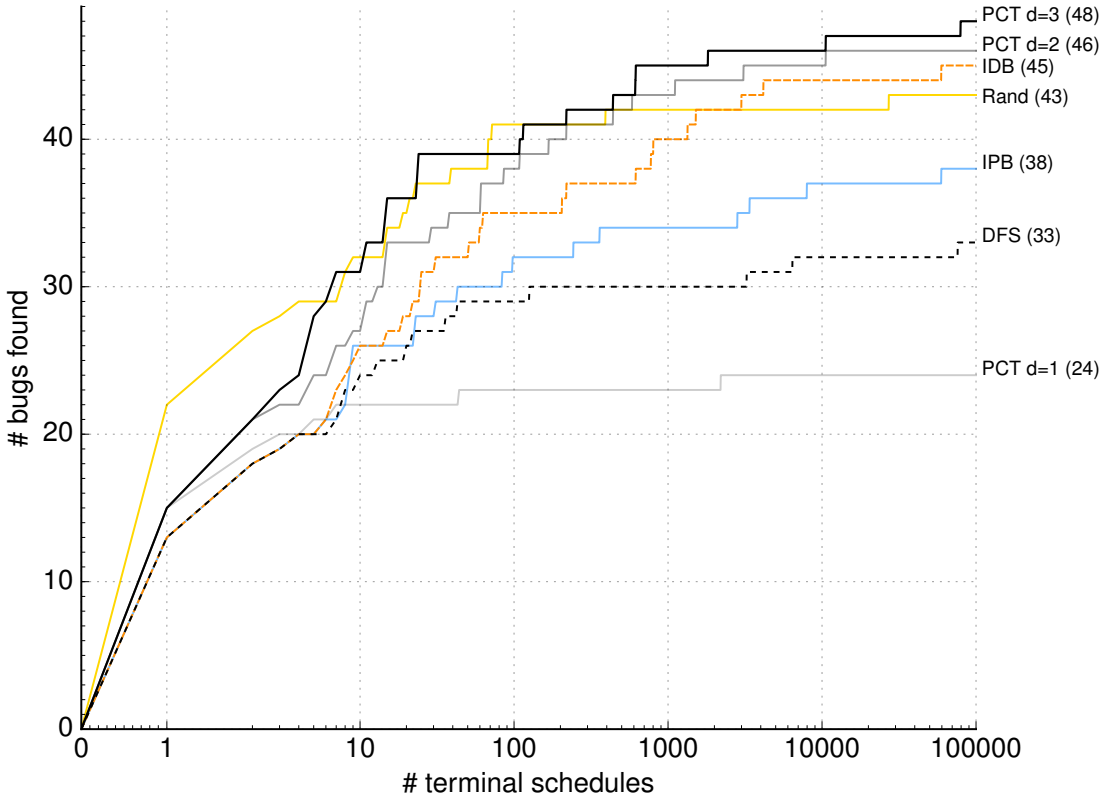


Figure 3.4: Cumulative plot, showing, for each SCT technique, the number of bugs found after x schedules over all the benchmarks. The plot is intended to be viewed in colour.

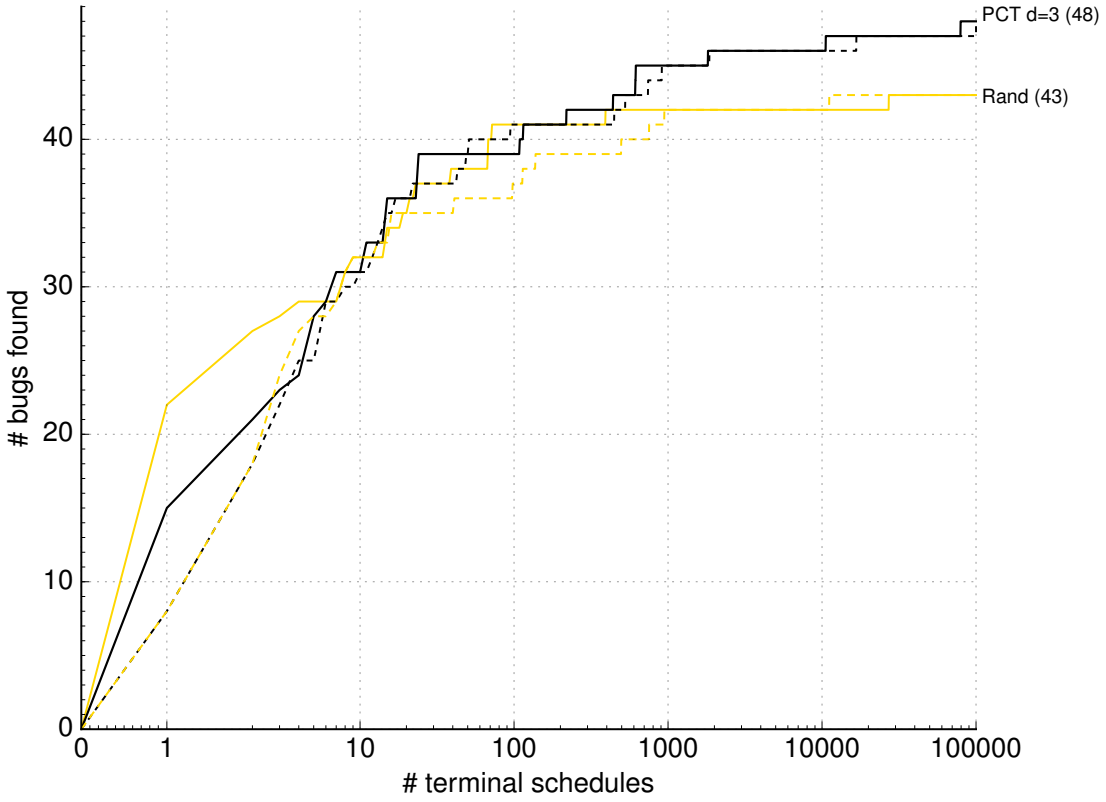


Figure 3.5: For PCT d=3 and Rand, compares the number of bugs found after x schedules as in Figure 3.4 (solid lines) with the *average* behaviour of the techniques (dashed lines). The plot is intended to be viewed in colour.

Our results show that PCT d=3 almost invariably finds more bugs than the other techniques, unless the schedule limit is extremely low. Thus, our findings for **RQ1** and **RQ2** apply for a range of schedule limits. One exception is that Rand overtook PCT d=3 at 100 schedules, but in the average case (Figure 3.5), PCT d=3 is still consistently above Rand when the schedule limit is 20 or higher. Regarding **RQ4**, the findings that IDB found more bugs than IPB and that IPB found more bugs than DFS both hold for schedule limits of 50 or higher; the difference in bugs found between these techniques increased with the schedule limit. Similarly, for **RQ3**, Rand beat IPB for *all* schedule limits up to and including 100,000, showing that this finding is not simply due to our choice of schedule limit. In the average case, Rand beat IPB for for all schedule limits of 10 or greater, indicating that for non-trivial limits this finding is independent of our choice of initial random seed. Rand was also ahead of IDB in terms of bugs found between schedule limits 1–1,000, giving further evidence for **RQ3** that Rand performed well. In fact, Rand found 27 bugs in the first 2 schedules and was ahead of all other techniques by at least 6 bugs; Figure 3.5 shows that this is not the case on average, but after 10 schedules, both Rand and averaged Rand found 32 bugs, which is the same as PCT d=3 (and more than all other techniques). The fact that Rand finds many of the bugs so quickly is evidence of the trivial nature of some of the benchmarks (**RQ7**), which we discuss in §3.7.4.

Regarding the average behaviour of PCT d=3 and Rand (Figure 3.5), both techniques still performed well and our main conclusions do not change. We can see that Rand was slightly "lucky" between 10–1,000 schedules compared to the average case and was slightly "unlucky" at finding the bug after 10,000 schedules.

3.7.3 Results tables

The full set of experimental data gathered for our benchmarks is shown in Tables 3.3 and 3.4. As explained in §3.6, we focus on the number of schedules explored rather than time taken for analysis. The execution time for one schedule of a single benchmark varied between 1–10 seconds depending on the benchmark. The longest time taken to perform ten data race detection runs for a single benchmark was five minutes, but data race detection was significantly faster in most cases. Data race detection could be made more efficient using an optimised, state-of-the-art method. Because data race analysis results are shared between all techniques (except MapleAlg), the time for data race analysis is not relevant when comparing these methods.

For each benchmark, *# max threads* and *# max enabled threads* show the total number

of threads launched and the maximum number of threads simultaneously enabled at any scheduling point, respectively. The *# max steps* column shows the maximum number of scheduling points (visible operations) k observed from when the initial thread first launches a second thread. As explained in §3.6, these numbers were obtained by running 1000 executions of PCT on the benchmarks.

Results for DFS-based techniques In Table 3.3, the smallest preemption or delay bound required to find the bug for a benchmark, or the bound reached (but not fully explored) if the schedule limit was hit, is indicated by *bound*; *# schedules to first bug* shows the number of schedules that were explored up to and including the detection of a bug for the first time; *# schedules* shows the total number of schedules that were explored; *# new schedules* shows how many of these schedules have exactly *bound* preemptions (for IPB) or delays (for IDB); *# buggy schedules* shows how many of the total schedules explored exhibited the bug. As explained in §3.6, when a bug is found during IPB or IDB, we continue to explore all buggy and non-buggy schedules within the preemption or delay bound; the schedule limit was never exceeded while doing this. An *L* entry denotes 100,000 (the schedule limit discussed in §3.6). When no bugs were found, the bug-related columns contain ✗. We indicate by *% buggy*, the percentage of schedules that were buggy out of the total number of schedules explored during DFS. We prefix the percentage with a '*' when the schedule limit was reached, in which case the percentage applies to all explored schedules, not the total number of possible schedules.

Results for randomisation techniques For the Rand and PCT techniques in Table 3.4, the *# schedules* column is omitted, as it is always 100,000 (although, as explained in §3.6, the `chess.IWSQWS` and `chess.SWSQ` benchmarks use a lower schedule limit of 10,000). This is because these techniques do not maintain a history of explored schedules and thus there is no notion of the search terminating. The *# schedules to first bug* column shows the number of schedules that were explored up to and including the detection of a bug for the first time. The *# buggy schedules* column shows how many of the 100,000 schedules exhibited a bug. For each value of d that we used for PCT and for each benchmark, we estimate the worst case (smallest) number of buggy schedules that we should find given a bug of depth d, parameters n and k from the benchmark, and our schedule limit of 100,000. This estimate is shown under *est. worst case # buggy* in Table 3.4, and is calculated by computing the worst-case probability that an execution using PCT will expose a depth-d bug (using the formula $1/nk^{d-1}$ discussed in §3.2.5) and multiplying this probability by 100,000 (the schedule limit). Of course, the estimate for each d is only relevant if the bug

Property	# benchmarks
Bug was found with a delay bound of 0	13
Total number of schedules $< 100{,}000$	18
$> 50\%$ of random terminal schedules were buggy	18
Every random terminal schedule was buggy	8

Table 3.2: Benchmarks where bug-finding is arguably trivial.

associated with a benchmark can in fact manifest with depth d.

Results for MapleAlg For the Maple algorithm, we report whether the bug was found (the *found?* column in Table 3.4), the total number of (not necessarily distinct) schedules explored, as chosen by the algorithm's heuristics, and the total time in seconds for the algorithm to complete. Benchmarks 32, 33 and 34 caused Maple to livelock, so the 24 hour time limit was exceeded. We indicate this with '-'.

3.7.4 Benchmark Properties

The *# max threads* and *# max steps* columns from the results tables can be used to estimate the total number of schedules, which may shed light on the complexity of a given benchmark. With at most n enabled threads and at most k steps, there are at most n^k terminal schedules. On the other hand, if most of the schedules are buggy then the number of schedules is not necessarily a good indication of bug complexity. For example, `CS.din_phil2_sat` has a relatively high number of schedules, but since 87% of them are buggy (see the DFS results in Table 3.3), this bug is trivial to find. Of course, the majority of benchmarks cannot be explored exhaustively, and estimating the percentage of buggy schedules from the partial DFS results is problematic because DFS is biased towards exploring deep context switches.

To answer **RQ7**, we present Table 3.2 which provides some further insight into the complexity of the benchmarks, using properties derived from Tables 3.3 and 3.4. Bugs found with a delay bound of zero (13 cases) will always be found on the initial schedule for IPB, IDB and DFS, as they all initially execute the same schedule. Any technique based on this same DFS will also find the bug immediately. We argue that the bugs in question are trivial since the schedule includes minimal interleaving (there are no preemptions). Benchmarks with fewer than 100,000 schedules total (as measured by unbounded DFS, which is exhaustive) will always be exhaustively explored (and so the bug will be found) by all DFS-based techniques (18 cases). Techniques can still be compared on how quickly they find the bugs in such benchmarks. Note that the two `chess` benchmarks that were ex-

plored using a schedule limit of 10,000 do not have fewer than 100,000 schedules. Bugs that were exposed more than 50% of the time when using the random scheduler could arguably be classified as "easy-to-find" (18 cases). Among these, bugs that were exposed 100% of the time when using the random scheduler (8 cases) are almost certainly trivial to detect; indeed, Tables 3.3 and 3.4 show that all of these benchmarks were buggy for all schedules explored by *all* techniques. For 5 of these benchmarks, DFS was exhaustive, showing that these bugs are not even schedule-dependent. Note that the `CS.din_phil7_sat` benchmark contains fewer schedules than the smaller versions of this benchmark and has 100% buggy schedules according to DFS. This is because `CS.din_phil7_sat` contains an additional, unintentional bug introduced by the original authors of the benchmark; when we converted the benchmark to use (non-recursive) pthread mutexes, the bug causes additional deadlocks. We did not fix this additional bug and instead used the benchmark as it was found.

Regarding **RQ7**: in our view the relatively trivial nature of some of the bugs exhibited by our benchmarks has not been made clear in prior work that studies these examples (prior to the conference version of this work [TDB14]). The controlled random scheduler can detect many of the bugs with a high probability. We regard these easy-to-find bugs as having value only in providing a minimum baseline for any respectable concurrency testing technique. Failure to detect these bugs would constitute a major flaw in a technique; detecting them does not constitute a major achievement.

3.7.5 Techniques In Detail

IPB vs. IDB Figure 3.6 compares IPB and IDB by plotting data from the following columns in Table 3.3: *# schedules to first bug* (as a cross) and *# schedules* (as a square). All benchmarks are shown for which at least one of the techniques found a bug. A benchmark is depicted as a line connecting a cross and a square. Each square is labelled with its benchmark *id* from Table 3.3. Where the bug was not found by one of the techniques, this is indicated with a cross at 100,000 (the schedule limit discussed in §3.6). However, as described in §3.6, benchmarks 33 and 34 used a schedule limit of 10,000 and so the crosses for these benchmarks on the line y =10,000 indicate that IPB hit the schedule limit without finding the bug. The cross indicates which technique was faster at finding the bug; crosses below/above the diagonal indicate that IPB/IDB was faster. The square indicates how many schedules exist with a bound less than or equal to the bound that found the bug. For example, when exploring benchmark 30 with IPB, the first buggy schedule is found after 243 schedules. This schedule involves one preemption, so the

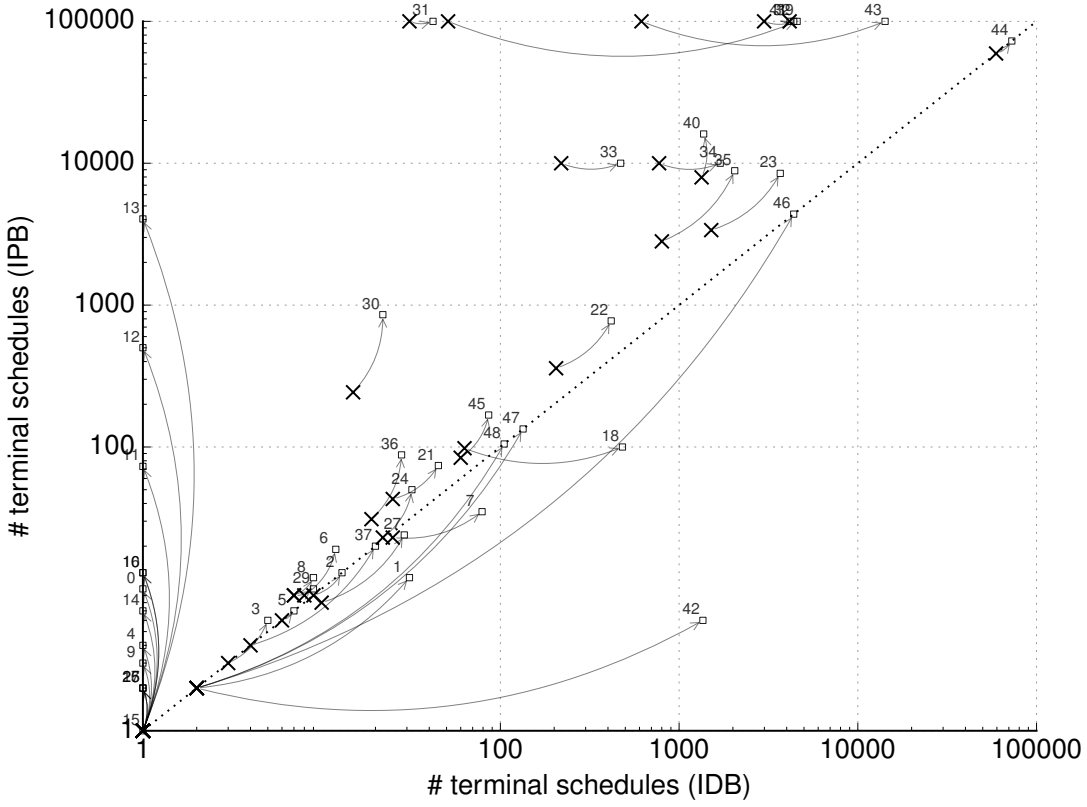


Figure 3.6: Comparison of IPB and IDB, showing the number of schedules to the first bug (cross) connected to the total number of schedules (square), up to the bound that found the bug. Squares are labelled with the benchmark id.

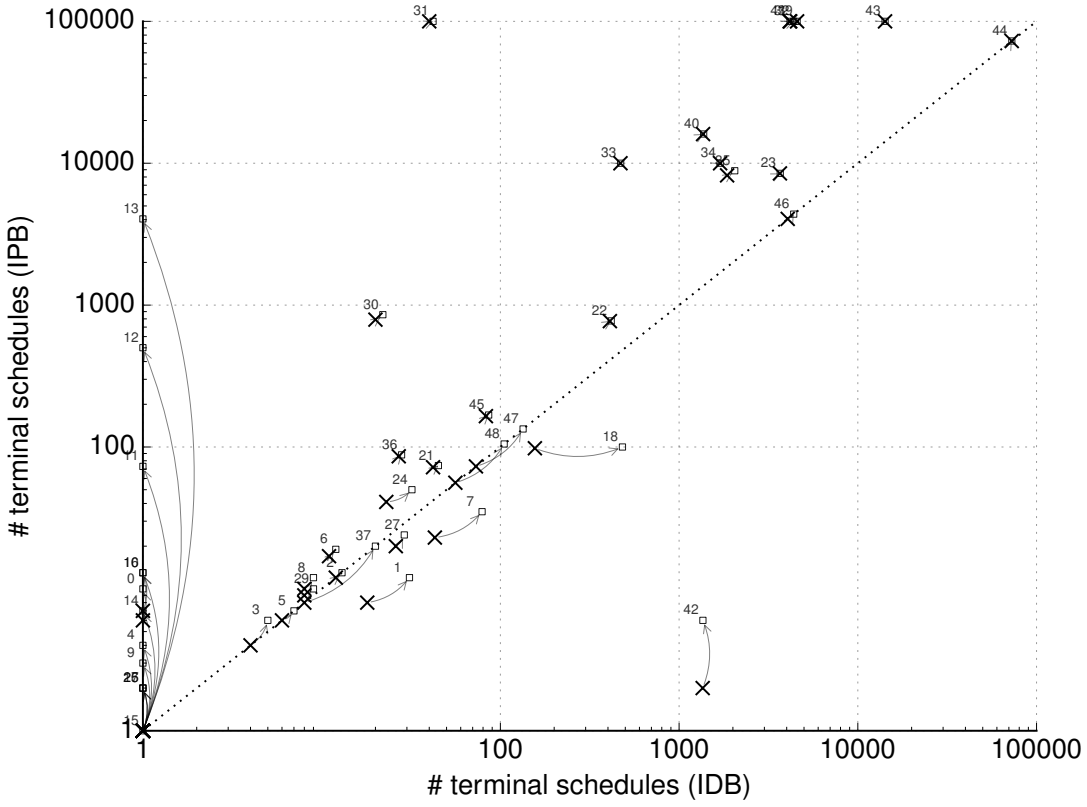


Figure 3.7: Comparison of IPB and IDB, showing total number of non-buggy schedules (cross) connected to the total number of schedules (square), up to the bound that found the bug. Squares are labelled with the benchmark id.

search continues until all 856 schedules with at most one preemption have been explored (bound at which the bug was found). Since the search terminated before reaching the schedule limit, we know that the bug would be found within the first 856 schedules even if we were using an underlying search strategy other than our DFS. Notice that a number of benchmarks appear at $(x, 100{,}000)$, with x <100,000: this is where IPB failed to find a bug and IDB succeeded (except for benchmarks 33 and 34, as explained above).

The bug-finding ability of the techniques in Figure 3.6 is tied to the underlying DFS. It is possible that this might cause one of the techniques to "get lucky" and find a bug quickly, while another search order could lead to many additional non-buggy schedules being considered before a bug is found. To avoid this implementation-dependent bias, in Figure 3.7 we consider the *worst-case* bug-finding ability. For each benchmark, a cross plots, for IDB and IPB, the total number of *non-buggy* schedules within the bound that exposed the bug. This corresponds to the difference between the *# schedules* and *# buggy schedules* columns presented in Table 3.3, and represents the worst-case number of schedules that might have to be explored to find a bug, given an unlucky choice of search ordering. The squares are the same as in Figure 3.6.

Overall, IDB finds all bugs found by IPB, plus an additional seven. Regarding **RQ4**: in Figure 3.6, most crosses fall on or above the diagonal, showing that IDB was as fast or faster than IPB in terms of number of schedules to the first bug. The same is mostly true for the squares, showing that IDB generally leads to a smaller total number of schedules than IPB (up to the bound at which the bug was found). In the worst case (Figure 3.7), some crosses fall under the line, but most are still very close, or represent a small number of schedules (less than 100) where the difference between the techniques is negligible. An outlier is benchmark 42 where, *in the worst case*, IPB requires 3 schedules to find the bug, while IDB requires 1356 schedules. Table 3.3 shows that the bug does not require any preemptions, but requires at least one delay; this difference greatly increases the number of schedules for IDB. We believe this can be explained as follows. First, there must be a small number of blocking operations, leading to a very small number of schedules with a preemption bound of zero. Second, the bug in question requires that when two particular threads are started and reach a particular barrier, the "master" thread (the thread that was created before the other) does *not* leave the barrier first. With zero preemptions, the non-master thread can be chosen at the first blocking operation (as any enabled thread can be chosen). With zero delays, only the master thread can be chosen, as one delay is required to skip over the master thread. Thus, this is an example where IDB performs worse than IPB. Nevertheless, IDB is still able to find the bug within the schedule limit.

The `CS.reorder_X_bad` benchmark (where X is the number of threads launched – see

Table 3.3) is the adversarial delay bounding example given in Figure 3.2 in §3.2.3; the smallest delay bound required for the bug to manifest is incremented as the thread count is incremented. However, IDB still performs better than IPB, as the number of schedules in IPB increases exponentially with the thread count. Furthermore, this is a synthetic benchmark for which the bug is found quickly by both techniques with a low thread count.

Effectiveness of SCT In answer to **RQ3**, we have shown above that Rand is surprisingly effective, finding more bugs than IPB and almost as many as IDB. The cumulative plots in Figure 3.4 and 3.5 show that these findings apply on average and for various schedule limits. A possible intuition for this is as follows. If a bug can be exposed with just one preemption, say, then there may be many scheduling points at which the preemption can occur so that the bug can be exposed. Furthermore, there may be a number of "unexpected" operations in other threads that will cause the bug to trigger (e.g. writing to a variable that the preempted thread is about to access). Any schedule where (a) the preemption occurs in a suitable place, and (b) additional preemptions do not prevent the bug from occurring, will also expose the bug. There may be many such schedules and thus a good chance of exposing the bug through random scheduling. More generally, one might suggest that if a bug can be exposed with a small delay or preemption count, there may be a high probability that a randomly selected schedule will expose the bug. A counter-example is the bug in the `parsec.ferret` benchmark, which is missed by Rand but found by IDB. The bug requires a thread to be preempted early in the execution and not rescheduled until other threads have completed their tasks. Since Rand is very likely to reschedule the thread, it is not effective at finding this bug. For IDB, only one delay is required, but, as seen in Table 3.3, only one buggy schedule was found; thus, the delay must occur at a specific scheduling point for the bug to manifest.

The CHESS benchmarks test several versions of a work stealing queue. They were used for evaluation in the introduction of preemption bounding [MQ07b] and thus were used to show the effectiveness of preemption bounding as a bug finding technique. DFS fails to find the bug in `chess.WSQ`, while IPB succeeds (as in prior work). The remaining CHESS benchmarks are more complex (lock-free) versions of `chess.WSQ`, which were also used in prior work. IPB and DFS fail to find the bugs in these benchmarks, while IDB is successful (which is relevant to **RQ4**). However, Rand is able to find all the bugs in these benchmarks (like IDB) *and* it also finds them in fewer schedules than IDB and IPB (which is highly relevant to **RQ3**). The prior work that introduced these techniques did not compare against a random scheduler in terms of bug finding ability.

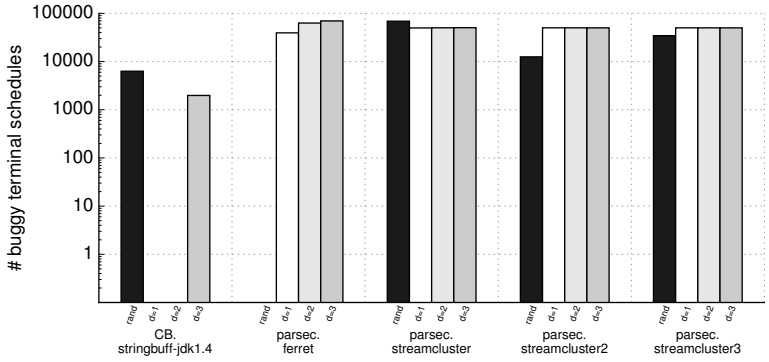


Figure 3.8: Shows, for the `stringbuff-jdk1.4` and `parsec` benchmarks, the number of buggy schedules explored by Rand, and PCT for each value of $d \in \{1, 2, 3\}$. Each technique explored 100,000 schedules.

Effectiveness of Probabilistic Concurrency Testing Figure 3.8 and Figure 3.9 compare the effectiveness of Rand and PCT for each value of $d \in \{1, 2, 3\}$ at finding bugs for a subset of benchmarks; the subset is not representative of all the benchmarks—we focus on benchmarks for which the probabilistic results are notable and worthy of discussion. The bars show the number of buggy terminal schedules exposed by the techniques within 100,000 schedules (except for `chess.IWSQWS` and `chess.SWSQ`, which use a schedule limit of 10,000). The graphs use a log scale for the y-axis. Regarding **RQ2** and **RQ3**, it is interesting to see that Rand is often similar and sometimes better than PCT in terms of number of buggy schedules found. As explained above, we conjecture that in these cases, there are probably many places at which preemptions can occur to allow the bug to manifest and many opportunities for an unexpected operation in a different thread to occur after the preemption. Nevertheless, PCT with d=3 finds all the bugs that Rand finds, plus an additional four, as shown in Figure 3.3b.

Recall that the PCT algorithm inserts $d-1$ priority change points (see §3.2.5). Looking at `CB.stringbuffer-jdk1.4` in Figure 3.8, we can see that this bug was found by both Rand and PCT d=3. Looking at Table 3.3, this benchmark only has 2 threads and around 10 execution steps, but the bug requires at least 2 preemptions or delays to occur. Note that, in the PCT algorithm, a lower priority thread T1 can enable a higher priority thread T2, in which case T2 will preempt T1, without the need for a priority change point. Nevertheless, for this benchmark, it seems likely at least two priority change points are needed for the bug to occur, which would explain why PCT did not find the bug with $d < 3$. Interestingly, Rand is more effective at finding this bug than PCT. The bug requires a preemption away from thread 1 and then a preemption away from thread 2 so

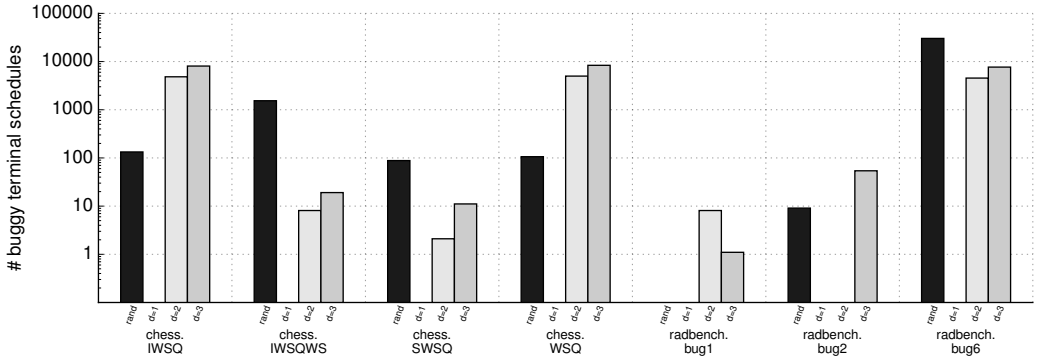


Figure 3.9: Shows, for the `chess` and `radbench` benchmarks, the number of buggy schedules explored by Rand, and PCT for each value of $d \in \{1, 2, 3\}$. Each technique explored 100,000 schedules (except for `chess.IWSQWS` and `chess.SWSQ`, which use a schedule limit of 10,000).

that execution of thread 1 continues. Unfortunately, due to the way in which PCT lowers priorities, the second priority change point may not change the priority ordering between the two threads—it depends on the priorities assigned to the priority change points. For example, assume $d = 3$ and an initial priority mapping of $\{T2 \rightarrow 3, T1 \rightarrow 4\}$, so that T1 has the highest priority. Let the first priority change point change T1's priority to 1, giving a priority mapping of $\{T1 \rightarrow 1, T2 \rightarrow 3\}$ and making T2 the new highest priority thread. Let the second priority change point change T2's priority to 2, giving a priority mapping of $\{T1 \rightarrow 1, T2 \rightarrow 2\}$. The second change point does not change the relative priority ordering between the threads. We speculate that this is the reason why PCT is less effective. This possibly highlights a weakness of the PCT algorithm; on the other hand, PCT was designed carefully to ensure the probabilistic guarantee described in §3.2.5, so "fixing" this issue while maintaining the guarantee may be non-trivial.

Similar observations can be made about the other benchmarks in Figure 3.8 and Figure 3.9 by cross-referencing with Table 3.3; if a bug requires c preemptions or delays, then the bug will usually not be found by PCT with $d-1 < c$ (fewer than c priority change points). We stress that this is not the case in general; an exception is the bug in `chess.WSQ`, which requires 2 preemptions, but was found by PCT d=2 (only 1 priority change point). Thus, this is an example where a lower priority thread unblocks a higher priority thread, resulting in a preemption. We speculate that this is because the benchmark involves blocking locks (the other CHESS benchmarks use spin locks). Similarly, `radbench.bug2` requires 3 preemptions, but was found with PCT d=3 (2 priority change points).

For many of the benchmarks shown in Figure 3.8 and Figure 3.9, increasing d makes

PCT more effective at finding bugs; this suggests that these bugs require certain change points at the right places, but additional change points are unlikely to prevent the bug from occurring. A good example is `parsec.ferret` which, as explained above, requires a thread to be preempted early in the execution and not rescheduled until other threads have completed their tasks. Unlike Rand, PCT is ideally suited to exposing this bug; once the required thread has its priority lowered, it will only be scheduled instead of other enabled threads if all other enabled threads also have their priorities lowered; this benchmark has, on average, 4 enabled threads. Thus, as long as $d < 5$, increasing d simply increases the chance of one of the priority change points occurring at the right place.

The `radbench.bug1` benchmark was found by IDB, PCT d=2 and PCT d=3; very few buggy schedules were found by PCT. The bug requires a thread to be preempted after destroying a hash table and a second thread to access the hash table, causing a crash; this explains why the bug requires only one delay and why PCT was able to find it with at least one priority change point. It is likely that the large number of scheduling points is what pushes this bug out of reach of the other techniques. PCT d=3 found 7 buggy schedules in `radbench.bug2`; the description of this bug is less clear [JPPS11]. This bug and the bug in `CB.stringbuffer-jdk1.4` are the only ones found by PCT that appear to require d=3 (i.e. 2 priority change points).

PCT d=2 and PCT d=3 were the only techniques to find the bugs in `CS.twostage_100_bad`, `CS.reorder_10_bad` and `CS.reorder_20_bad`. However, these benchmarks have identical counterparts with lower thread counts. Recall that, when performing SCT, the thread count should be decreased as much as possible while still capturing an interesting concurrency scenario. Thus, these benchmarks are perhaps not realistic test cases for SCT. Furthermore, IDB found the bugs in the versions of these benchmarks with lower thread counts, plus all the other bugs that were found by PCT d=3. If we ignore these "high thread count" benchmarks, then IDB and PCT found the same number of bugs within the schedule limit; thus, it could be argued that IDB performed similarly to PCT, which is relevant to **RQ1** and **RQ2**. Nevertheless, PCT found these bugs directly, without the thread count having to be reduced, which is an interesting result.

As explained above, the `CS.reorder_X_bad` benchmark (where X is the number of threads launched) are versions of the adversarial delay bounding example given in Figure 3.2 in §3.2.3. One priority change point at the right place (and a particular permutation of initial thread priorities) is sufficient for PCT to expose this bug. Thus, PCT manages to find the bug even when the number of threads is doubled (compare `CS.reorder_10_bad` and `CS.reorder_20_bad` in Table 3.4). This is in contrast to DFS-based techniques, where increasing the thread count increases the number of schedules explored before the first bug,

until the point where the bug is not found within the schedule limit (see Table 3.3).

Recall that, for each value of d that we used with PCT and for each benchmark, we estimated the worst case (smallest) number of buggy schedules that we should find given a bug of depth d, parameters n and k from the benchmark, and our schedule limit of 100,000 (see §3.2.5 and §3.7.3). The estimate for each d is only relevant if the bug associated with a benchmark can in fact manifest with depth d. These estimates can be see in Table 3.4. The minimum value of d for which PCT found a bug provides an *upper bound* on the bug depth; the actual bug depth may be smaller. Assuming that the minimum value of d for which PCT found a bug is, in fact, the depth of the bug, it can be seen that PCT always found many more schedules than the estimated number for that bug. For example, consider `chess.IWSQ` in Table 3.4. It is likely that this bug has depth d=2, since PCT d=1 was not able to find the bug. Assuming this, the estimated worst case number of buggy schedules that we should find (in the PCT d=2 column) is less than 1, yet the actual number of buggy schedules found was 4,829. In fact, for d=2 and d=3 the majority of the benchmarks had a worst case estimate of less than 1 schedule, suggesting that the bugs should not be found within our schedule limit (yet, most bugs were found). Our results agree with the original evaluation of PCT [BKMN10], which showed that the number of buggy schedules found in practice is usually much greater than the smallest number of buggy schedules predicted by the formula.

Comparison with the default Maple algorithm As shown in Figure 3.3d, MapleAlg missed 20 bugs overall, 19 of which were found by other techniques. This includes benchmarks like `CS.bluetooth_driver_bad` and `CS.circular_buffer_bad`, which were quickly found by most other techniques. Maple livelocked on the CHESS benchmarks; this is presumably a bug in the tool that could be fixed. MapleAlg attempts to force certain patterns of inter-thread accesses (or *interleaving idioms*) that might lead to concurrency bugs. It is possible that some of the bugs it misses require interleaving idioms that are not included in MapleAlg.

Small schedule bounds To answer **RQ5**, we note that schedule bounding exposed 45 of the 49 bugs, and 44 of these require a preemption bound of two or less (note that, if a bug can be found with a delay bound of c, then it can also be found with a preemption bound of c, although not necessarily within the schedule limit when using IPB). Furthermore, 42 of these were found using a delay bound of two or less. Thus, a large majority of the bugs in SCTBench can be found with a small preemption or delay bound. This supports previous claims that many bugs can be exposed using a small number of preemptions or

delays [MQ07b, MQB+08, EQR11].

The DFS-based techniques missed the bugs in `CS.reorder_10_bad`, `CS.reorder_20_bad` and `CS.twostage_100_bad`, which, as explained above, are duplicates of other benchmarks but with higher thread counts. The `CS.reorder_X_bad` benchmark is the adversarial delay bounding example given in Figure 3.2 in §3.2.3. Thus, these benchmarks require a delay bound of one less than X (where X is the number of threads). However, it is not clear whether such a scenario is likely to occur in real multithreaded programs.

The bug in `radbench.bug2` requires three preemptions or delays to occur (see Table 3.3). The benchmark is a test case for the SpiderMonkey JavaScript engine in Firefox. A bug requiring three preemptions and delays has been reported before in [EQR11] and this was the first time CHESS had found such a bug. Note that we reduced the number of threads in radbench.bug2 from six to two; thus, IPB and IDB explore exactly the same schedules. Nevertheless, two threads is enough to expose the bug.

The bug in `misc.safestack` was missed by all techniques and reportedly requires five preemptions and three threads. Given this information, we tried running PCT with $d = 6$ for 100,000 executions, but the bug did not occur. We reproduced the bug using Relacy[8], a weak memory data race detector that performs either preemption bounding or controlled random scheduling for C++ programs that use C++ atomics. The bug was found using the random scheduling mode after 75,058 schedules. It is unclear why Maple's random scheduler did not find the bug. It is possible that the number of scheduling points with Maple is higher, as Relacy only inserts scheduling points before atomic operations.

SPLASH-2 benchmarks As explained in §3.4.1, we reduced the input values in the SPLASH-2 benchmarks; this resulted in fewer scheduling points and allowed our data race detector to complete, without exhausting memory. Due to these changes, the results are not directly comparable with other experiments that use the SPLASH-2 benchmarks (unless parameters are similarly reduced). However, the bugs are found by all DFS-based techniques after just two schedules; this would be the same, regardless of parameter values. Therefore, the *# schedules to first bug* data for the DFS-based techniques are comparable to other techniques.

3.8 Main findings

We now summarise the main findings of our study, which relate to the research questions posed in §3.5. The conclusions we draw of course only relate to the 49 benchmarks in

[8] `http://www.1024cores.net/home/relacy-race-detector`

id	name	# max threads (n)	# max enabled threads	# max steps (k)	IPB bound	IPB # schedules to first bug	IPB # schedules	IPB # new schedules	IPB # buggy schedules	IDB bound	IDB # schedules to first bug	IDB # schedules	IDB # new schedules	IDB # buggy schedules	DFS # schedules to first bug	DFS # schedules	DFS # buggy schedules	DFS % buggy
0	CB.aget-bug2	4	3	24	0	1	10	10	4	0	1	1	1	1	1	46486	29513	63%
1	CB.pbzip2-0.9.4	4	4	54	0	2	12	12	4	1	2	31	30	13	2	L	68226	*68%
2	CB.stringbuffer-jdk1.4	2	2	10	2	9	13	8	1	2	9	13	8	1	7	24	1	4%
3	CS.account_bad	4	3	8	0	3	6	6	2	1	3	5	4	1	3	28	4	14%
4	CS.arithmetic_prog_bad	3	2	20	0	1	4	4	4	0	1	1	1	1	1	19680	19680	100%
5	CS.bluetooth_driver_bad	2	2	13	1	6	7	6	1	1	6	7	6	1	36	177	10	5%
6	CS.carter01_bad	5	3	19	1	9	19	16	2	1	8	12	11	1	8	1708	49	2%
7	CS.circular_buffer_bad	3	2	31	1	23	35	32	12	2	25	79	56	36	20	3991	2043	51%
8	CS.deadlock01_bad	3	2	11	1	9	12	9	2	1	7	9	8	1	10	46	3	6%
9	CS.din_phil2_sat	3	2	21	0	1	3	3	3	0	1	1	1	1	1	5336	4686	87%
10	CS.din_phil3_sat	4	3	32	0	1	13	13	13	0	1	1	1	1	1	L	85542	*85%
11	CS.din_phil4_sat	5	4	43	0	1	73	73	73	0	1	1	1	1	1	L	86231	*86%
12	CS.din_phil5_sat	6	5	39	0	1	501	501	501	0	1	1	1	1	1	L	L	*100%
13	CS.din_phil6_sat	7	6	49	0	1	4051	4051	4051	0	1	1	1	1	1	L	L	*100%
14	CS.din_phil7_sat	8	7	59	0	1	7	7	7	0	1	1	1	1	1	924	924	100%
15	CS.fsbench_bad	28	27	155	0	1	1	1	1	0	1	1	1	1	1	L	L	*100%
16	CS.lazy01_bad	4	3	11	0	1	13	13	6	0	1	1	1	1	1	118	81	68%
17	CS.phase01_bad	3	2	11	0	1	2	2	2	0	1	1	1	1	1	17	17	100%
18	CS.queue_bad	3	2	83	1	98	100	97	2	2	63	482	420	326	43	L	59036	*59%
19	CS.reorder_10_bad	11	10	40	0	✗	L	L	0	5	✗	L	38129	0	✗	L	0	*0%
20	CS.reorder_20_bad	21	20	89	0	✗	L	L	0	4	✗	L	21023	0	✗	L	0	*0%
21	CS.reorder_3_bad	4	3	12	1	43	74	61	2	2	25	45	35	3	126	2494	23	<1%
22	CS.reorder_4_bad	5	4	16	1	359	774	701	3	3	205	417	330	7	6409	L	86	*<1%
23	CS.reorder_5_bad	6	5	20	1	3378	8483	7982	4	4	1513	3681	2843	15	✗	L	0	*0%
24	CS.stack_bad	3	2	43	1	23	50	47	9	1	22	32	31	9	22	L	6361	*6%
25	CS.sync01_bad	3	2	9	0	1	2	2	2	0	1	1	1	1	1	6	6	100%
26	CS.sync02_bad	3	2	18	0	1	2	2	2	0	1	1	1	1	1	88	88	100%
27	CS.token_ring_bad	5	4	11	0	8	24	24	4	2	10	29	22	3	8	280	57	20%
28	CS.twostage_100_bad	101	100	792	0	✗	L	L	0	2	✗	L	99304	0	✗	L	0	*0%
29	CS.twostage_bad	3	2	11	1	9	10	7	1	1	7	9	8	1	13	87	3	3%
30	CS.wronglock_3_bad	5	4	25	1	243	856	783	66	1	15	22	21	2	3233	L	3006	*3%
31	CS.wronglock_bad	9	8	49	0	✗	L	L	0	1	31	42	41	2	✗	L	0	*0%
32	chess.IWSQ	3	3	169	1	✗	L	99997	0	2	2990	4378	4264	192	✗	L	0	*0%
33	chess.IWSQWS	3	1	660	1	✗	10000	9997	0	1	219	471	470	1	✗	10000	0	*0%
34	chess.SWSQ	3	1	2406	1	✗	10000	9997	0	1	773	1698	1697	1	✗	10000	0	*0%
35	chess.WSQ	3	3	161	2	2814	8852	8626	640	2	801	2048	1974	192	✗	L	0	*0%
36	inspect.qsort_mt	3	3	81	1	31	88	84	2	1	19	28	27	1	75861	L	2127	*2%
37	misc.ctrace-test	3	2	22	1	4	20	19	12	1	4	20	19	12	4	20	12	60%
38	misc.safestack	4	3	117	1	✗	L	99987	0	3	✗	L	95958	0	✗	L	0	*0%
39	parsec.ferret	11	11	24453	0	✗	L	L	0	1	51	4575	4574	1	✗	L	0	*0%
40	parsec.streamcluster	5	2	1373	1	7951	16072	16066	19	1	1336	1372	1371	10	✗	L	0	*0%
41	parsec.streamcluster2	7	3	4177	0	✗	L	L	0	1	4153	4175	4174	20	✗	L	0	*0%
42	parsec.streamcluster3	5	2	1373	0	2	6	6	4	1	2	1359	1358	4	2	L	60785	*60%
43	radbench.bug1	4	3	21889	1	✗	L	99962	0	1	616	14206	14205	1	✗	L	0	*0%
44	radbench.bug2	2	2	171	3	59354	72704	69895	48	3	59354	72704	69895	48	✗	L	0	*0%
45	radbench.bug6	3	3	101	1	84	168	165	3	1	60	86	85	3	✗	L	0	*0%
46	splash2.barnes	2	2	4449	1	2	4378	4377	326	1	2	4378	4377	326	2	L	23504	*23%
47	splash2.fft	2	2	152	1	2	134	133	61	1	2	134	133	61	2	L	75434	*75%
48	splash2.lu	2	2	140	1	2	105	104	49	1	2	105	104	49	2	L	49887	*49%

Table 3.3: Experimental results for SCT using iterative preemption bounding (IPB), iterative delay bounding (IDB) and unbounded depth-first search (DFS). Entries marked 'L' indicate 100,000, our schedule limit. A '✗' indicates that no bug was found. A percentage prefixed with '*' does not apply to *all* schedules, only those that were explored via DFS before the schedule limit was reached.

id	name	# max threads (n)	# max enabled threads	# max steps (k)	Rand		PCT d=1			PCT d=2			PCT d=3			MapleAlg		
					# schedules to first bug	# buggy schedules	# schedules to first bug	# buggy schedules	est. worst case # buggy	# schedules to first bug	# buggy schedules	est. worst case # buggy	# schedules to first bug	# buggy schedules	est. worst case # buggy	found?	# schedules	total time (seconds)
0	CB.aget-bug2	4	3	24	4	48591	7	25053	43	7	40313	1	5	46938	<1	✓	17	37
1	CB.pbzip2-0.9.4	4	4	54	1	41771	1	16466	8	1	22971	<1	5	27385	<1	✓	4	20
2	CB.stringbuffer-jdk1.4	2	2	10	23	6308	✗	0	500	✗	0	50	1	1979	5	✓	9	7
3	CS.account_bad	4	3	8	8	11912	5	25060	390	5	21936	48	5	19527	6	✓	20	12
4	CS.arithmetic_prog_bad	3	2	20	1	L	1	L	83	1	L	4	1	L	<1	✓	1	1
5	CS.bluetooth_driver_bad	2	2	13	8	6436	✗	0	295	11	3871	22	11	5968	1	✗	11	7
6	CS.carter01_bad	5	3	19	1	46877	✗	0	55	9	16028	2	3	29243	<1	✓	6	5
7	CS.circular_buffer_bad	3	2	31	1	91146	✗	0	34	1	12818	1	1	28763	<1	✗	17	12
8	CS.deadlock01_bad	3	2	11	1	37405	✗	0	275	15	9020	25	15	17234	2	✗	7	5
9	CS.din_phil2_sat	3	2	21	1	96860	1	L	75	2	95337	3	2	93558	<1	✓	1	1
10	CS.din_phil3_sat	4	3	32	1	92850	1	L	24	1	93792	<1	1	90207	<1	✓	1	1
11	CS.din_phil4_sat	5	4	43	1	88754	1	L	10	1	93040	<1	1	88414	<1	✓	1	1
12	CS.din_phil5_sat	6	5	39	1	L	1	L	10	1	L	<1	1	L	<1	✓	1	1
13	CS.din_phil6_sat	7	6	49	1	L	1	L	5	1	L	<1	1	L	<1	✓	1	1
14	CS.din_phil7_sat	8	7	59	1	L	1	L	3	1	L	<1	1	L	<1	✓	1	1
15	CS.fsbench_bad	28	27	155	1	L	1	L	<1	1	L	<1	1	L	<1	✓	1	1
16	CS.lazy01_bad	4	3	11	2	60626	1	49847	206	1	53343	18	1	56197	1	✓	1	1
17	CS.phase01_bad	3	2	11	1	L	1	L	275	1	L	25	1	L	2	✓	1	1
18	CS.queue_bad	3	2	83	1	99986	✗	0	4	38	818	<1	6	14046	<1	✓	2	1
19	CS.reorder_10_bad	11	10	40	✗	0	✗	0	5	439	89	<1	439	135	<1	✗	11	7
20	CS.reorder_20_bad	21	20	89	✗	0	✗	0	<1	219	131	<1	219	224	<1	✗	11	7
21	CS.reorder_3_bad	4	3	12	39	2498	✗	0	173	168	2653	14	115	4559	1	✗	10	7
22	CS.reorder_4_bad	5	4	16	68	726	✗	0	78	86	1257	4	7	2053	<1	✗	11	8
23	CS.reorder_5_bad	6	5	20	68	202	✗	0	41	7	688	2	7	1057	<1	✗	11	7
24	CS.stack_bad	3	2	43	2	60949	✗	0	18	2	27680	<1	2	40159	<1	✗	10	8
25	CS.sync01_bad	3	2	9	1	L	1	L	411	1	L	45	1	L	5	✓	1	1
26	CS.sync02_bad	3	2	18	1	L	1	L	102	1	L	5	1	L	<1	✓	1	1
27	CS.token_ring_bad	5	4	11	9	13004	44	8238	165	11	13655	15	11	16908	1	✓	5	4
28	CS.twostage_100_bad	101	100	792	✗	0	✗	0	<1	10548	5	<1	10548	6	<1	✗	11	9
29	CS.twostage_bad	3	2	11	15	7848	✗	0	275	15	12097	25	15	19840	2	✓	8	5
30	CS.wronglock_3_bad	5	4	25	1	31302	✗	0	32	13	6442	1	4	11107	<1	✓	6	4
31	CS.wronglock_bad	9	8	49	1	32534	✗	0	4	29	3636	<1	24	6693	<1	✓	6	4
32	chess.IWSQ	3	3	169	19	133	✗	0	1	61	4829	<1	24	8069	<1	✗	7	-
33	chess.IWSQWS	3	1	660	3	1538	✗	0	<1	584	8	<1	616	19	<1	✗	9	-
34	chess.SWSQ	3	1	2406	15	88	✗	0	<1	1109	2	<1	612	11	<1	✗	7	-
35	chess.WSQ	3	3	161	392	106	✗	0	1	61	4993	<1	24	8357	<1	✗	12	12
36	inspect.qsort_mt	3	3	81	72	1024	✗	0	5	109	1271	<1	109	2346	<1	✗	142	102
37	misc.ctrace-test	3	2	22	1	24487	2193	7	68	5	27307	3	5	33607	<1	✓	1	1
38	misc.safestack	4	3	117	✗	0	✗	0	1	✗	0	<1	✗	0	<1	✗	23	16
39	parsec.ferret	11	11	24453	✗	0	3	39389	<1	3	63027	<1	3	69745	<1	✓	27	205
40	parsec.streamcluster	5	2	1373	1	68746	1	49831	<1	1	50194	<1	1	50428	<1	✓	1	2
41	parsec.streamcluster2	7	3	4177	21	12514	2	50135	<1	2	50096	<1	2	50075	<1	✗	24	149
42	parsec.streamcluster3	5	2	1373	2	34448	1	50081	<1	1	50081	<1	1	50081	<1	✓	1	1
43	radbench.bug1	4	3	21889	✗	0	✗	0	<1	3084	8	<1	79190	1	<1	✗	583	13811
44	radbench.bug2	2	2	171	27071	9	✗	0	1	✗	0	<1	1813	54	<1	✗	239	950
45	radbench.bug6	3	3	101	1	30211	✗	0	3	15	4543	<1	15	7675	<1	✗	11	10
46	splash2.barnes	2	2	4449	2	49933	2	49967	<1	2	49967	<1	2	49967	<1	✓	1	1
47	splash2.fft	2	2	152	2	62188	2	49967	2	2	50007	<1	2	50017	<1	✓	2	2
48	splash2.lu	2	2	140	1	97329	2	49967	2	2	53574	<1	2	56605	<1	✓	2	3

Table 3.4: Experimental results for randomisation techniques—the controlled random scheduler (Rand) and PCT for each $d \in \{1, 2, 3\}$—and the Maple algorithm (MapleAlg). Entries marked 'L' indicate 100,000, our schedule limit. A '✗' indicates that no bug was found. In the MapleAlg results, '-' indicates that the Maple tool timed out after 24 hours.

SCTBench, but this does include publicly available benchmarks used in prior work to evaluate concurrency testing tools. We refer to the Venn diagrams of Figure 3.3 and the cumulative graphs in Figure 3.4 and 3.5 from §3.7. Recall that these diagrams provide an overview of our results in terms of the bug-finding ability of the techniques we study: iterative preemption bounding (IPB), iterative delay bounding (IDB), depth-first search with no schedule bound (DFS), three parameterised versions of probabilistic concurrency testing (PCT $d = n$, for $n \in \{1, 2, 3\}$) and a controlled random scheduler (Rand). Recall that, for each controlled technique evaluated, a limit of 100,000 schedules per benchmark was used, except for the CHESS benchmarks where (as explained in §3.6) a lower limit was used.

RQ1, RQ2: PCT d=3 performed best. With a limit of 100,000 schedules, PCT d=3 found bugs in 48 of the 49 benchmarks—more than any other technique—including all 45 bugs found by IDB, the next best non-PCT technique in terms of number of bugs found. For lower schedule limits, PCT d=3 still found the most bugs, except for very low schedules limits (<10). This concurs with the findings of prior work, in which PCT found bugs faster than IPB [BKMN10]. However, we note that the three bugs missed by IDB (but found by PCT d=3) are in benchmarks with high thread counts and IDB *was* able to find these bugs within the schedule limit when the thread count was reduced.

RQ3: Controlled random scheduling performed better than IPB and comparably with IDB. Because it is so straightforward, our assumption prior to this study was that use of a controlled random scheduler for bug-finding would not be effective. We initially investigated this method merely because it provides a simple baseline that more sophisticated techniques should surely improve upon (and because this was suggested by a reviewer of the conference version of this work [TDB14]). The effectiveness of controlled random scheduling for bug finding is not addressed in prior work; the papers that introduced preemption bounding [MQ07b] and delay bounding [EQR11] only include DFS or preemption bounding as a baseline for finding bugs (see Footnote 1, p. 29). Our findings, summarised in Figure 3.3c, contradict our assumption: with a schedule limit of 100,000, Rand found 43 bugs, more than IPB (38) and DFS (33), and found all but 2 of the bugs found by IDB (45). Furthermore, as shown in Figure 3.4, the results are similar when lower schedule limits are considered: for schedule limits between 10 and 1000, Rand finds up to 6 more bugs than IDB. We also note that Rand found the bugs in the 4 versions of the CHESS work stealing queue benchmark (ids 32–35) after only a small number of schedules (392 in the worst case), thus performing better than IPB (which missed 3 of the bugs) and IDB. Yet, when IPB was introduced, this benchmark was used to demonstrate/evaluate

its bug finding ability [MQ07b]. This raises two important questions: Does IPB actually aid in bug finding, compared to more naïve approaches? Are the benchmarks used to evaluate concurrency testing tools (captured by SCTBench) representative of real-world concurrency bugs? Our findings indicate that the answer to at least one of these questions must be "no". Nevertheless, schedule bounding still provides simple counterexample traces and bounded coverage guarantees, which is not questioned by our findings.

RQ3, RQ7: Researchers should compare against controlled random scheduling. Much prior work that introduced new techniques did not compare against a controlled random scheduler. Many benchmarks contain defects that can be trivially found using a controlled random scheduler. We stress that future work should use the controlled random scheduler as a baseline, to give an accurate representation of the benchmarks used and the improvement obtained by the new technique.

RQ4: IDB beats IPB. Schedule bounding beats DFS. With a schedule limit of 100,000, IDB found all of the 38 bugs that were found by IPB, plus an additional 7 (see Figure 3.3a). This is in line with experimental claims of prior work [EQR11]. A straightforward DFS with no schedule bounding only exposed bugs in 33 benchmarks, all of which were also found by IPB, as well as by IDB. This also validates prior work [MQ07b, EQR11]. Results were similar in terms of number of bugs found at various lower schedule limits (see Figure 3.4).

RQ5: Many bugs could be found using a small schedule bound. With a schedule limit of 100,000, schedule bounding exposed each bug in 45 of the 49 benchmarks, and 44 of these require a preemption bound of 2 or less. Thus, a large majority of the bugs in SCTBench can be found with a small schedule bound. This supports previous claims that in practice many bugs can be exposed using a small number of preemptions or delays [MQ07b, MQB+08, EQR11]. It also adds weight to the argument that bounded guarantees provided by schedule bounding are useful. However, we note that one bug that was found by schedule bounding requires 3 preemptions and another is reported to require a minimum of 5 preemptions. Also note that certain synthetic benchmarks (such as `reorder_X_bad` and `twostage_X_bad`) are challenging for schedule bounding when the number of threads parameter, X, is increased; as X is incremented, so is the number of delays required for IDB to find the bug. However, it is not clear whether such a scenario is likely to occur in real multithreaded programs.

RQ6: SCT techniques can be difficult to apply. There were 8 distinct programs (providing 26 potential test cases) that could not easily be included in our study, as they use nondeterministic features or additional synchronisation that is not modelled

or controlled appropriately by most tools. This includes network communication, multiple processes, signals (other than pthread condition variables) and event libraries. It is sometimes possible to apply randomisation or heuristic techniques, such as PCT and random scheduling, to such benchmarks. However, this depends on how the techniques are implemented.

Additionally, program modules were often difficult to test in isolation due to direct dependencies on system functions and other program modules. Thus, creating isolated tests suitable for concurrency testing (or even unit testing) may require significant effort, especially for testers who are not familiar with the software under test.

RQ7: Trivial benchmarks. We argue that certain benchmarks used in prior work are "trivial" (based on properties which we discuss in §3.7.4 and summarise in Table 3.2) and cannot meaningfully be used to compare the performance of competing techniques. Instead, they provide a minimum baseline for any respectable concurrency testing technique. For example, the bugs in 18 benchmarks were exposed 50% of the time when using random scheduling; in 8 of these cases, the bugs were exposed 100% of the time.

RQ7: Non-trivial benchmarks. We believe most benchmarks from the CHESS, PARSEC and RADBench suites, as well as the `misc.safestack` benchmark (see §3.4), present a non-trivial challenge for concurrency testing tools. Furthermore, these represent real bugs, not synthetic tests. Future work can use these challenging benchmarks to show the improvement obtained over prior techniques. We also recommend that the research community focus on increasing the corpus of non-trivial concurrency benchmarks that are available for evaluation of analysis tools.

We also summarise several notable findings that do not directly relate to our research questions:

Data races are common. Many (30) of the benchmarks we tested exhibited data races. Although we did not analyse these data races in detail, to the best of our knowledge they are not regarded as bugs by the relevant benchmark developers. Treating data races as errors would hide the more challenging bugs that the benchmarks capture. Future work that uses these benchmarks must take this into account. For the study, we explore the interleavings arising from sequentially consistent outcomes of data races in order to expose assertion failures, deadlocks, crashes or incorrect outputs.

Some bugs may be missed without additional checks. Some concurrency bugs manifest as out-of-bounds memory accesses, which do not always cause a crash. Tools need to check for these, otherwise bugs may be missed or manifest nondeterministically, even when the required thread schedule is executed. Performing such checks reliably and

efficiently is non-trivial.

3.9 Related work

Background on SCT was discussed in §2.1. We now discuss similar prior work and other relevant concurrency testing techniques.

A prior study created a benchmark suite of concurrent programs to evaluate the bug detection capabilities of several tools and techniques [RM09]. Our 49 test programs are drawn from 35 distinct bugs in pthread benchmarks written in C/C++, while the prior study uses 12 distinct bugs in benchmarks written in both Java and C#.[9] Thus, our study is over a larger set of benchmarks, which are mostly distinct from the set used in the prior study. Furthermore, 8 of our benchmarks are derived from open source desktop libraries and applications and a further 7 are from parallel performance benchmark suites (the PARSEC and SPLASH2 benchmarks). The C# benchmarks from the prior study are standalone synthetic test cases. Our study is focused on comparing five SCT techniques (or seven SCT techniques if the different parameter values for PCT are treated as distinct techniques), implemented in the same SCT framework within the same tool, plus the Maple algorithm. This allows us to compare the *techniques* fairly in a single tool (as opposed to comparing several distinct tools that may implement the techniques in different manners), because each technique operates on the same low level implementation, e.g. they use the same notion of scheduling points. In contrast, the prior study tests six techniques implemented over four tools.

We introduced POR [God96] in §2.1. POR reduces the number of schedules that need to be explored soundly (i.e. without missing bugs, assuming the search completes). It relies on the fact that schedules can be represented as a partial-order of operations, where each partial-order reaches the same state. As explained in §3.6, Maple's data race detector is not optimised for use during SCT and the information stored for data race detection is similar to that needed for POR techniques, such as DPOR. Thus, performing POR was deemed infeasible without significant engineering effort. We consider POR techniques in Chapter 4.

The parallel PCT algorithm [NBMM12] improves the PCT algorithm by allowing parallel execution of many threads, as opposed to always serialising execution. This provides increased execution speed but maintains the probabilistic guarantee from PCT. We focus

[9] The companion website for the prior study shows 17 benchmarks that were translated to C#, although only 12 were used in the published study; translation was necessary so that the benchmarks could be used with CHESS.

on SCT techniques where the program is serialised; since we report the number of terminal schedules, increased execution speed does not affect our results.

In addition to the Maple algorithm, there has been a wide-range of work on other non-systematic approaches, including [EFN+02, Sen08, PLZ09]. Like parallel PCT, these approaches are appealing as they allow parallel execution of many threads and can handle complex synchronisation and nondeterminism.

Randomisation has been shown to be effective for search diversification in stateful model checking, where it can be used to allow independent searches to occur in parallel for improved coverage on multicore systems within a predefined time limit [HJG11]. In our study, we use a schedule limit instead of a time limit; it is worth noting that PCT and controlled random scheduling are both trivially parallelisable, and that DFS-based techniques can also be parallelised with additional effort [SBGH12].

We do not consider relaxed memory models in this study; as in prior work [MQB+08, YNPP12], we assume sequential consistency. Finding weak memory bugs would at least require instrumenting memory accesses (similar to performing data race detection during SCT), which would have been far too slow using Maple's built-in support for this. Recent work has shown an efficient approach for testing relaxed memory models with SCT using DPOR [AAA+15, ZKW15].

Our study has briefly touched on dynamic data race detection issues. A discussion of this wide area is out of scope here, but we refer to [FF09] for the state-of-the-art.

3.10 Conclusion

We have presented an independent empirical study on SCT techniques. In future work we believe it would be fruitful to expand SCTBench through the addition of further non-trivial benchmarks to enable larger studies to be conducted. We consider POR techniques in Chapter 4. However, in future reproduction studies, it would be useful to include POR techniques, including techniques such as bounded POR [CMM13].

4 The lazy happens-before relation

Exploring all terminal schedules/states of a program is usually infeasible. In Chapter 3, we evaluated bounding and randomisation SCT techniques that alleviate this by aiming to explore only a subset of schedules to find concurrency bugs quickly. However, these techniques do not avoid redundant schedules that reach already-explored states. In this chapter, we consider *partial-order reduction* (POR), which uses the *happens-before relation* (HBR) to avoid redundant schedules. POR techniques, such as *dynamic partial-order reduction* (DPOR), can be used during SCT to *soundly* skip redundant schedules, guaranteeing that all terminal states will still be explored if the search completes. Thus, such techniques aim to enumerate all terminal states efficiently. Motivated by the fact that POR can be hampered by the use of mutexes, we present a new approach. The main contributions of this chapter are:

- The lazy happens-before relation (lazy HBR) that can provide reduction beyond what is possible with any POR technique for programs that use mutexes. We prove that schedules with identical lazy HBRs are guaranteed to reach identical states.
- Lazy HBR caching—a sound technique that improves upon HBR caching for programs that use mutexes by using the lazy HBR.
- Lazy DPOR—an unsound technique that is inspired by DPOR and uses the lazy HBR to attempt to enumerate terminal states more efficiently than DPOR for programs that use mutexes.
- An evaluation of the lazy HBR and our two techniques using JESS, our new SCT tool for Java programs, over 79 publicly available benchmarks. Our evaluation shows both a large potential and large practical improvement from exploiting the lazy HBR.

Mutex types We only consider (and our techniques only work with) exclusive locking; a thread that locks a mutex owns it exclusively, unlike with reader-writer locks. Additionally, our Java benchmarks only use well-nested, reentrant mutexes, but we do not believe our techniques are limited to these cases.

Relation to published work The lazy HBR was briefly described and evaluated in [TD15].

4.1 Motivation

Partial-order reduction (POR) [God96] can be applied during SCT to reduce the number of schedules explored, without sacrificing the guarantees provided by an exhaustive exploration. POR was originally proposed as a static state-space reduction technique for explicit-state model checkers, such as SPIN [Hol03]; the relevant insight is that a schedule (a total-order of transitions) can be seen as a particular serialisation of the schedule's *happens-before relation* (HBR)—a *partial-order* over the transitions in the schedule. The set of schedules that can be derived from the same HBR are all equivalent—they all reach the same state. Thus, exploring all unique terminal HBRs (HBRs that reach a terminal state) is sufficient to guarantee that all terminal states have been explored and thus that all safety property violations have been detected.

Two methods for applying POR during systematic concurrency testing are *happens-before relation caching* (HBR caching), used e.g. by CHESS [MQB+08], in which exploration of a schedule ceases as soon as the HBR associated with the schedule matches a previously seen HBR, and *dynamic partial-order reduction* (DPOR) [FG05], which detects dependent operations on-the-fly during SCT and subsequently considers alternative orderings to explore distinct HBRs on demand. A potential issue with these methods is that all interleavings of operations on the same mutex must be explored; these operations are totally-ordered in the HBR. This is indeed necessary when operations protected by a common mutex access and update shared data: different interleavings of these operations may lead to different states. However, our hypothesis is that mutexes are often locked *conservatively* so that protected operations actually access *disjoint* data in some scenarios. This can occur, for example, if: threading is used to interleave tasks, without the goal of increased performance; mutexes do not have high contention, and so coarse-grained locking is used for simplicity; it is expensive, complicated, or impossible to check at runtime if locking a mutex is required. A concrete example supporting our hypothesis is a study showing that the Linux kernel has enough data independence between potentially critical sections to benefit from certain "optimistic" concurrency optimisations [PHW07]. Furthermore, the study shows that these critical sections only access independent data *some of the time*; thus, the locks cannot simply be removed.

A conservative locking discipline is arguably easier for programmers to understand than an intricate discipline and thus may be attractive from a software engineering perspective.

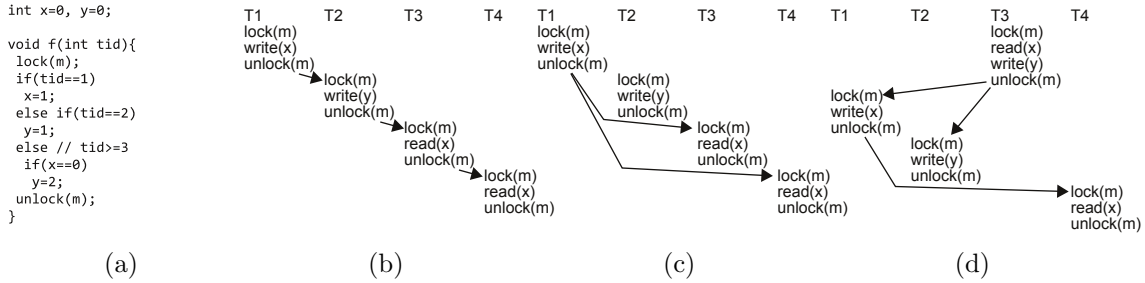


Figure 4.1: A simple multithreaded program (a), and several schedules (b-d). The arrows indicate the HBR; (b) uses the traditional HBR while (c) and (d) use a representation of our *lazy* HBR.

For example, guidance on device driver development recommends that, for simplicity, a driver should enforce mutual exclusion between critical sections using a *single* lock wherever possible [CRKH05]. However, this creates a problem when applying SCT techniques to the resulting software because coarse use of mutexes mitigates the extent to which HBR caching, DPOR, and other POR-based methods, can reduce schedule-explosion.

4.1.1 The lazy HBR: an illustrative example

We illustrate the lazy happens-before relation (lazy HBR) and the benefits it can bring to SCT using the simple multithreaded program of Figure 4.1a (contrived for illustration purposes). The function `f` will be executed in parallel by four threads, although note that the lazy HBR can easily give improved reduction for programs with fewer threads. Each thread will have a distinct integer `tid` parameter drawn from the set $\{1, 2, 3, 4\}$. Figure 4.1 (b), (c) and (d), give three examples of terminal schedules of this program; we show the list of operations performed by each thread from top-to-bottom. We refer to a lock-unlock region (a subsequence of operations by a thread that starts with a lock and ends with the next unlock of the same mutex) as a *critical section* (or just *section*). Since every thread in this program locks and unlocks the same mutex, there are $4! = 24$ schedules (the number of permutations of four sections). We include arrows that indicate a partial-order between the critical sections; the partial-order is indicative of the HBR in (b) and the lazy HBR in (c) and (d), although applying the relation to just the critical sections is a simplification. The relation is transitive but edges that can be obtained via transitivity are omitted from the figure. A pair of schedules are equivalent iff they have the same partial-order; thus schedules that can be obtained by swapping adjacent unordered sections are equivalent.

Using the traditional *happens-before* relation (defined formally in §4.2) the sections in this program are always totally-ordered, illustrated for one terminal schedule in Figure 4.1b.

Each of the 24 possible terminal schedules has a different partial-order, meaning that POR cannot reduce the number of schedules that must be explored. In contrast, Figure 4.1c shows (conceptually) the partial-order obtained from our novel *lazy happens-before relation* applied to the critical sections. The partial-order depends on the variables accessed within the sections: a pair of sections is ordered iff they access a common variable and at least one of the accesses is a write. T1's section writes to `x` and thus is ordered before the sections of T3 and T4, which both read from `x`. This captures the important fact that T3 and T4 read a value that was written by T1. Executing, say, T3's section before T1's would cause a different value of `x` to be read by T3, causing T3 to additionally write to `y`, leading to a different state. This partial-order captures 8 schedules. For example, the order of T3's and T4's sections can be swapped, respecting the partial-order, and the final state of the variables will remain the same. Thus, 7 of these schedules do not need to be explored. Figure 4.1d shows a schedule with a different partial-order according to our lazy happens-before relation that captures 3 schedules. There is an additional arrow because, for this schedule, T3's section writes to `y` as well as reading from `x`. However, T4's section still only reads from `x`. Note that, even though `f` can access `x` and `y` in various ways, the arrows are added dynamically based on the accesses in this particular schedule.

Thus, for this example, there are 24 partial-orders according to the happens-before relation while there are only 11 partial-orders according to our lazy happens-before relation. Any POR algorithm based on the HBR must explore at least 24 schedules even though only 11 need to be explored.

4.2 Background

In this section we refine our concurrent program model and introduce dependencies, events, the HBR, HBR equivalence, and HBR caching.

Concurrent program model We modify our concurrent program model P from §2.2 to separate mutex state from the rest of the shared state. A state is now a triple $s = (ss, mss, tss)$, where ss is the shared state (as before, modulo mutex states), tss is the thread state of every thread (as before), and $mss \in \mathsf{Mutex} \to \mathsf{Tid} \cup \{\bot\}$ is the mutex state of every mutex. As described in §2.3, we let the value of a mutex be $\bot$ iff no thread owns the mutex and thread id $tid \in \mathsf{Tid}$ iff thread tid owns the mutex. We insist that Mutex and Object are disjoint sets. The shared object accessed by a transition can be from either set.

We distinguish *mutex transitions* as being the only transitions that access the mutex

states. Given a transition $t_{tid,ts} \in \mathsf{Transition}$, the transition is a mutex transition iff $obj(t_{tid,ts}) \in \mathsf{Mutex}$. A mutex transition $t_{tid,ts} : (\mathsf{Tid} \cup \{\bot\}) \rightharpoonup (\mathsf{Tid} \cup \{\bot\}) \times \mathsf{ThreadState}$ is a partial function that defines how to update a mutex in Mutex and also yields the next thread state. Otherwise, $t_{tid,ts}$ is a *non-mutex transition* that defines how to update an object in Object and also yields the next thread state, as before. Thus, the transition relation δ is defined by the following two rules:

$$\frac{\begin{array}{c} obj(t_{tid,ts}) \notin \mathsf{Mutex} \\ tss(tid) = ts \qquad o = obj(t_{tid,ts}) \qquad v = ss(o) \qquad t_{tid,ts}(v) \text{ is defined} \\ t_{tid,ts}(v) = (v', ts') \qquad ss' = ss[o \mapsto v'] \qquad tss' = tss[tid \mapsto ts'] \end{array}}{(ss, mss, tss) \xrightarrow{t_{tid,ts}} (ss', mss, tss')} \text{(Non-mutex transition)}$$

$$\frac{\begin{array}{c} obj(t_{tid,ts}) \in \mathsf{Mutex} \\ tss(tid) = ts \qquad m = obj(t_{tid,ts}) \qquad v = mss(m) \qquad t_{tid,ts}(v) \text{ is defined} \\ t_{tid,ts}(v) = (v', ts') \qquad mss' = mss[m \mapsto v'] \qquad tss' = tss[tid \mapsto ts'] \end{array}}{(ss, mss, tss) \xrightarrow{t_{tid,ts}} (ss, mss', tss')} \text{(Mutex transition)}$$

A mutex transition is either a *lock transition* or an *unlock transition*. We define lock and unlock transitions the same way we described lock and unlock operations in §2.3. A lock transition $t_{tid,ts}(v) = (v', ts')$ is defined iff $v = \bot$. If it is defined then $v' = tid$, which corresponds to thread *tid* locking the mutex. Notice that there is only a single thread state that can be reached after executing a lock transition (the transition is only defined for one value of v). For an unlock transition, we assume that thread *tid* must only ever try to unlock a mutex that it owns (i.e. that has state *tid*). Doing otherwise could be treated as an invalid program or could be assumed to be a bug in the program such that further exploration is unnecessary. Thus, we let an unlock transition $t_{tid,ts}(v) = (v', ts')$ be defined iff $v = tid$. If it is defined then $v' = \bot$. As with lock transitions, notice that there is only a single thread state that can be reached after executing an unlock transition.

Dependencies The notion of *dependent* transitions is central to POR methods: adjacent independent transitions in a schedule can be swapped without changing the state reached. We use the definition of a valid dependency relation from [FG05].

Definition 1 (Valid dependency relation)**.** *Let $D \subseteq \mathsf{Transition} \times \mathsf{Transition}$ be a binary, reflexive and symmetric relation. D is a valid dependency relation iff for all pairs of transitions (t_1, t_2), if $(t_1, t_2) \notin D$ (they are independent) then both of the following hold for all states $s \in \mathsf{State}$:*

1. *if* $s \xrightarrow{t_1} s'$*, then* $t_2 \in s$.enabled *iff* $t_2 \in s'$.enabled *(independent transitions neither enable nor disable each other)*

2. *if* t_1 *and* t_2 *are both in* s.enabled*, then there exist states* s', s'', s''' *such that* $s \xrightarrow{t_1} s'' \xrightarrow{t_2} s'$ *and* $s \xrightarrow{t_2} s''' \xrightarrow{t_1} s'$ *(independent events commute)*

A trivial valid dependency relation is one where all transitions are dependent with each other: $D = \mathsf{Transition} \times \mathsf{Transition}$. However, this is extremely conservative and gives no potential for reduction. In practice, it is infeasible to precisely detect *all* independent transitions to get the "perfect" dependency relation (as we do not actually build a model of the state-space). Instead, it is common to define a dependency relation that lies between perfect and trivial using certain properties of transitions. For example, in our JESS tool (used in our experimental evaluation (§4.6) and described further in Chapter 5) we use a common approach that is described explicitly in [CMM13]: a pair of transitions is in D iff either: (a) they are from the same thread, or (b) they access the same shared object and at least one is a write to the shared object. The concept of a shared object was described in §2.2.4 and a transition is typically a write if it changes the value of its accessed shared object. However, the techniques discussed work for *any* valid dependency relation, although we assume a non-conditional dependency relation [God96, KP92] (i.e. it is not influenced by the shared state or mutex state) and that all transitions from a particular thread are dependent with each other, as in [FG05]. We henceforth assume that D is some valid dependency relation. We also assume that transitions that access different shared objects are independent (as this is trivially always the case); in particular, we assume that mutex and non-mutex transitions are independent. Mutex transitions are dependent with each other iff they access the same mutex.

Events When writing an SCT tool, we do not precisely capture shared states or thread states. As such, it is not possible to store a schedule as a list of transitions, as each transition corresponds to a thread state. Instead, we typically store information about each transition in an *event*. Let an event be a tuple $e = (e.tid, e.obj, e.type, e.pti)$. Given a transition $t_{tid,ts}$, we define the event of the transition as:

$$event(t_{tid,ts}) = (tid, obj(t_{tid,ts}), type(t_{tid,ts}), pti(t_{tid,ts}))$$

where: *tid* is the thread of the transition, $obj(t_{tid,ts})$ is the shared object accessed by the transition, and we define $type(t_{tid,ts}) \in \{lock, unlock, other\}$ as the operation type of the transition (*lock* for a lock transition, *unlock* for an unlock transition, and *other* for a non-

mutex transition), and we define $pti(t_{tid,ts}) = i$ as the *per-thread index* of the transition, which indicates that the transition is the ith transition of thread tid.

Thus, given a schedule as a list of transitions $S = \langle t_1, t_2, \ldots, t_k \rangle$, we now represent this as a list of events: $E = \langle e_1, e_2, \ldots, e_k \rangle$, where event $e_i = event(t_i)$. Let Event be the set of all events. Let $TransitionList(E) = S$ yield the list of executed transitions that would be obtained if we executed the transition for each thread $e_1.tid, e_2.tid, \ldots, e_k.tid$ in sequence from the start of the program. Let $state(E) = state(TransitionList(E))$ be the state reached after executing E. We consider the next events and enabled events that correspond to the next transitions and enabled transitions of a state. Let $next(E)$ yield the set of next events from $state(E)$ (one for each next transition, including disabled transitions) and let $enabled(E)$ yield the set of *enabled* events:

$$next(E) = \{event(t) \mid t \in state(E).next\}$$

$$enabled(E) = \{event(t) \mid t \in state(E).enabled\}$$

Thus we think of events being enabled or disabled in a state in the same way as transitions and threads.

Dependent events Two different transitions can yield the same event; for example, a transition $t_{tid,ts}$ that locks a mutex m could yield event $(tid, m, lock, 3)$ but a different transition $t_{tid,ts'}$ (with a different thread state) could yield the same event (although not in the same schedule). We avoid losing potential reduction opportunities (due to being unable to distinguish between distinct transitions) by defining a dependency relation for events *in the context of a schedule* in terms of our valid dependency relation over transitions. We stress that the use of events does *not* lead to a loss of reduction; it is instead the way in which dependencies between events are defined that determines the extent of possible reduction.[1]

Definition 2 (Schedule dependency relation)**.** *Given a schedule as a list of events* $E = \langle e_1, e_2, \ldots, e_k \rangle$*, let* $T = TransitionList(E)$*. Let* $D_E \subseteq \mathsf{Event} \times \mathsf{Event}$ *be a binary, reflexive and symmetric relation defined as follows:*

$$(E(i), E(j)) \in D_E \text{ iff } (T(i), T(j)) \in D$$

[1] In fact, defining the dependency relation over transitions, as we do, is already somewhat limited, as it is not possible for a pair of transitions to only be dependent from certain states. See [AAJS14] for an approach that avoids this limitation.

In other words, we say that $E(i)$ and $E(j)$ are *dependent in* E iff the corresponding transitions are dependent. In JESS, this is approximated by inspecting the events.

The happens-before relation Given a schedule E, the happens-before relation $\rightarrow_E$ of E is a strict (irreflexive, transitive and asymmetric) partial-order over the *set* of events in E (events in a schedule are guaranteed to be unique).

Definition 3 (Happens-before relation $\rightarrow_E$)**.** *We say that* e_i happens-before e_j *in schedule* E*, and write* $e_i \rightarrow_E e_j$ *iff* e_i *occurs before* e_j *in* E *and one of the following holds:*

1. e_i *and* e_j *are dependent in* E*, or*
2. *there exists an event* e *such that* $e_i \rightarrow_E e$ *and* $e \rightarrow_E e_j$ *(the relation is transitive).*

The relation $\rightarrow_E \subseteq \mathsf{Event} \times \mathsf{Event}$ *is called the* happens-before relation *for* E*. Observe that* $\rightarrow_E$ *is the transitive closure of the dependency relation of* E *(*D_E*) intersected with the total-order* E*.*

HBR equivalence A schedule E is one particular linearisation of the partial-order $\rightarrow_E$, and all other linearisations reach the same state. This is a well-known result, but we present its proof since our development of the lazy HBR in §4.3 requires an analogous proof.

Theorem 4.2.1 (HBR equivalence)**.** *Let* E_1 *and* E_2 *be schedules with identical HBRs. That is,* $\rightarrow_{E_1} = \rightarrow_{E_2}$*. Then* E_1 *and* E_2 *lead to the same state:* $\mathit{state}(E_1) = \mathit{state}(E_2)$*.*

Proof. Because E_1 and E_2 have identical HBRs, it must be possible to transform E_1 into E_2 by repeatedly swapping adjacent, unordered (and therefore *independent*) events. Thus, it remains to show that swapping adjacent, independent events in a schedule preserves the state that is reached.

Let $E = v \cdot \langle e_i, e_j \rangle \cdot w$ be a schedule, where e_i and e_j are independent events in E. Consider $\mathit{state}(v)$. We know that $e_i \in \mathit{enabled}(v)$. By rule 1 of a valid dependency relation (Definition 1), and the fact that e_i and e_j are independent, we also have $e_j \in \mathit{enabled}(v)$. By rule 2 of a valid dependency relation the schedule $v \cdot \langle e_j, e_i \rangle$ (with the events swapped) must exist, and we must have $\mathit{state}(v \cdot \langle e_i, e_j \rangle) = \mathit{state}(v \cdot \langle e_j, e_i \rangle)$. Thus, $\mathit{state}(v \cdot \langle e_i, e_j \rangle \cdot w) = \mathit{state}(v \cdot \langle e_j, e_i \rangle \cdot w)$.

□

Algorithm 4 HBR caching.

```
1: procedure EXPLORE(E)
2:     H = H ∪ {→_E}
3:     for each e ∈ enabled(E)
4:         if →_{E·⟨e⟩} ∉ H
5:             Explore(E · ⟨e⟩)
6: end procedure
```

HBR caching Exhaustive SCT may explore many distinct schedules that reach the same state. This problem can be reduced by a simple form of POR called *HBR caching* [MQ07a][2] whereby the HBR of every (not necessarily terminal) schedule explored is stored in a set. If the HBR of the current schedule is in the set already then we have already explored all terminal states reachable from this state and so exploration from the current state ceases and the search backtracks.

We show HBR caching as a recursive procedure in Algorithm 4. The global set H stores previously explored HBRs; the recursive call is only made for successor HBRs that are not in H. Recall that returning from a recursive call to EXPLORE corresponds to backtracking which, in practice, requires restarting the target program from the start. Notice that we can check whether each successor HBR is in H without having to make the recursive call. Thus, the search actively *avoids* already-explored HBRs that are one step away which avoids backtracking (i.e. restarting the target program) if there are other successor HBRs that have not been explored. Note that in practice a set of HBR *hashes* are stored using an incremental hash function (see §4.5).

One of the benefits of HBR caching is its simplicity and the fact that it can be soundly combined with preemption bounding in a simple manner [MQ07b, MQ07a]; combining the more sophisticated dynamic partial-order reduction method with preemption bounding poses subtle challenges [CMM13].

4.3 The lazy HBR

In this section, we introduce our first main contribution of this chapter—the lazy HBR. The intuition is that, in many cases, the order in which lock and unlock events are executed does not affect the state that is reached. Our *lazy* HBR treats lock and unlock events as independent, which can allow a pair of distinct HBRs to be detected as reaching equivalent states. This reduction goes beyond what is possible with any POR technique because the

[2] In prior work the term happens-before *graph* caching is used [MQ07a].

dependency relation that we use is *not* valid. Despite this, we show that the lazy HBR is still useful, as it yields a more accurate representation of a state (and thus closer estimates for the number of states explored during SCT), and more efficient HBR caching.

Lazy happens-before We define the *lazy* dependency relation D', based on D, in which lock and unlock transitions from different threads are regarded as being *independent*:

Definition 4 (Lazy dependency relation D')**.** *A pair of transitions* $(t_1, t_2) \notin D'$ *(they are* lazy independent*) iff:*

1. $(t_1, t_2) \notin D$*, or*
2. $obj(t_1) \in \mathsf{Mutex}$ *and* $obj(t_2) \in \mathsf{Mutex}$ *and* $thread(t_1) \neq thread(t_2)$*.*

In other words, a pair of mutex transition from different threads are always lazy independent. For non-mutex transitions, the valid dependency relation is used. Recall that we already assume that a mutex transition will never be dependent with a non-mutex transition.

Given a schedule E, we define the *lazy* schedule dependency relation D'_E using D', and we define the *lazy* happens-before relation $\rightarrow'_E$ using D'_E.

Definition 5 (Lazy schedule dependency relation)**.** *Given a schedule* E*, the lazy schedule dependency relation* D'_E *is defined identically to* D_E *(Definition 2) except that* D' *is used instead of* D*.*

Definition 6 (Lazy happens-before relation $\rightarrow'_E$)**.** *Given a schedule* E*, the* lazy happens-before relation $\rightarrow'_E$ *is defined identically to* $\rightarrow_E$ *(Definition 3) except that the* lazy *schedule dependency relation* D'_E *is used instead of* D_E*.*

Notice that swapping adjacent lazy independent events in a schedule may now lead to an *invalid* schedule—a schedule that cannot actually be executed on our concurrent program P because one of the events in the schedule will be disabled. For example, consider a schedule $E = w \cdot \langle u, l \rangle \cdot v$, where u is an unlock event that unlocks mutex m and l is a lock event from a different thread that locks m. Clearly, event u enables event l and so these are dependent. However, they are lazy independent. Swapping these events yields an invalid schedule $E' = w \cdot \langle l, u \rangle \cdot v$ because $l \notin enabled(w)$.

Thus, D' is an *invalid* dependency relation (i.e. it is not a valid dependency relation as defined by Definition 1). Regardless, we prove the analogue of Theorem 4.2.1 for the lazy happens-before relation, showing that two *valid* schedules with the same lazy HBR reach the same state. We assume schedules are valid unless otherwise stated.

Theorem 4.3.1 (Lazy happens-before equivalence)**.** *Let E_1 and E_2 be schedules with identical lazy HBRs. That is, $\rightarrow'_{E_1} = \rightarrow'_{E_2}$. Then E_1 and E_2 lead to the same state: $state(E_1) = state(E_2)$.*

The challenge that prevents a proof of Theorem 4.3.1 analogous to the proof of Theorem 4.2.1 is that D' is not *valid* for P and so we cannot just consider a pair of adjacent events. Our proof strategy is to consider an alternative transition system P' that provides a more relaxed semantics than P. We will then relate P and P' in order to get the desired result. Thus, assume that P' is identical to P except that lock and unlock transitions are assumed to always be enabled and do not change the mutex state (they are no-ops). Given a transition t, let $\pi_{ts}(t) = \{t(v).ts \mid v \in dom(t)\}$ yield the set of all thread states that can be reached via the transition; if t is a mutex transition then we know that there is always precisely one thread state that can be reached and so $\pi_{ts}(t)$ will result in a singleton thread state. Thus, the updated inference rule for mutex transitions in P' is:

$$\frac{\begin{array}{c} obj(t_{tid,ts}) \in \mathsf{Mutex} \\ tss(tid) = ts \\ \pi_{ts}(t_{tid,ts}) = \{ts'\} \qquad tss' = tss[tid \mapsto ts'] \end{array}}{(ss, mss, tss) \xrightarrow{t_{tid,ts}} (ss, mss, tss')} \text{(Mutex transition in P')}$$

Thus, we can define P' using the same set of transitions ($\mathsf{Transition}$) as in P.

Observe that the lazy dependency relation D' *is* a valid dependency relation for P' as lock and unlock transitions neither enable nor disable each other and, since they no longer mutate the mutex state, lock and unlock transitions commute.

To distinguish between states of P and P' we use $state'(E)$ to denote the state reached by a schedule E on P'. For a state s of P or P', let $\pi_\mu(s) = s.mss$ and $\pi_{(\sigma,\tau)}(s) = (s.ss, s.\tau)$ project the mutex and non-mutex components of s, respectively.

Our proof of Theorem 4.3.1 uses the following lemmas:

Lemma 4.3.2. *If E_1 and E_2 are schedules of P such that $\rightarrow'_{E_1} = \rightarrow'_{E_2}$ then $\pi_\mu(state(E_1)) = \pi_\mu(state(E_2)))$.*

Proof. The proof is by a straightforward counting argument on lock and unlock events. Given $\rightarrow'_{E_1} = \rightarrow'_{E_2}$, we know that E_1 and E_2 contain the same events. Thus, for each mutex m, E_1 and E_2 contain the same lock and unlock events that access m. Assuming x lock events for mutex m, there must be x or $x-1$ unlock events for mutex m. Case x unlock events: mutex m has been locked and unlocked x times and so is mapped to $\perp$ in both $\pi_\mu(state(E_1))$ and $\pi_\mu(state(E_2))$. Case $x-1$ unlock events: the extra lock event must

be from the same thread, *tid*, in both E_1 and E_2 because they contain the same events. Thus, mutex m has been locked and unlocked $x-1$ times and then locked one more time by thread *tid* and so is mapped to *tid* in both $\pi_\mu(state(E_1))$ and $\pi_\mu(state(E_2))$. □

Lemma 4.3.3. *Let E be a schedule of P. It must the case that $\pi_{(\sigma,\tau)}(state(E)) = \pi_{(\sigma,\tau)}(state'(E))$.*

Proof. The proof is by structural induction on E. Base case: $\pi_{(\sigma,\tau)}(state(\langle\rangle)) = \pi_{(\sigma,\tau)}(state'(\langle\rangle))$ trivially holds. Inductive step: Assuming $\pi_{(\sigma,\tau)}(state(E')) = \pi_{(\sigma,\tau)}(state'(E'))$ we show that $\pi_{(\sigma,\tau)}(state(E' \cdot \langle e\rangle)) = \pi_{(\sigma,\tau)}(state'(E' \cdot \langle e\rangle))$, by showing that t, the corresponding transition of e for both P and P', updates the thread states and shared state identically in both P and P'. If t is a non-mutex transition, then note that $state(E').ss(obj(t)) = state'(E').ss(obj(t))$ due to our inductive hypothesis. Thus, by considering the inference rule for non-mutex transitions (which is the same for both P and P' and given in §4.2), we can see that t updates the thread states and shared state identically in both P and P'. If t is a mutex transition, then t does not update the shared state and updates the thread state of $thread(t)$ to only one possible thread state (regardless of the mutex states reached after E'). Thus, t updates the thread states and shared state identically in both P and P'. □

Lemma 4.3.4. *Let E_1 and E_2 be schedules of P' with identical lazy HBRs. That is, $\rightarrow'_{E_1} = \rightarrow'_{E_2}$. Then E_1 and E_2 lead to the same state in P': $state'(E_1) = state'(E_2)$.*

Proof. The proof is the same as that of Theorem 4.2.1 because D' is a valid dependency relation for P'. □

We are now in a position to prove Theorem 4.3.1.

Proof of Theorem 4.3.1. Let E_1 and E_2 be schedules of P such that $\rightarrow'_{E_1} = \rightarrow'_{E_2}$. We must show that $state(E_1) = state(E_2)$. From Lemma 4.3.2 we have $\pi_\mu(state(E_1)) = \pi_\mu(state(E_2))$. It remains to show that $\pi_{(\sigma,\tau)}(state(E_1)) = \pi_{(\sigma,\tau)}(state(E_2))$. We have:

$$\begin{aligned}
\pi_{(\sigma,\tau)}(state(E_1)) &= \pi_{(\sigma,\tau)}(state'(E_1)) && \text{(Lemma 4.3.3)}\\
&= \pi_{(\sigma,\tau)}(state'(E_2)) && \text{(Lemma 4.3.4)}\\
&= \pi_{(\sigma,\tau)}(state(E_2)) && \text{(Lemma 4.3.3)}
\end{aligned}$$

□

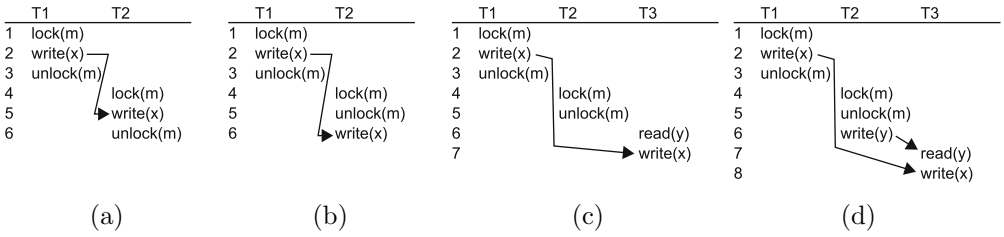
Lazy HBR caching An immediate consequence of Theorem 4.3.1 is that the HBR caching technique discussed in §4.2 can be optimised to use the lazy HBR. We evaluate the effectiveness of lazy HBR caching in §4.6.

Mutex-deadlock states We call a terminal state where at least one thread is blocked at a lock operation a *mutex-deadlock state*; we describe an example later in §4.4.3. Note that P' does not contain any mutex-deadlock states, even if P does, because mutex operations do not block in P'. Early in our investigation of the lazy HBR, we believed that this may lead to cases where we cannot detect mutex-deadlock states in P if using the lazy HBR. However, this issue does not apply when comparing lazy HBRs (like in lazy HBR caching). In §4.4.3, we describe how this affects our lazy DPOR algorithm and how we handle this.

4.4 Lazy DPOR

We present our second main contribution of this chapter—the lazy dynamic partial-order reduction (lazy DPOR) algorithm, which exploits the lazy HBR to explore unique states more efficiently (in fewer schedules) than DPOR. Unlike DPOR, lazy DPOR is unsound but instead provides efficient terminal state coverage beyond that which is possible when using POR and, we believe, is unlikely to miss terminal states in most cases; we consider several counter-examples. We recap the original DPOR algorithm (§4.4.1), motivate the design of lazy DPOR using examples (§4.4.2), and then describe lazy DPOR in full (§4.4.3).

Lazy DPOR unsoundness We originally intended lazy DPOR to be a sound technique except for programs that contain mutex-deadlock states. However, we did not achieve this and so leave a sound extension (modulo mutex-deadlock) for future work. We believe lazy DPOR is likely to explore all terminal states in many cases (except for programs that contain mutex-deadlock states) and is still useful when thorough (but not necessarily complete) terminal state coverage is desired, particularly in cases where enumerating all terminal HBRs is infeasible (and so a sound, POR-based analysis is infeasible anyway). Interestingly, we note in our experiments (§4.6.3) that lazy DPOR never missed a terminal state that was explored via DPOR (except for the benchmarks where we detected potential mutex-deadlock and for one other easily-detectable case, which we describe in §4.4.3); thus, terminal state coverage was always greater than or equal to DPOR for our benchmarks and schedule limit (except for detected incompatible benchmarks). We also note in §4.6.3 that lazy DPOR was able to explore more terminal states in less or equal time compared to DPOR.



(a) (b) (c) (d)

Figure 4.2: Four (unrelated) terminal schedules that we considered when designing lazy DPOR. The arrows indicate the lazy races.

4.4.1 Dynamic partial-order reduction (DPOR)

DPOR [FG05] attempts to avoid exploring equivalent terminal schedules using the HBR. Intuitively, DPOR attempts to reverse pairs of *dependent* events in order to explore *different* terminal happens-before relations (and thus different terminal states). Given a schedule E, we say that a pair of events from E *race* in E iff they are from different threads and are *directly* related by the HBR:

Definition 7. *A pair of events* (e, e') *is a* race *in schedule* E *iff* $e \rightarrow_E e'$ *and* $e.tid \neq e'.tid$ *and there does not exist* e'' *in* E *such that* $e \rightarrow_E e''$ *and* $e'' \rightarrow_E e'$.

Intuitively, given a race in a schedule, we can always swap adjacent, independent events such that the racing events are adjacent. We can then try to swap the order of the racing events to get to a different HBR and a different terminal state.

Algorithm 5 presents the DPOR algorithm, omitting some optimisations that are part of the original [FG05] to simplify our presentation. The EXPLORE procedure recursively explores the state-space in a depth-first manner, backtracking at certain points to explore alternative schedules. When EXPLORE(E) returns, all terminal states reachable from $state(E)$ are guaranteed to have been explored. EXPLORE is initially called on the empty schedule $\langle\rangle$. Thus, when EXPLORE($\langle\rangle$) returns, all terminal states will have been explored. If no error states were encountered then the program is verified to be free from safety property violations.

A set $backtrack(E)$ is associated with each non-terminal schedule E that is explored. In straightforward SCT without DPOR (§3.2.1), at each schedule E reached, every event $e \in enabled(E)$ is "explored" by calling EXECUTE recursively for each schedule $E \cdot \langle e \rangle$. In the DPOR algorithm, only the events in $backtrack(E)$ are explored. At line 4, $backtrack(E)$ is initialised to contain a single arbitrary event from $enabled(E)$ to force exploration towards a terminal state. Then, at line 8, each event in $backtrack(E)$ is explored via recursive calls. Within a recursive call to EXPLORE($E \cdot w$), for some list of events w, events may

Algorithm 5 DPOR algorithm.

```
1:  procedure EXPLORE(E)
2:    AddBacktrack(E)
3:    if ∃e ∈ state(E).enabled
4:      backtrack(E) = {e}
5:      let done = ∅
6:      while ∃e′ ∈ (backtrack(E) \ done)
7:        add e′ to done
8:        Explore(E · ⟨e′⟩)
9:  end procedure
10: procedure ADDBACKTRACK(E)
11:   for all e_n ∈ state(E).next
12:     for all i ∈ dom(E) s.t. (E(i), e_n) races in E · ⟨e_n⟩
13:       let E′ = E[1 : i − 1]
14:       if ∃e_b ∈ state(E′).enabled s.t. e_b.tid = e_n.tid
15:         add e_b to backtrack(E′)
16:       else
17:         add all of state(E′).enabled to backtrack(E′)
18: end procedure
```

be added to *backtrack*(E) via the ADDBACKTRACK procedure which is called at line 2. ADDBACKTRACK considers each event e_n in *next*(E) and finds every index i in *dom*(E) such that $E(i)$ races with e_n in the schedule $E \cdot \langle e_n \rangle$. (Strictly, $E \cdot \langle e_n \rangle$ may not be a valid schedule because e_n may not be in *enabled*(E); the definition of a race (and therefore, the happens-before relation, dependency relation, and *TransitionList*) is applied to the list $E \cdot \langle e_n \rangle$ in the obvious manner.) When such a race is found, E' is defined (line 13) such that *state*(E') is the state from which $E(i)$, the first event participating in the race, was executed. One or more events are then added to *backtrack*(E') to attempt to *reverse* the race by forcing an alternative schedule in which $E(i)$ occurs after e_n. There are several different ways in which to add to the backtrack set; in both this paper and our tool, we use the simpler variant from the original DPOR paper [FG05]. The algorithm attempts to add an event from *enabled*(E') that has the same *tid* as e_n. If such an event is not enabled, then the algorithm conservatively adds *all* events from *enabled*(E') to *backtrack*(E).

4.4.2 From DPOR to lazy DPOR

Our aim is to optimise DPOR to use the lazy HBR. However, simply modifying DPOR to use lazy races (by interpreting Definition 7 w.r.t. the lazy HBR) does not work. Sup-

pose the schedule E of Figure 4.2a is the first terminal schedule to be explored. We use `write(x)` and `read(x)` to indicate a write and read access, respectively, of the shared variable (shared object) x. The arrows indicate lazy races. In EXPLORE($E[1:4]$), the lazy race between event $E(2)$ and the next event of T2 will be detected, so DPOR will add events to $\mathit{backtrack}(E[1:1])$) to try to reverse the events involved. However, only T1 is enabled at $\mathit{state}(E[1:1])$) and its next event is already in the backtrack set. DPOR will terminate without exploring the schedule in which T2 is the first to write to x. It is not possible to reverse the events from $\mathit{state}(E[1:1])$) because T1 owns m and T2 cannot execute until m is unlocked; to unlock m, T1 must execute, but T1's next event is the write to x.

A possible solution would be to consider the set of mutexes held by T1 at $\mathit{state}(E[1:1])$) and the set of mutexes held by T2 at $\mathit{state}(E[1:4])$); if the intersection of these sets is non-empty we will not be able to reverse the lazy race. We could consider "moving the backtrack point" back one event at a time until the intersection is empty. Thus, we would add to $\mathit{backtrack}(\langle\rangle)$ and the lazy race would be reversed. However, now assume that E is the schedule given by Figure 4.2b. Adding to $\mathit{backtrack}(E[1:1])$ would not work (as before), but the intersection of the set of mutexes held by T1 at $\mathit{state}(E[1:1])$) and the set held by T2 at $\mathit{state}(E[1:5])$) is empty, and so our proposed solution would fail. We could try to fix this by considering the set of mutexes held by T1 at $\mathit{state}(E[1:1])$) (as before) and the set of mutexes locked (even if they are subsequently unlocked) within the event subsequence $E[2:5]$ (i.e. from the first event of the lazy race up to the second); the intersection of these sets is $\{m\}$ and we so could now move the backtrack point back to a point where T1 does not own m. Thus, we would add to $\mathit{backtrack}(\langle\rangle)$ as before. However, consider if E was given by Figure 4.2c. Now it *is* possible to reverse the lazy race by adding to $\mathit{backtrack}(E[1:1])$ but our proposed solution would force us to backtrack at $\mathit{backtrack}(\langle\rangle)$. Thus, it may seem like we only need to consider mutexes locked by T2. Now assume E is given by Figure 4.2d. This is a similar schedule except we assume that T3's write to x only occurs if the write to y happens-before the read of y. Thus, we must add to $\mathit{backtrack}(\langle\rangle)$ in order to reverse the race on the writes to x. This schedule illustrates our final solution which is based on the idea that we must consider all mutexes locked that *lazy happen-before* the second event in the race.

Let $\mathit{Held}(E \cdot \langle e \rangle)$, denote the set of mutexes owned by thread $e.\mathit{tid}$ in the state reached after executing E:

$$\mathit{Held}(E \cdot \langle e \rangle) = \{m \in \mathsf{Mutex} \mid \mathit{state}(E).\mathit{mss}(m) = e.\mathit{tid}\}$$

and $\mathit{Held}(\langle\rangle) = \emptyset$. Now consider a schedule $E \cdot w$ and an event e that is a next event for the state reached, i.e. $e \in \mathit{next}(E \cdot w)$ but e might not be in $\mathit{enabled}(E \cdot w)$. Let $\mathit{Relevant}(E, w, e)$ yield the set of mutexes locked by events in w where the lock events must lazy happen-before e in $E \cdot w \cdot \langle e \rangle$:

$$\begin{aligned}\mathit{Relevant}(E, w, e) = \{m \in \mathsf{Mutex} \mid \exists i \in \mathit{dom}(E \cdot w) \wedge \\ i > |E| \wedge \\ (E \cdot w)(i).\mathit{type} = \mathit{lock} \wedge \\ (E \cdot w)(i).\mathit{obj} = m \wedge \\ (E \cdot w)(i) \rightarrow'_{E \cdot w \cdot \langle e \rangle} e\}.\end{aligned}$$

Thus, our solution to reversing a lazy race $(E(i), E(j))$ is to identify the largest $i' \leq i$ for which:

- $E(i).\mathit{tid} = E(i').\mathit{tid}$ and
- $\mathit{Held}(E[1 : i']) \cap \mathit{Relevant}(E[1 : i'], E[i' + 1 : j - 1], E(j)) = \emptyset$,

and add to $\mathit{backtrack}(E[1 : i' - 1])$. For example, assume E is given by Figure 4.2d once again. In Explore($E[1 : 7]$), the lazy race between $E(2)$ and the next event of T3 $E(8)$ is detected, so let $i = 2$ and $j = 8$. With $i' = 2$, the above intersection is:

$$\begin{aligned} & \mathit{Held}(E[1:2]) \cap && \mathit{Relevant}(E[1:2], E[3:7], E(8)) \\ = \; & \{m\} \cap && \{m\} \\ = \; & \{m\} \end{aligned}$$

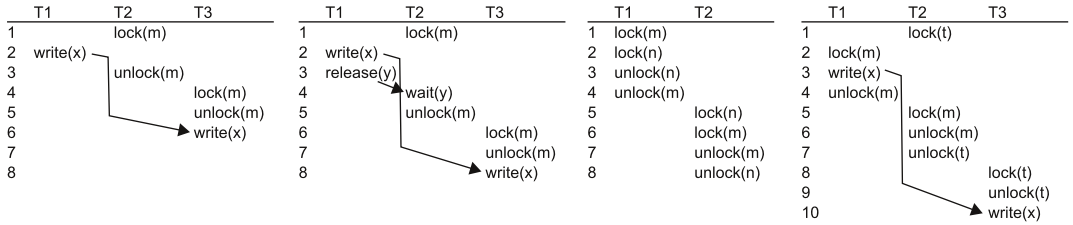
Thus, we must let $i' = 1$ to give $\mathit{Held}(E[1 : 0]) = \emptyset$. Thus, we add an event for T3 to $\mathit{backtrack}(\langle\rangle)$ which will eventually lead to the race being reversed.

4.4.3 Lazy DPOR algorithm

Algorithm 6 shows our lazy DPOR algorithm. The Explore procedure is not shown; it is the same as in DPOR (Algorithm 5), except that the call to AddBacktrack is replaced with a call to AddBacktrack'. Referring to Algorithm 6, line 2 considers each next event e_n of $\mathit{state}(E)$. At line 3, each index i of E is considered such that $E(i)$ races with e_n according to the lazy happens-before relation. Line 4 finds the set I of suitable values for i', as described above; line 5 chooses the maximum value in I. The rest of the procedure is the same as in DPOR.

Algorithm 6 Lazy DPOR algorithm.

```
1: procedure AddBacktrack'(E)
2:   for all e_n ∈ state(E).next
3:     for all i ∈ dom(E) s.t. (E(i), e_n) lazy races in E · ⟨e_n⟩
4:       let I = {i' ∈ dom(E) | i' < i ∧ E(i').tid = E(i).tid ∧
                   Held(E[1 : i']) ∩ Relevant(E[1 : i'], E[i' + 1 : |E|], e_n) = ∅}
5:       let i' = max(I)
6:       let E' = E[1 : (i' − 1)]
7:       if ∃e_b ∈ state(E').enabled s.t. e_b.tid = e_n.tid
8:         add e_b to backtrack(E')
9:       else
10:        add all of state(E').enabled to backtrack(E')
11: end procedure
```



(a) (b) (c) (d)

Figure 4.3: Several terminal schedules that demonstrate potential issues for lazy DPOR. The arrows indicate the lazy races.

As discussed earlier, the lazy DPOR algorithm may miss terminal states, in particular mutex-deadlock states. We now consider several issues with the lazy DPOR algorithm that lead to unsoundness.

Issue 1 (Blocking operations inside critical sections)**.** *Consider a schedule E given by Figure 4.3a. The lazy race can be reversed by adding to backtrack$(E[1:1])$. Now let E be given by Figure 4.3b. Assume* `wait` *is some operation that blocks unless* `release` *is executed. Lazy DPOR will try to reverse the lazy race on x by adding to backtrack$(E[1:1])$ but this will not work because T2 cannot unlock m from this state without first executing T1's* `release`*. Instead of trying to handle this, we simply detect operations that block inside critical section; if we encounter such an operation we issue a warning so that the user can either accept the unsoundness or switch to using regular DPOR. We note that blocking while owning a mutex is typically avoided by programmers in practice as it can lead to deadlock. In particular, note that a* `wait` *operation as found in Java programs unlocks an associated mutex before blocking. Thus, a thread will only block inside a critical section*

in this case if it also owns additional mutexes.

Issue 2 (Mutex-deadlock states)**.** *Consider Figure 4.3c. This terminal schedule has no lazy races and so lazy DPOR will not explore any further schedules. However, deadlock can occur if T1 locks m and then T2 locks n; at this point both threads then block forever on their next lock operations. At noted earlier, we call a terminal state where at least one thread is blocked at a lock operation a* mutex-deadlock state*. Lazy DPOR is very likely to miss mutex-deadlock states. Furthermore, if lazy DPOR encounters a mutex-deadlock state it may miss further terminal states as it will not try to reverse the lock operations.*

We address the issue by conservatively detecting whether mutex-deadlock states are likely to exist and, thus, warn the user if terminal states are likely to be missed because of this. We construct a wait-for graph*, a labelled directed graph, for each terminal schedule explored. The graph is constructed according to insights from prior work [BH05]—we use one of the most straightforward approaches, which we found to be effective for our benchmarks (§4.6). The graph is initially empty for each schedule. A mutex m is added to the graph when it is acquired by a thread i; for each mutex n that is already held by thread i, an edge is added from n to m with label i. We define a* may-deadlock *cycle in this graph as a path of the form* $m_1 \xrightarrow{t_1} m_2 \xrightarrow{t_2} \ldots m_k \xrightarrow{t_k} m_1$*, where each thread label* t_j *for* $j \in 1, \ldots, k$ *is unique. If the wait-for graph of every terminal schedule of the program contains no may-deadlock cycles, then mutex-deadlock cannot occur (although lazy DPOR does not guarantee that all schedules will be explored). If a may-deadlock cycle is found, we warn the user.*

Issue 3 (Mutexes owned by other threads)**.** *Consider a schedule E given by Figure 4.3d. Lazy DPOR will attempt to reverse the lazy race by adding to* $backtrack(E[1:2])$*. However, it is not possible to execute T3's write to x before T1's from this state because T2 cannot unlock t unless T1 executes and unlocks m, and T3 cannot execute until t is unlocked. Thus, it seems that we would need to consider the regular HBR in this case. We currently do not try to handle this issue and instead accept that lazy DPOR may miss terminal states in certain scenarios. However, we claim that lazy DPOR is still a very efficient approach for enumerating* most *terminal states and we surmise that it will often explore all terminal states in practice. In our experimental evaluation, we observe that lazy DPOR did not miss any terminal states that were explored by DPOR on our benchmarks (except when Issue 1 or Issue 2 was detected).*

As discussed earlier, a refinement of the lazy DPOR algorithm may be able to ensure soundness given the assumption that there are no mutex-deadlock states but this will require further investigation.

4.5 JESS: an SCT tool for Java programs

We have implemented an SCT tool for Java programs called JESS which includes lazy and regular versions of HBR caching and DPOR. JESS is written in Java and uses Java bytecode instrumentation (targeting Java 1.7) to control the thread schedule at each visible operation. JESS performs a left-recursive depth-first search of the schedule-space; at each "node" in the tree, the enabled operations (the branches) are in thread creation order, starting (on the left) with the thread that executed most recently, wrapping in a round-robin fashion. The shared state consists of all array elements and the fields of all objects (matching *ss* in our model), and the mutexes that the JVM associates with objects (matching *mss* in our model). The program counter, local variables and call stack constitute the local state of each thread (matching *tss* in our model). We describe JESS in more detail in Chapter 5. In this section, we briefly discuss important implementation details related to HBRs and POR.

Sleep sets JESS incorporates the *sleep sets* reduction with both DPOR and lazy DPOR to further reduce redundant schedules explored [God96]. Naively combining DPOR with sleep sets is potentially unsound; an addendum [FG11] was released for the original DPOR paper [FG05] describing a sound combination which we have implemented. The sleep sets algorithm requires a valid dependency relation; thus, we use D (Definition 1) as the sleep-sets dependency relation with both DPOR and lazy DPOR; as discussed in §4.3, the lazy relation D' (Definition 4) is not a valid dependency relation.

DPOR optimisation for mutex operations The DPOR algorithm can be optimised by observing that it is not possible to reverse an unlock event and lock event on the same mutex. When a lock event e_2 races with an earlier unlock event e_1 it is thus only necessary to add a backtracking point to the preceding lock event of $e_1.tid$ that locks the mutex $e_1.obj$ [FG05]. Our DPOR implementation incorporates this optimisation. The optimisation does not apply to lazy DPOR.

Representing and recording HBRs as hashes We described the HBR as a partial-order over the set of events in a schedule in §4.2. We store regular HBRs and lazy HBRs efficiently using a method proposed in [MQ07a]. We classify all events as either write or read events as briefly described in §4.2. Each event e is also extended with an additional component $e.numWrites$ which counts the number of write events that have occurred on $e.obj$ up to and including e. The per-thread index, $e.pti$, encodes the total-order over

events of a thread and the number of writes, $e.numWrites$ encodes the order over events on the same object. This means that the HBR of a schedule is implicitly and canonically represented by the set of events in the schedule; we store a hash of this set to represent the HBR, and implement HBR caching based on these hashes. Note that hash collisions can lead to unsoundness. We describe this approach in more detail in §5.4.4.

During exploration, regardless of which algorithm is in use, we record both the HBR and lazy HBR for every terminal schedule. We make use of this recorded data during our evaluation (§4.6).

4.6 Experimental Evaluation

In this section, we evaluate:

- the potential reduction offered by the lazy HBR by showing the extent to which distinct terminal HBRs have identical terminal *lazy* HBRs when running DPOR (§4.6.1);
- HBR caching vs. lazy HBR caching, in terms of number of terminal lazy HBRs explored vs. the number of schedules explored (§4.6.2);
- DPOR vs. lazy DPOR, in terms of number of terminal lazy HBRs explored vs. the number of schedules explored and time taken (§4.6.3), with the caveat that lazy DPOR is unsound in general.

We use a set of 79 publicly available benchmarks. Our full results table (including the benchmark ids used in the graphs in this section), the JESS tool and our benchmarks are available online:

`https://github.com/mc-imperial/jtool-sct`

Benchmarks Our multithreaded Java benchmarks are largely drawn from prior works [FF09, FF13, CWY11, EP14, Sen08, PJ14, RM09, PL11b, PL11a]. For each benchmark, we require a target method with deterministic behaviour (given the same schedule). Thus, any nondeterminism (from random numbers, time, user input, etc.) had to be fixed; e.g. random number generators were given fixed seeds. For efficiency, we ensure that the method can be executed repeatedly without restarting the JVM. To achieve this, we had to manually modify most benchmarks, for instance, by adding code to reinitialise data before each schedule. We also modified benchmarks to reduce the amount of memory allocated, work performed and the number of threads created. Unlike in stress testing,

Suite	#bench	#unique	#w/o mutexes	#barriers	#sp	source
Spin14	8	4	0	2	127–883	[PJ14]
CCompar	41	24	0	0	24–439	[RM09]
JGF	11	8	8	5	45–40618	[SB01]
ASE11	5	5	0	0	457–1580	[PL11a]
Rhino	4	3	0	0	1480–3151	[B+07]
StringUtils	1	1	1	0	387	[Nit14]
Regression	9	9	1	1	14–40	-
Total:	79	54	10	8		

Table 4.1: Benchmark summary.

this is generally the best approach when performing SCT; many of the benchmarks were not designed with exhaustive testing in mind and the memory overhead for race detection can be large.

Table 4.1 provides a summary of the benchmarks, which consist of several benchmark suites. "#bench" denotes the number of benchmarks that we derived from each suite. In some cases, we created an additional version of a benchmark by varying the parameters (e.g. number of threads); "#unique" denotes the number of these that are truly distinct. "#w/o mutexes" denotes the number of (not necessarily unique) benchmarks that do not use mutexes. "#barriers" denotes the number of benchmarks that use barriers for synchronisation. We would not expect our lazy approaches to provide benefit for "embarrassingly parallel" benchmarks that do not exhibit synchronisation or synchronise only via barriers, but we still include such benchmarks. The "#sp" column shows the minimum and maximum number of *scheduling points* associated with benchmarks in a suite. More specifically, we count the number of scheduling points where more than one thread is enabled from the *first* schedule of each benchmark (which is the same for all techniques). We provide this as a metric of the benchmark complexity. All benchmarks use at least 2 threads; most use 2–4 threads; `repworkers-8t-8` had the largest thread count with 9 threads.

The "source" column indicates the source or prior work from which we obtained the benchmark suite. The *Spin14* suite contains several tests including a test for `cache4j`, a multithreaded in-memory Java object cache. The *CCompar* suite, from a study of concurrency bug finding tools [RM09], includes a large number of benchmarks, such as `elevator` (discrete-time elevator simulator), `philo` (dining philosophers simulation), `alarmclock`, `boundedbuffer` and `piper` (producer-consumer airline simulator with a known deadlock bug). The *JGF* (Java Grade Forum) suite includes multithreaded kernels and simulations such as `series` (Fourier coefficient analysis), `lufact` (LU factorisation), `crypt`

(IDEA encryption), `motecarlo` (Monte Carlo simulation) and `raytracer` (3D ray tracer). The *ASE11* suite includes `cocome` (prototype trading system for supermarkets), `credemo` (high-level prototype for a system providing WiFi at airports), `daisy` (simple file system developed as a challenge for verification tools) and `papabench` (a model of on-board control software for an aerial vehicle). We created the *Rhino* suite, which tests the Rhino JavaScript engine [B+07], using two tests from Rhino's bug tracker and a third test constructed by us in which 2 or 4 threads access disjoint locations in a shared JavaScript array. The *StringUtils* suite contains a single test case that we found online [Nit14]; the class being tested is from the spymemcached[3] project (a memcached[4] client written in Java). The *Regression* suite contains tests we used to guide the development of JESS. We skipped one benchmark that requires Java 1.6 (recall that JESS targets Java 1.7) and one benchmark whose memory requirements we could not reduce sufficiently for feasible analysis.

Experimental setup We conducted our experiments on a Linux cluster, with Red Hat Enterprise Linux Server release 6.4 and Oracle's 64-bit JDK1.7.0_60 Java VM. We ran each variation of SCT: lazy and non-lazy HBR caching, and lazy and non-lazy DPOR, on each benchmark with a limit of 100,000 terminal schedules. In each case we recorded the number of terminal schedules explored, the set of terminal lazy and regular HBRs explored, and the total time associated with exploration. For lazy DPOR, we also recorded whether we encountered a blocking operation inside a critical section (Issue 1) or detected the potential of mutex-deadlock (Issue 2). As noted, we do not detect all cases where lazy DPOR may miss terminal states (Issue 3). However, we compare the terminal lazy HBRs to see if lazy DPOR missed any terminal states explored by DPOR within the schedule limit. We henceforth assume that the number of schedules, HBRs, and lazy HBRs refers to the number of *terminal* schedules, *terminal* HBRs, and *terminal* lazy HBRs, respectively.

4.6.1 Potential reduction offered by lazy HBR

To study the extent to which the lazy HBR can identify equivalent terminal states regarded as distinct by the regular HBR we compare, for each benchmark, the set of terminal lazy and regular HBRs explored by regular DPOR. For a given benchmark we have the following inequality (modulo possible hash collisions when storing HBRs):

$$\#\text{states} \leq \#\text{lazy HBRs} \leq \#\text{HBRs} \leq \#\text{schedules} \leq 100{,}000$$

[3] `http://code.google.com/p/spymemcached/`
[4] `http://memcached.org/`

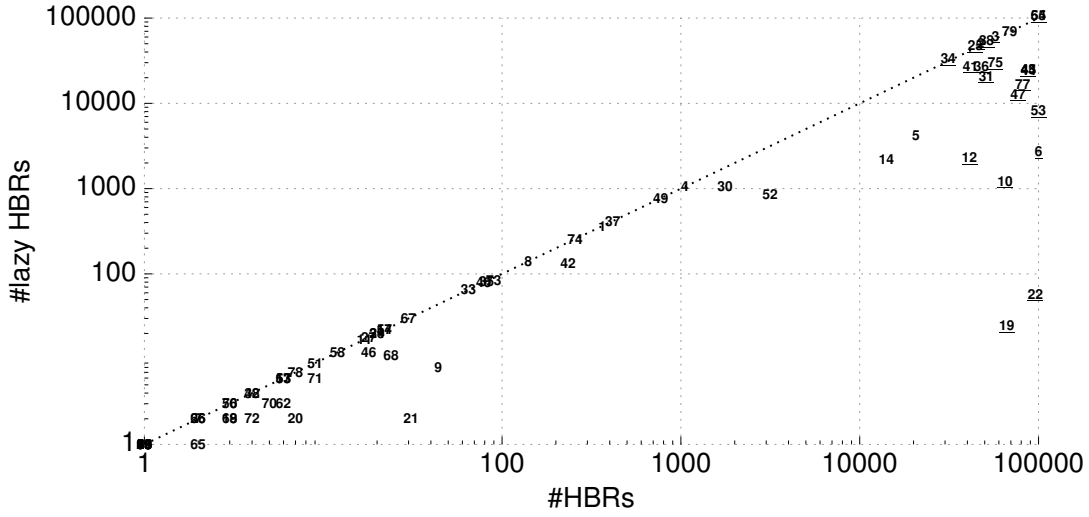


Figure 4.4: The number of terminal HBRs and terminal lazy HBRs explored by the first 100,000 terminal schedules of DPOR.

If the difference between #lazy HBRs and #HBRs is large then, because #lazy HBRs is a tighter upper bound for #states, significant benefit can be gained by exploiting the lazy HBR.

Figure 4.4 plots our results using a log scale. Each point is a benchmark id; if an id has coordinates (x, y) this means that regular DPOR explored x regular and y lazy HBRs for the benchmark. A benchmark is underlined if DPOR hit the schedule limit, in which case unexplored terminal states are likely to remain; otherwise all terminal states were explored. By the above inequality a benchmark cannot lie above the diagonal; a benchmark below the diagonal indicates that fewer lazy vs. regular HBRs were explored. Increasing the schedule limit could cause underlined benchmarks to move, but they could *not* move closer to the diagonal: any previously unseen lazy HBR is also a previously unseen HBR. Of the 79 benchmarks, 46 lie on the diagonal; in 10 cases this is expected as mutexes are not used (see Table 4.1). For the other 36 cases, equality between #lazy HBRs and #HBRs suggests the benchmarks do not exhibit coarse-grained locking or that the DPOR algorithm did not reveal any redundant HBRs before hitting the schedule limit.

There are 33 benchmarks below the diagonal: the lazy HBR could be exploited when analysing these benchmarks. DPOR completed in 18 of these cases. Across the 33 benchmarks below the diagonal, 80% of the unique HBRs explored were found to be redundant. Further schedules could cause this percentage to decrease, but the number of redundant

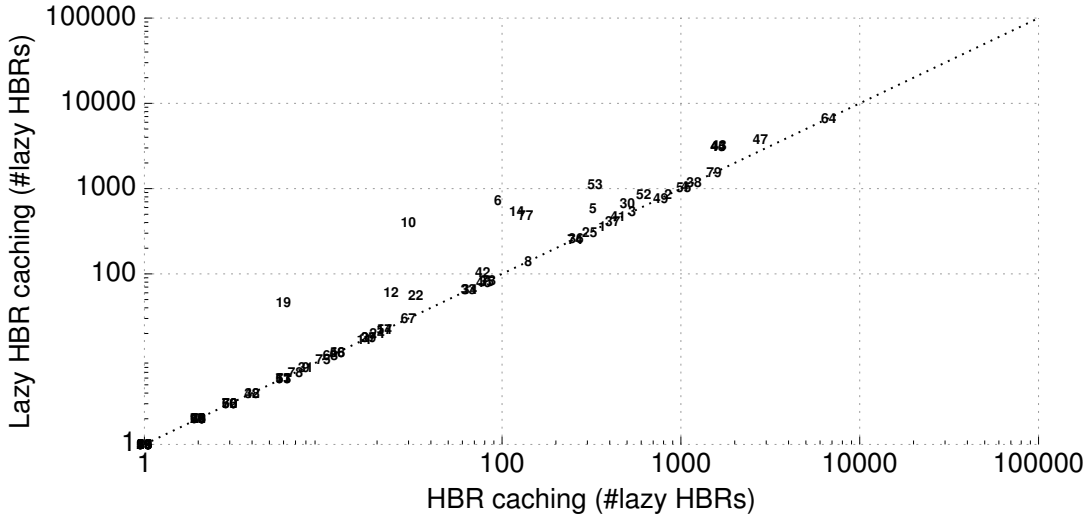


Figure 4.5: The number of terminal lazy HBRs explored by the first 100,000 terminal schedules of lazy HBR caching and HBR caching.

HBRs (910,007 total) would only increase or stay the same. For 17 benchmarks, the percentage of redundant HBRs was 70% or greater. If we ignore these extreme cases we still find that 49% of the HBRs explored were redundant.

4.6.2 Comparing lazy and regular HBR caching

We evaluated lazy vs. regular HBR caching by comparing the number of lazy HBRs explored by both techniques within the schedule limit. Figure 4.5 plots our results using a log scale. Each point is a benchmark id; an id at (x, y) indicates the number of lazy HBRs explored by HBR caching and lazy HBR caching, respectively. We henceforth use *states* to refer to lazy HBRs since (by the above inequality) they provide the closest estimate available to the number of unique states explored. By the above inequality a benchmark cannot lie below the diagonal—lazy HBR caching cannot explore fewer terminal states than HBR caching. A benchmark above the diagonal indicates that lazy HBR caching explored more terminal states than regular HBR caching within the schedule limit. We predicted that at most 33 benchmarks could benefit from exploiting the lazy HBR. We found that for 18 benchmarks lazy HBR caching explored more terminal states than regular HBR caching (i.e. there are 18 benchmarks above the diagonal). As expected, regular HBR caching never explored more lazy HBRs. Across the 18 benchmarks that saw a benefit, lazy HBR caching explored a total of 8,969 (84%) more terminal lazy HBRs than

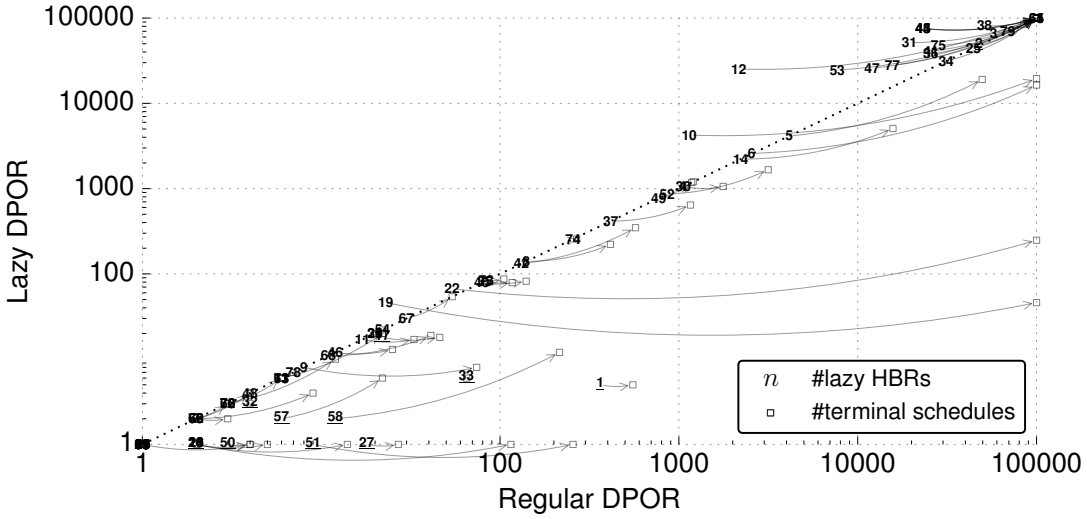


Figure 4.6: Number of terminal lazy HBRs (id) and terminal schedules (square) explored for each benchmark by the first 100,000 terminal schedules of regular and lazy DPOR.

regular HBR caching.

4.6.3 Lazy vs. regular DPOR

Figure 4.6 compares lazy and regular DPOR based on the number of lazy HBRs and schedules explored within the schedule limit, using a log scale. We again use *states* to refer to lazy HBRs. Each benchmark is represented by its id connected to a square. A benchmark id with coordinates (x, y) indicates that x vs. y states were explored by regular vs. lazy DPOR. The id is underlined if lazy DPOR encountered a blocking operation within a critical section or a may-deadlock cycle (see §4.4.3). A square at (x, y) indicates that regular and lazy DPOR completed after x and y schedules, respectively, for the associated benchmark.

A benchmark id above the diagonal shows that lazy DPOR managed to explore more states than DPOR within the schedule limit. A square below the diagonal shows that lazy DPOR completed within fewer schedules than regular DPOR. The benefit of lazy DPOR is shown when the benchmark id is above the diagonal and/or when the square is below the diagonal (more states explored by fewer schedules). Benefit was seen for 39 benchmarks; in these cases, lazy DPOR explored 344,161 (91%) more states than regular DPOR. For example, consider benchmark 10 in Figure 4.6, a configuration of `cache4j`

from the *Spin14* suite. Lazy DPOR explored more states than regular DPOR (4,248 vs. 1,144), and regular DPOR hit the schedule limit while lazy DPOR completed after 19,494 schedules.

Some benchmark ids, e.g. 19, 22, 14 and 5, appear on or close to the diagonal indicating that both algorithms explored the same or a similar number of states. However, if there was no schedule limit, *all* benchmark ids would appear on the diagonal because both techniques will eventually explore all terminal states. The substantial benefit of lazy DPOR in these cases is shown by the squares, which all lie significantly below the diagonal (recall that the graph has a log scale). Again, lazy DPOR was able to complete on all of these benchmarks; DPOR completed on only half of these and required many more schedules to do so.

For some underlined benchmark ids (e.g. 1, 33 and 51) we see that both algorithms completed with regular DPOR exploring more states than lazy DPOR (the ids lie below the diagonal). In these cases, lazy DPOR encountered a blocking operation within a critical section or a may-deadlock cycle; as discussed in §4.4.3, lazy DPOR is very likely to miss many states in these cases.

For a number of benchmarks, e.g. 12, 53, 77 and 31, neither DPOR nor lazy DPOR completed exploration. However, notice that lazy DPOR explored substantially more states than DPOR within the schedule limit, indicated by the ids lying above the diagonal. We believe these examples show a real benefit of lazy DPOR; complete terminal state coverage is probably infeasible but lazy DPOR provides much greater terminal state coverage.

Missed states and DPOR-fallback We note that DPOR never explored a lazy HBR hash that was not also explored by lazy DPOR except for the cases where lazy DPOR encountered a blocking operation within a critical section or a may-deadlock cycle (see §4.4.3). Of course, where lazy DPOR surpassed the number of states explored by DPOR, lazy DPOR may have unsoundly skipped terminal states (that we cannot detect because DPOR did not reach these states within the schedule limit). However, the fact that lazy DPOR explored a superset of the terminal states explored by DPOR (except for the detected exceptions) gives us confidence that lazy DPOR is an effective heuristic for efficiently attempting to enumerate most terminal states.

We also note that a blocking operation within a critical section or a may-deadlock cycle was always detected within just one or two schedules; thus, it would be possible to fallback to DPOR in these cases with almost no overhead.

Lazy vs. regular DPOR: analysis time Figure 4.7 compares the analysis time associated with regular and lazy DPOR. The reported times are *not* averaged over multiple

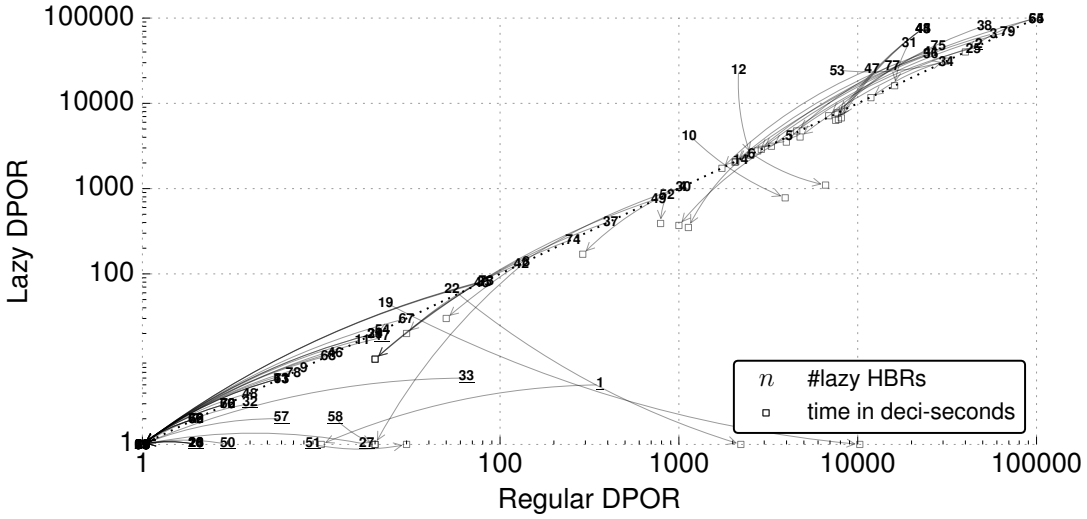


Figure 4.7: Number of lazy HBRs (id) explored and deci-seconds taken (square) for each benchmark with regular and lazy DPOR.

runs, and are subject to fluctuations due to multiple benchmarks running simultaneously on individual nodes of our cluster; we preferred to optimise our compute resources towards exploring large per-benchmark schedule counts. Still, due to the large number of schedules per benchmark, it is likely that fluctuations are averaged out. Each benchmark is again represented by its id connected to a square; a benchmark id at (x, y) indicates that x and y lazy HBRs were explored by DPOR and lazy DPOR, respectively. A benchmark id is underlined if lazy DPOR encountered a blocking operation within a critical section or a may-deadlock cycle (see §4.4.3). A square at (x, y) indicates that DPOR and lazy DPOR completed or hit the schedule limit within x and y deci-seconds, respectively. Thus lazy DPOR was faster than DPOR for a benchmark if the associated square is below the diagonal. The timing results show that, for the majority of benchmarks, lazy DPOR was as fast or faster than DPOR. In the worst case, lazy DPOR took 28 seconds (4%) longer than DPOR to hit the schedule limit (for the `piper-6-3-2` benchmark with id 38). However, lazy DPOR explored over 30,000 (58%) more terminal lazy HBRs than DPOR for this benchmark, suggesting that the additional analysis time is worthwhile. Furthermore, since lazy DPOR explored different terminal lazy HBRs, it is difficult to compare the algorithms fairly as many of the schedules explored by each may have been significantly different. Per schedule, in the worst case, lazy DPOR was 20% slower than DPOR (for the `raxextendedenvfirst-1-3-3` benchmark). However, again, lazy DPOR explored over

50,000 (200%) more lazy HBRs than DPOR for this benchmark. Overall, we believe that Figure 4.7 shows the benefits of lazy DPOR when considering the time taken with the number of lazy HBRs explored.

Bug-finding Overall, bugs (uncaught exceptions or deadlock) were found in 25 benchmarks by at least one of the techniques; DPOR found bugs in all these benchmarks, while lazy DPOR missed bugs in 4 benchmarks where a blocking operation within a critical section or a may-deadlock cycle was encountered (see §4.4.3). However, as described above, we could easily fall back to DPOR in these cases with almost no overhead. Excluding these cases, lazy DPOR found bugs using the same or fewer schedules than regular DPOR, requiring fewer schedules in 13 cases.

4.7 Related work

Optimal DPOR [AAJS14] improves upon DPOR by guaranteeing that precisely one schedule from for each HBR is explored (whereas DPOR may still redundantly explore some schedules with the same HBR). In this chapter, we used the original, simpler DPOR algorithm [FG05] and our lazy DPOR algorithm is also based on this. In future work, we could compare lazy DPOR with optimal DPOR, and try to make a lazy version of optimal DPOR. The improvement provided by the lazy HBR is orthogonal to optimal DPOR; optimal DPOR explores the minimum number of schedules (one) for each schedule equivalence class (HBR) while the lazy HBR is an alternative to the HBR that leads to larger schedule equivalence classes. However, the lazy HBR cannot typically be used directly in place of the HBR because the lazy HBR is based on an invalid dependency relation.

As explored in Chapter 3, schedule bounding techniques mitigate schedule-explosion unsoundly by bounding the number of preemptions or delays in a schedule. Soundly combining preemption bounding with DPOR (so that a preemption bounded DPOR search is still guaranteed to explore all terminal states reachable within the preemption bound) is nontrivial [CMM13]. Bounded DPOR [CMM13] achieves this. In future work, we could combine schedule bounding techniques with lazy DPOR which might improve its bug-finding capability.

A *conditional* valid dependency relation [GP93] can be used to get increased reduction from POR techniques. However, this is also orthogonal to the lazy HBR and any improvement to the HBR (such as using a conditional dependency relation) could likely be used to improve the lazy HBR. The lazy HBR uses an *invalid* dependency relation to achieve reduction of mutex operations that is not possible via any POR technique (including the

use of a conditional dependency relation).

The causally-precedes relation (CPR) [SES+12] is used to soundly detect data races in a trace that cannot be detected when using the happens-before relation; re-orderings of critical sections are considered (without having to execute them) such that all read operations still read the same values as in the original execution. This is achieved via an offline, polynomial-time datalogue analysis of the trace; a finite re-ordering window can be used to make the analysis linear (but unsound). The authors note that they have not discovered a way to implement an online version of their approach. Thus, the intuition (that critical sections can sometimes be re-ordered) is similar to ours but both the goal and approach are different; in particular, we do not use anything similar to a datalogue analysis—we use an online analysis. In fact, calculating the lazy HBR hash adds no overhead compared to computing the HBR hash (such as when performing lazy HBR caching) and our lazy DPOR implementation uses vector clocks just like in DPOR; using vector clocks to compute the CPR is currently unsolved, as noted by the authors.

The maximal causality reduction (MCR) [Hua15] is a reduction technique for SCT that explores the state-space of a concurrent program with a provably minimal (optimal) number of schedules with respect to a maximal causal model (MCM). The MCM is extracted from a schedule as a set of quantifier-free first-order formulae and captures the largest possible set of equivalent executions that can be obtained from re-ordering events. Note that the representation of a schedule is more detailed than in many SCT tools (such as CHESS, Maple, and JESS); for example, read and write events include the values that are accessed. An offline analysis using an SMT solver identifies alternative interleavings that lead to different states. The approach goes beyond what is possible with POR (e.g. even optimal DPOR), considers critical section re-orderings (like the lazy HBR), and (we believe) will still detect mutex-deadlock (unlike lazy DPOR). Thus, this approach appears to be the "truly optimal" reduction approach for SCT, surpassing both optimal DPOR and lazy DPOR in a sound algorithm. On the other hand, the runtime overhead of executing SMT queries can be high and is unpredictable in general; more evaluation would be useful to determine whether this high overhead and greater reduction is typically worthwhile in comparison to the low overhead and smaller reduction associated with lazy DPOR and optimal DPOR. The implementation of MCR is not currently available. We believe that POR and the lazy HBR are still useful techniques when efficient, light-weight, online analyses are desired.

A symbolic method efficiently represents *all* HBRs of a program up to some depth-bound, allowing input nondeterminism and weak memory behaviours to be captured [AKT13]. However, unlike DPOR, this approach cannot handle deep schedules of

programs with large loop bounds, due to infeasibly large SMT queries. The MCR [Hua15] has also been extended [HH16] to support the TSO and PSO memory models, again using an SMT solver. Chronological traces [AAA+15] allow optimal DPOR to check programs under the TSO and PSO weak memory models *without* using an SMT solver.

4.8 Conclusion

We have presented the lazy HBR that provides reduction beyond POR for programs that use mutexes. We have shown the large *potential* reduction from using the lazy HBR and the large *practical* improvement that lazy HBR caching and lazy DPOR can provide benefit over their non-lazy counterparts, although we note that lazy DPOR is unsound in general. In future work, it would be useful to investigate refining the lazy DPOR algorithm so that it is sound for programs without mutex-deadlock states. It would also be useful to consider optimal DPOR [AAJS14] and maximal causality reduction [Hua15].

5 Implementing an SCT tool

Despite the fact that researchers have produced many concurrency testing tools in recent years, there is a lack of detailed material on how to create such tools. In this chapter, we describe implementation details of JESS,[1] our systematic concurrency testing tool for Java programs (used to evaluate the lazy HBR in §4.6). We stress that this is not a description of straightforward engineering. We focus on subtle technical issues that required innovative solutions. Our contribution is to present these issues and our solutions in a cohesive manner as a resource for future researchers. In §5.1, we provide an overview of how to use JESS. In the remainder of the chapter, we present technical details of how JESS works. In §5.2, we describe challenges associated with instrumenting Java programs. In §5.3, we describe key details of creating an SCT tool. In §5.4, we cover some more advanced implementation issues relating to partial-order reduction (POR), including our race detection algorithm that we believe is more efficient than prior work. We discuss related work in §5.5 and conclude in §5.6.

5.1 Overview of the tool

In this section, we show how to use JESS to perform systematic concurrency testing of a Java program.

5.1.1 Creating a concurrency test case

Systematic concurrency testing typically requires writing a concurrency test case, similar to a unit test but with multiple threads. For this purpose, JESS provides a `ConcurrencyTestCase` interface with a single `execute()` method, as shown in Listing 5.1. The `execute` method will be executed repeatedly by JESS and must be deterministic modulo schedule nondeterminism. Given a program or library, the approach taken to create a concurrency test case can vary greatly. For example, given a simple compression tool like pbzip2, a concurrency test case might test the entire program by calling the program

[1] The name JESS is a homage to CHESS [MQB+08], a systematic concurrency testing tool for C/C++ and C# programs. The J in JESS signifies that our tool is for Java programs.

```
1  package org.jtool.test;
2
3  public interface ConcurrencyTestCase
4  {
5    public void execute() throws Exception;
6  }
```

Listing 5.1: `ConcurrencyTestCase` interface.

entry point with appropriate command line arguments so that a small file is compressed or decompressed using several threads. The test case could assert that the resulting file matches the expected output. Given a complex server application, a concurrency test case might test a component of the server in isolation, so as to avoid methods that perform network communication, which most SCT tools (including JESS) will not be able to control; we discussed this and other common barriers to applying SCT in §3.4.2. A test case for the server might create several threads that send messages to the component under test (without using network communication) in order to ensure that all interleavings of the messages are handled appropriately.

In this overview, we consider Mozilla's `Rhino`, a JavaScript interpreter and compiler written in Java. We will write a concurrency test case in which two Java threads are interpreting JavaScript in parallel. This example is from the `Rhino` benchmark suite used in the evaluation of our lazy happens-before relation in §4.6 and is based on a bug found on Rhino's bug tracker.

The test case class, `TestRhinoBug1`, is shown and described (via comments) in Listing 5.2. If the result of executing `thread1code` is not `2.0` then a `RuntimeException` is thrown (line 35). The constructor for `TestRhinoBug1` includes the actual JavaScript code and is shown in Listing 5.3. Our JavaScript program defines three global variables `x`, `f1` and `f2`. On line 4, `x` is defined to be an object with two fields, `POSITIVE_INFINITY` and `NEGATIVE_INFINITY`, which both map to the value `2.0`. On line 6, `f1` is defined to be a function that simply returns the value of `x.NEGATIVE_INFINITY`. In our test case, `f1` should always return `2.0` and this is the property that is checked on line 35 of Listing 5.2. On line 10, `f2` is defined to be a function that adds 16 additional fields to `x`. Note that, in Rhino, accesses to disjoint fields on the same object from multiple threads are synchronised using monitors such that `f1` should always return `2.0`. In our test case, `f1` and `f2` are executed concurrently in the main thread and in an additional thread respectively (see lines 25–31 of Listing 5.2).

```java
package rhinotest1.org.mozilla.test;

import org.jtool.test.ConcurrencyTestCase;
import rhinotest1.org.mozilla.javascript.Context;
import rhinotest1.org.mozilla.javascript.Scriptable;

public class TestRhinoBug1 implements ConcurrencyTestCase {
  private final String globalDefinitions, thread1code, thread2code;

  private static class OtherThread extends Thread { ... }

  public TestRhinoBug1() { ... }

  public void execute() throws Exception {
    // Create the JavaScript context in which to execute code.
    final Context cx = Context.enter();
    // Use the interpreter mode.
    cx.setOptimizationLevel(-1);
    try {
      // Get a standard scope in which to execute code.
      final Scriptable scope = cx.initStandardObjects();
      // Execute the globalDefinitions JavaScript code.
      cx.evaluateString(scope, globalDefinitions, "init", 1, null);

      // Create and launch a second thread that will execute thread2code.
      OtherThread t2 = new OtherThread(scope, thread2code);
      t2.start();
      // In the current thread, execute thread1code and store the result.
      final Object result = cx.evaluateString(scope, thread1code, "thread1", 1, null);
      t2.join();
      // Both threads have now finished executing JavaScript code.

      // If the result is not 2.0 then an error occurred.
      final double res = Context.toNumber(result);
      if(res != 2.0) {
        throw new RuntimeException("Unexpected result!");
      } else {
        System.out.println("SUCCESS!");
      }

    } finally {
      Context.exit();
    }
}
```

Listing 5.2: `TestRhinoBug1` class—a concurrency test case for the `Rhino` JavaScript interpreter.

```
1  public TestRhinoBug1() {
2    // Defines x, f1 and f2.
3    globalDefinitions =
4        "var x = { POSITIVE_INFINITY : 2, NEGATIVE_INFINITY: 2 };"
5
6      + "var f1 = function() {"
7      + " return x.NEGATIVE_INFINITY; "
8      + "};"
9
10     + "var f2 = function() {"
11     + "  for (var i = 0; i < 16; i++) "
12     + "  { "
13     + "    x[\"prop\" + i] = 1; "
14     + "  }"
15     + "};";
16
17   // Thread 1 will evaluate f1();
18   thread1code = "f1();";
19
20   // Thread 2 will evaluate f2();
21   thread2code = "f2();";
22 }
```

Listing 5.3: Constructor for the `TestRhinoBug1` class of Listing 5.2.

5.1.2 Creating a test harness

In order to execute the test case, it is necessary to create a test harness; this is the code that will be executed as a Java application (or executed from a testing framework, such as `JUnit`) and will execute the test case repeatedly using JESS. There is no provided test harness or command line tool for executing test cases; thus, JESS could be described as more of a concurrency testing *library* than a tool.

A test harness that executes the Rhino test case is shown in Listing 5.4. First, the test case is created (line 10). As explained already, a test case should be deterministic modulo scheduling nondeterminism. However, this is often difficult to achieve unless the system under test has been designed with concurrency testing in mind. In particular, it is common for certain static constants to be initialised *lazily* or for commonly-used objects to be *cached*; this can make test cases nondeterministic as the exact sequence of operations performed by the `execute` method will vary depending on whether objects have been initialised or cached. To alleviate this, we perform two warm up executions where the test case is executed without the use of JESS (line 13); this is sufficient to make future executions of the Rhino test case deterministic, although this is not guaranteed in general. Without the warm up executions, JESS throws an exception during the second execution

because, when trying to replay a prefix of the first execution, it encounters a different operation; thus, JESS detects this nondeterminism and throws an exception so that the user is made aware of the issue.

To use JESS, we create an `ExecutionManager`, passing the test case to the constructor (line 19), and we set the scheduling strategy to `DFSStratagy` (line 20), to perform a straightforward depth-first search (see §3.2.1). We then execute the test case up to 16 times (we chose this number for illustration) under the control of JESS by calling `doExecution()` in a loop (line 28). On line 29, we check whether an error occurred and if so we output the execution count and the exception. If `doExecution()` throws a `NoMoreExecutionsException` (which would be caught on line 35) then all schedules were explored; this is determined by the given scheduling strategy. For example, if we had enabled (non-iterative) preemption bounding (see §3.2.2) then `NoMoreExecutionsException` being thrown would indicate that all schedules were explored *within the preemption bound*. Lines 39–43 show how we output the number of executions, number of terminal hashes and the number of lazy terminal hashes (see Chapter 4 for an explanation of terminal state hashes).

5.1.3 Performing offline JDK instrumentation

JESS works by performing dynamic Java bytecode instrumentation to automatically monitor and control threads, which we describe in more detail in §5.2. It is necessary to perform offline instrumentation of the JDK once before using JESS, by running the `JDKInstrumenter`:

```
java -cp jtool-runtime.jar org.jtool.jdkinstr.JDKInstrumenter
```

The `JDKInstrumenter` will output `rt_instr.jar`, the instrumented JDK, which is used when running the test harness.

5.1.4 Running the test harness

We can now run the test harness:

```
java \
  -Xbootclasspath/p:rt_instr.jar \
  -javaagent:jtool-runtime.jar   \
  -cp rhino_src_1-1.jar          \
  MainHarness
```

```java
import org.jtool.runtime.ExecutionManager;
import org.jtool.strategy.DFSStrategy;
import org.jtool.strategy.NoMoreExecutionsException;
import org.jtool.test.ConcurrencyTestCase;
import rhinotest1.org.mozilla.test.TestRhinoBug1;

public class MainHarness {

  public static void main(final String[] args) throws InterruptedException {
    final ConcurrencyTestCase testCase = new TestRhinoBug1();
    System.out.println("Warm up executions:");
    try {
        testCase.execute(); testCase.execute();
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
    System.out.println("Warm up done.");

    final ExecutionManager em = new ExecutionManager(testCase);
    em.setSchedulingStrategy(new DFSStrategy());

    final int EXECUTION_LIMIT = 16;

    System.out.println("Starting systematic concurrency testing:");
    try {
      int i = 0;
      for (; i < EXECUTION_LIMIT; ++i) {
        em.doExecution();
        if(em.currentExecutor.errorOccurred != null) {
          System.out.println(em.getNumExecutions() + ": "
            + em.currentExecutor.errorOccurred);
        }
      }
      System.out.println("Stopping after " + i + " executions.");
    } catch (NoMoreExecutionsException e) {
      System.out.println("Completed.");
    }

    System.out.println("Num executions: " + em.getNumExecutions());
    System.out.println("Num normal terminal hashes: "
      + em.getNumNormalTerminalHashes());
    System.out.println("Num lazy terminal hashes  : "
      + em.getNumLazyTerminalHashes());
  }
}
```

Listing 5.4: `MainHarness` class for executing the `TestRhinoBug1.java` test case.

The `-Xbootclasspath/p:rt_instr.jar` parameter loads the instrumented JDK before any other classes. The `-javaagent:jtool-runtime.jar` parameter allows JESS to perform dynamic bytecode instrumentation. The `-cp rhino_src_1-1.jar` parameter ensures the Rhino source code and test case (Listing 5.2) are on the class path, assuming these classes are packaged into `rhino_src_1-1.jar`. Finally, `MainHarness` is our test harness class (Listing 5.4), which we assume is already on the class path. The output of running the test harness is the following:

```
Warm up runs:
SUCCESS!
SUCCESS!
Warm up done.
Starting systematic concurrency testing:
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
SUCCESS!
Stopping after 16 executions.
Num executions: 16
Num normal terminal hashes: 1
Num lazy terminal hashes  : 1
```

As indicated by the `SUCCESS!` messages, the 2 warm up executions pass successfully, as do the 16 controlled executions. The `Stopping after 16 executions` message indicates that there are more executions that were not explored. The number of lazy and non-lazy terminal hashes indicates that only one terminal state was explored; thus, 15 of the executions were redundant as they reach the same terminal state as the first execution. See §4.5 for an explanation of lazy and non-lazy terminal hashes.

We can improve the testing by modifying line 20 of Listing 5.4 to the following:

```
em.setSchedulingStrategy(new DFSStrategy().setDpor(true).setSleepSets(true));
```

This enables sleep sets (see §4.5) and DPOR (see §4.4.1). Running the test harness again gives the following output:

```
Warm up runs:
SUCCESS!
SUCCESS!
Warm up done.
Starting systematic concurrency testing:
SUCCESS!
2: java.lang.RuntimeException: Unexpected result!
3: java.lang.RuntimeException: Unexpected result!
4: java.lang.RuntimeException: Unexpected result!
5: java.lang.RuntimeException: Unexpected result!
6: java.lang.RuntimeException: Unexpected result!
7: java.lang.RuntimeException: Unexpected result!
8: java.lang.RuntimeException: Unexpected result!
9: java.lang.RuntimeException: Unexpected result!
10: java.lang.RuntimeException: Unexpected result!
11: java.lang.RuntimeException: Unexpected result!
12: java.lang.RuntimeException: Unexpected result!
SUCCESS!
SUCCESS!
SUCCESS!
Completed.
Num executions: 15
Num normal terminal hashes: 15
Num lazy terminal hashes  : 15
```

The output shows that on executions 2–12 a `RuntimeException` was thrown; this is because `thread1code` did not return the expected result of 2.0 (see line 36 of Listing 5.2). Thus, the bug was revealed on these executions. Also note the `Completed` message which indicates that there are no more executions, even though only 15 executions were explored. Since DPOR and sleep sets are sound reductions (see Chapter 4), the total number of terminal hashes indicates that there are only 15 unique terminal states in this test case which were all explored. Thus, all behaviours of this test case have been explored.

The bug demonstrated by this test case occurs because Rhino implements JavaScript objects using a hash table that is resized when the 16^{th} entry is added; a new map is allocated and the elements are copied into the new map but there is a small window in which the map appears to be empty. Thus, on certain interleavings, the main thread sees `x.NEGATIVE_INFINITY` as `undefined` (see line 7 of Listing 5.3).

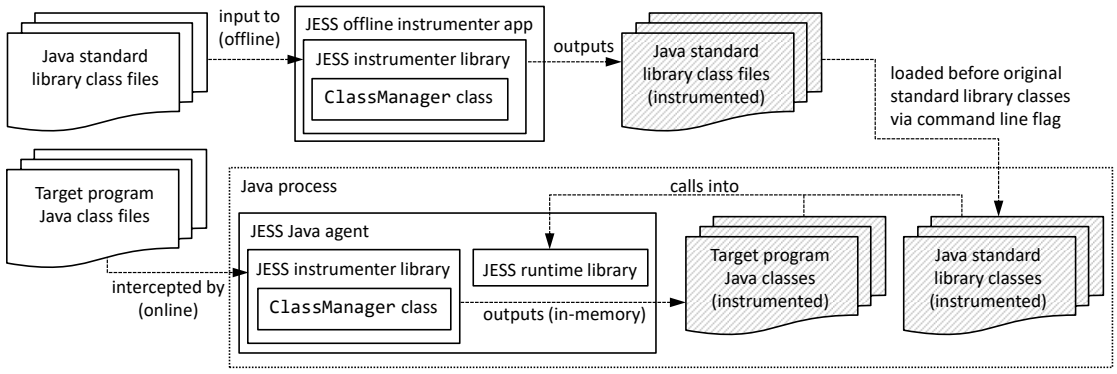


Figure 5.1: A diagram showing how our JESS tool instruments code offline and at run-time. The Java standard library class files are instrumented offline and then loaded via a command line flag. The target program class files are instrumented online (i.e. at run-time). The instrumented code calls into the JESS runtime library so that thread execution is controlled.

5.2 Instrumenting Java programs

In this section we describe many subtle issues involved in instrumenting Java code and our innovative solutions as a resource for researchers. For context, an overview of how instrumentation is performed when using JESS is given in Figure 5.1. We focus on general issues relating to Java instrumentation and dynamic analysis. SCT-specific issues are described in §5.3 and §5.4. We proceed as follows:

- §5.2.1: we motivate the project and approach by discussing the advantages of dynamic bytecode instrumentation and the advantages of targeting a high-level intermediate representation (IR) like Java bytecode.
- §5.2.2: we introduce the ASM library [ELC02], Java agents, and our `ClassManager` class; components like these are likely to be used in any Java bytecode instrumentation project.
- §5.2.3: we introduce the method doubling technique which is crucial in allowing us to instrument the Java standard library classes while keeping uninstrumented versions available for our tool.
- §5.2.4: we describe how we maintain shadow fields, arrays and objects to store per-field, per-array and per-object information, respectively.

- §5.2.5: we cover various issues encountered and techniques used when instrumenting Java code using the method doubling approach.
- §5.2.6: we note some limitations of our approach.

5.2.1 The advantages of dynamic bytecode instrumentation

When writing a dynamic analysis tool, such as JESS or a dynamic data race detector, it is necessary to decide whether to perform offline instrumentation (such as compile-time instrumentation), dynamic instrumentation, or a combination of both. For example, Google's `ThreadSantizier` version 1 [SI09] performs dynamic instrumentation of binaries using valgrind, which requires minimal user effort, while version 2[2] performs compile-time instrumentation of the LLVM intermediate representation (IR), which greatly reduces runtime overhead with increased user effort. In the context of Java programs, the differences between offline and dynamic instrumentation are less significant; class files of the target and its dependencies can typically be found and instrumented easily since the files must be on the class path and the overhead of loading and JIT compiling class files is already relatively high (compared to loading binaries), which makes the slowdown of dynamic instrumentation less of a bottleneck. In JESS, we use dynamic bytecode instrumentation where possible for ease-of-use as well as offline bytecode instrumentation where necessary.

Targeting a high-level IR like Java bytecode also has some advantages over targeting lower-level representations like LLVM or x86. Java bytecode is a simple IR, arguably simpler than than x86 and LLVM, which makes instrumentation more straightforward. Analysing memory accesses is a common concern for dynamic analysis tools and is much easier in the case of Java compared with languages like C++; heap regions are clearly identifiable—they are either fields or array elements. An out-of-bounds memory access will always cause an exception to be thrown, and local variables cannot be shared between threads. In fact, heap accesses (i.e. field accesses and array accesses) have their own instructions, distinct from those that manipulate local variables. In contrast, in x86, catching out-of-bounds memory accesses is a nontrivial problem. We have also found that most Java debuggers continue to work in the presence of bytecode instrumentation, whereas debugging instrumented binaries is less straightforward. Finally, the existence of the ASM library [ELC02] for instrumenting Java bytecode is an advantage; it is a powerful tool that greatly simplifies interacting with Java bytecode and is used in the Java ecosystem. For example, when using the Apache Maven Shade plugin for Maven (a popular Java build tool), ASM is used to rewrite references to dependent class files

[2] `https://github.com/google/sanitizers/wiki/ThreadSanitizerAlgorithm`

in order to avoid conflicts. Thus, it is already used every day by Java developers when building their Java applications.

5.2.2 Use of the ASM library, Java agents and our `ClassManager`

ASM [ELC02] is a Java library for manipulating Java bytecode. It can be used both online (modifying or creating classes in memory) and offline (reading and writing class files on disk). ASM provides two different APIs for instrumenting classes: the visitor API and the tree API.

With the visitor API, user-defined visitors are used to manipulate and/or create classes and methods. Note that the structure that is visited is the array of bytes that make up a class, not a class hierarchy as in the traditional Visitor design pattern. Indeed, as described in [ELC02], the visitor API intentionally avoids using an object representation of the class for much better performance compared with alternative approaches. The visitor API works as follows. The user defines a class that extends the `ClassVisitor` abstract class, which includes methods like `visitField` and `visitMethod`. An instance of the visitor is passed to a `ClassReader`, which reads a class and calls the relevant method of the visitor upon reading a particular structure. For example, `visitField` is called when a field is read. `ClassWriter` is a provided class visitor (it extends `ClassVisitor`) that outputs the class file that is described by the calls to the various `visit*` methods. Thus, passing a `ClassWriter` to a `ClassReader` will result in the `ClassWriter` outputting the same class file with no changes. To manipulate a class, the developer writes many class visitors that will wrap a `ClassWriter` (and each other), forming a chain of visitors; each visitor (by default) delegates to the next. A user-defined visitor overrides certain `visit*` methods to call different methods on the next visitor. For example, a visitor that overrides `visitField` to do nothing will cause all fields to be removed, as the next visitor will never receive any calls to `visitField`. The outer-most visitor is passed to a `ClassReader` in order to get a modified version of the original class.

An important consequence of this design for tool authors is that at no point is the entire class stored in memory. This makes it difficult to query arbitrary properties of a class, such as whether a private field is unused, without visiting the entire class and creating some object representation of the results. Indeed, from a performance perspective, a class is ideally only read once, from start to finish, with the chained visitors performing their manipulations in this single pass.

In contrast, the tree API of ASM (which is implemented using the visitor API) provides an object representation of an entire class (via the `ClassNode` class), potentially resulting

in an object for every instruction in every method. Thus, it is heavy-weight but more convenient for querying arbitrary information about a class.

In JESS, we use the visitor API where possible for increased run-time performance. Our visitors occasionally need information about other classes (mainly the fields and superclasses of a class) and thus use the tree API to read entire classes, skipping method bodies to reduce the space and time overhead.

Java agents In Java, classes are loaded lazily. The Java platform provides a service to allow classes to be dynamically instrumented as they are loaded via *Java agents*. A Java agent is a JAR file with certain properties and allows us to register a `ClassFileTransformer` object that will be invoked each time a class is loaded. As explained in §5.1.4, the Java agent is specified on the command line. The agent receives a byte array for each class that is loaded. We use the ASM library to parse and modify the byte arrays. When transformers first execute, certain classes are already loaded and so cannot be modified or can only have method bodies modified (no fields or methods can be added/removed). We discuss this further below.

`ClassManager`: accessing information about classes without reflection As explained above, while visiting a class, it is often useful to retrieve information about the current class or some other class. Since the class in question may not be loaded (since we instrument classes before they are loaded), it is not possible to use Java's reflection API (`java.lang.reflect`). Perhaps the strangest example of this is ASM's `ClassWriter.getCommonSuperClass(...)` method, which uses reflection despite the fact that this approach is likely to fail due to classes not being loaded; we had to override this method to use our reflection-free approach.

To solve this issue, we created the `ClassManager` class, a container that acquires and caches information about classes when requested. We use ASM's tree API to read the entire class (but skipping method bodies) from which we obtain the set of superclasses and the fields of the class and its superclasses. This allowed us to give each field of every object a unique index which was useful in efficiently representing field read and write operations (see §5.4.4). We store this information in a `ClassInfo` object.

5.2.3 Instrumenting Java code and standard libraries via method doubling

When instrumenting Java code, instrumenting the standard library classes (e.g. `ArrayList`) is desirable, otherwise this code cannot be monitored. For example, in JESS,

it is necessary to track shared memory accesses and ignoring standard classes would cause many accesses, such as on collections like lists and sets, to be missed. Unfortunately, this is nontrivial [FSS04, PSE07]. First, many standard classes are already loaded before Java agents can execute; for certain classes, the JVM disallows instrumentation completely and, for the remaining classes, only allows *method bodies* to be modified dynamically (so adding/removing/modifying fields and method signatures is disallowed). Second, the Java agent code and any methods that are called by the instrumentation typically call methods in standard classes, which can lead to inadvertently monitoring tool code or even infinite recursion where instrumented code calls into itself. Thus, it is desirable to keep an uninstrumented version of the standard classes to be used by the tool. When creating JESS, we realised that it is actually useful to keep an uninstrumented version of *all* code, not just standard library classes: this allows us to execute warm up executions, as described in §5.1.2, on completely uninstrumented code; we also wanted any static initialisers, which may create objects and call non-static methods, to execute uninstrumented code. Thus, in general we wanted to execute uninstrumented code by default and, only at certain points, start executing the instrumented test case.

One approach we could have used to maintain instrumented and uninstrumented versions of code is the twin class hierarchy (TCH) approach [FSS04], which was originally proposed as a solution to instrumenting standard library classes. In TCH, a renamed copy of every class is created and instrumented. References to classes in instrumented code are modified to refer to the instrumented versions. However, this approach presents several problems that must be handled carefully, as explained in [FSS04]. One major issue highlighted in [PSE07] is that native methods[3] only work on the original class and it is not clear how this can be worked around in general.

We instead use *method doubling*, based on the ideas presented in [PSE07]. A renamed copy of every method is added to the target class and instrumented. For a method named `method`, we add a copy of the method named `method$instr` and then apply our chain of visitors to the new method's body. References to methods in instrumented code are modified to refer to the instrumented versions. An exception is native method copies, which are replaced with non-native methods that simply call the original native method. We perform offline instrumentation on the JDK classes and use dynamic instrumentation for all other classes. This method doubling approach has relatively few issues that need to be handled compared to TCH.

[3]Note that native methods can be used by developers to invoke functions written in lower-level languages like C/C++, but they are also used in standard library classes to access low-level features, such as class loading, retrieving object hash codes, I/O, etc.

Doubling constructors In Java bytecode, a constructor is a method named `<init>` with `void` return type. Creating a renamed copy is not possible as constructors *must* be named `<init>`. To solve this, we overload each constructor with a copied, instrumented version, appending an additional parameter to the method signature of type `ConstrInstrMarker` (a class that we define). We use this type to ensure the constructor's signature is unique. We change constructor invocations in instrumented code by (a) adding the `ConstrInstrMarker` parameter to the signature and (b) pushing `null` onto the JVM stack just before the invocation.

5.2.4 Shadow fields, shadow arrays and shadow objects

Prior work [PSE07] notes that per-object data needed by instrumentation can be stored in a weak identity hash map[4] so that the layout of fields in the original class is unchanged. This works around the fact that changing the fields of some standard classes (such as `Object` and `String`) causes the HotSpot JVM to crash because it makes assumptions about the structure of these classes.

In JESS, we need per-object data for tracking objects that are used as monitors/locks and we need per-field and per-array-element data for tracking reads and writes to shared memory. Using one or more hash maps to store this data is undesirable: the slowdown is significant due to the typically large number of field reads and writes. A convenient and efficient solution for storing per-object data would be to add a field to the `java.lang.Object` class, referencing a *shadow object* for storing per-object data. Unfortunately, this causes the HotSpot JVM to crash due to assumptions about the `Object` class. Thus, for per-object data we are forced to use a weak identity hash map. We map each object to a `SyncObjectData` object, which is used for storing synchronisation information.

For per-field data, we use a more efficient solution, similar to what is described in [FF10]. For each field named `field`, we create a *shadow* field named `field$shadow` with type `SyncObjectData` in the same class. When a field is read from or written to, we add instrumentation to initialise the shadow field if it is null, and call a callback, passing in the `SyncObjectData`. We exclude certain classes when adding shadow fields, such as `java.lang.String`, as this would cause JVM crashes because the HotSpot JVM makes assumptions about the layout of the fields of these classes. Omitting these classes may be an issue for certain dynamic analyses, as it will not be possible to track accesses to fields of these classes. However, for JESS, this is sufficient because the classes are either immutable

[4] The weak identity hash map uses the identity of its keys for hashing and equality testing, and maps objects to their per-object data. The keys are weak references so that the objects can still be garbage collected.

or only used internally by the JVM. Note that, in JESS, we also do not need to add shadow fields for `final` (read-only) fields, as field read operations are always independent.

For per-array-element data, we were also forced to use an identity hash map because array objects cannot contain fields. We map an array object to an array of `SyncObjectData` of the same length. When an array is accessed, the array and offset are passed to a callback that finds the corresponding `SyncObjectData` array in the hash map. Inspired by [FF10], we added a basic cache in front of the identity hash map so a small number of the most recently looked up arrays can be retrieved without looking in the hash map to improve performance of the common case where code accesses many elements of the same array.

5.2.5 Issues and techniques

In this subsection, we cover various issues encountered and techniques used when instrumenting Java code using the method doubling approach.

Instrumentation within constructors Throughout the development of JESS, we came across several issues related to instrumenting constructors. For example, we insert a callback inside every constructor to detect when objects are created; this allows us to detect objects that are created from native code, which would not be possible if we merely inserted callbacks before constructor invocation.[5] However, the reference to the object being constructed, `this`, cannot be passed to any method until after the call to `super`; the JVM will reject bytecode that attempts this. Thus, we were forced to insert the callback after the call to `super`. Identifying the `INVOKESPECIAL` bytecode instruction that corresponds to the call to `super` is nontrivial. For example, if the call to `super` is of the form `super(new MyClass());` this will result in a `NEW` instruction to allocate an instance of `MyClass`, an `INVOKESPECIAL` instruction to invoke the constructor of `MyClass`, and then a second `INVOKESPECIAL` to invoke `super`. In general, dataflow analysis is required to identify the call to `super` but this is not ideal when trying to implement a lightweight visitor. We used a simple heuristic that we believe works reliably for bytecode generated by the Java compiler. We keep track of the number `INVOKESPECIAL` instructions that invoke `<init>` methods that are visited, minus the number of `NEW` instructions visited, in a variable called `invokeSpecialMinusNewCount`. Thus, when we visit an `INVOKESPECIAL` instruction that invokes `<init>` and `invokeSpecialMinusNewCount` is 0, then we assume that this is the call to `super`, because there was not a previous `NEW` instruction (except for those followed by an `INVOKESPECIAL` instruction).

[5] Note that we do not instrument the constructor of `java.lang.Object` as this causes JVM crashes, so we do in fact add additional callbacks before `java.lang.Object` instances are created.

We also encountered an unexpected issue in instrumenting `LinkedHashMap`. One of the constructors of `LinkedHashMap` invokes a particular super-constructor which invokes a method `init` that is overridden in `LinkedHashMap`; this method accesses fields of `LinkedHashMap`. Thus, before the invocation of `super()` returns, the fields of `LinkedHashMap` are are accessed. At one point in development of JESS, we initialised shadow fields of a class in every constructor, after the invocation of `super`. Thus, this unexpected scenario led to uninitialised shadow fields being accessed. In §5.4.4, we explain an optimisation that made it unnecessary to initialise shadow fields in this way, making this a non-issue, but we note that this may be an issue for other tools.

Intercepting calls to JDK methods We commonly need to replace calls to certain JDK methods with calls to callbacks. For example, in JESS, we replace calls to to `Object.wait`, `Object.notify` and `Thread.join` with calls to our own methods that simulate these operations. Since we are already doubling methods, it may seem sensible to simply replace the body of the instrumented method. However, key methods that we needed to intercept, such as `Object.wait` and `Object.notify`, are in classes on which we do not perform method doubling in order to avoid JVM crashes. Conveniently, these methods are `final`, which means they cannot be overridden. Thus, we can trivially replace calls to these methods by finding all occurrences of the `INVOKEVIRTUAL` instructions that match the method signature in question and replacing them with `INVOKESTATIC` instructions that call a static callback method with the same signature as the replaced method, except for an additional prepended parameter that receives the object on which the method was original invoked. We note one potential pitfall: invocations of `join` (a final method) on subclasses of `Thread` are expressed in the bytecode as a virtual invocation of `SomeClass.join` where `SomeClass` is the subclass of `Thread`. Thus, we could not simply replace calls to `Thread.join` but instead had to consider all `join` methods (with the matching signature) where the class referenced is a subclass of `Thread` (or `Thread` itself). We use `ClassManager` to check if this condition holds and replace the call if so.

Instrumenting methods of `java.lang.Object` using method body doubling Although we can avoid modifying final methods of `Object`, the virtual methods of `Object` (`hashCode`, `equals`, `clone` and `toString`) are harder to handle. We cannot double these methods because doing this would crash the JVM. Furthermore, there is no point in doubling these methods in other classes because these instrumented versions will not be overriding methods of `Object` and so bytecode that calls one of the methods, say `Object.hashCode$instr`, will be rejected by the JVM because the method does not ex-

ist. Changing instrumented code to invoke `Object.hashCode` is also incorrect, because this will invoke the uninstrumented method. To solve this, we introduce *method body doubling*, an alternative to method doubling that is only used in overridden methods of `Object`. For these methods, we revisit the method body a second time, adding a branch at the start of the method body to jump to one of the two versions of the body. We instrument only the first version of the method body. The branch condition checks whether the field `java.lang.Thread.instrumented` of the current thread is true; this is a field that we add to the `Thread` class to keep track of whether the thread is executing instrumented code. The approach is demonstrated by the code in Listing 5.5 and Listing 5.6, which shows an overridden `hashCode` method before method body doubling and the same `hashCode` method after method body doubling, respectively.

```
1  @Override
2  public int hashCode() {
3    return this.a + this.b + this.c;
4  }
```

Listing 5.5: An overridden `hashCode` method before method body doubling.

```
1  @Override
2  public int hashCode() {
3    if(Thread.currentThread().instrumented) {
4      // instrumented version of method body
5      int temp;
6      Callbacks.fieldRead(this.a$shadow, ...);
7      Callbacks.fieldRead(this.b$shadow, ...);
8      temp = a + b;
9      Callbacks.fieldRead(this.c$shadow, ...);
10     temp = temp + c;
11     return temp;
12   } else {
13     return this.a + this.b + this.c;
14   }
15 }
```

Listing 5.6: The `hashCode` method from Listing 5.5 after method body doubling. Note that the code in the instrumented body is only a representation.

The fact that this approach is needed is unfortunate; it is less elegant and less efficient (due to the additional field access), but most importantly, it requires significant care to make sure that the value of `Thread.instrumented` remains correct when entering and leaving instrumented code. In particular, every callback invoked from instrumented code has to set `Thread.instrumented` to `false` on entry and to `true` on return. Furthermore,

the JVM can start executing class loader code at various points and it was necessary to update the `Thread.instrumented` at this time; we achieved this by instrumenting the `ClassLoader.loadClass` method.

Synchronized blocks (monitors) In Java, every object can be used as a monitor; i.e. a combination of a mutex and a condition variable. Java code uses `synchronized` blocks to lock and unlock the monitor, which results in a pair of `MONITORENTER` and `MONITOREXIT` instructions in bytecode. Calling `wait` on a monitor (i.e. an object) unlocks the monitor. Calling `notify` on a monitor unblocks one thread that is waiting on the monitor (there is no guarantee about which thread is chosen—see 17.2.2 Notification in [GJS+13]) which will immediately try to relock the monitor. Calling `notifyAll` on a monitor unblocks all threads that are waiting on the monitor and they will all try to lock the monitor. As with any lock operation, there is no guarantee on the order in which threads will succeed in locking the monitor; indeed, some other thread that did not call `wait` on the monitor may preempt all waiter threads and acquire the lock first.

In JESS, we needed to control precisely which threads are awoken and when they lock the monitor. Thus, we reimplemented `wait`, `notify` and `notifyAll`. Furthermore, since `MONITORENTER` and `MONITOREXIT` pairs must be within the same method, we also had to reimplement these operations, e.g. so that we could unlock a monitor within our implementation of `wait`. Finally, Java bytecode also has `synchronized` methods; one might expect these to compile to ordinary methods that contain `MONITORENTER` and `MONITOREXIT` instructions, but this is not the case. Thus, we used an additional visitor to transform all `synchronized` static and instance methods into equivalent unsynchronized methods that instead use `MONITORENTER` and `MONITOREXIT` instructions; this visitor comes before our visitor that replaces `MONITORENTER` and `MONITOREXIT` instructions in the chain of visitors.

Intercepting thread start and controlling thread entry points via run-method-renaming In JESS, we needed a callback from a parent thread that is starting a child thread and a callback from the child thread before it starts executing. Furthermore, we needed to control whether a child thread would start executing instrumented code.

In Java, for an existing parent thread to start a new child thread, the parent thread calls `Thread.start` on the child thread object. Since `Thread.start` is a virtual method and we don't want to monitor its method overrides, we insert a callback in the body of `Thread.start$instr` to gain control just before the thread is about to be started.

Ensuring that child threads start executing instrumented code is less straightforward. The JVM starts executing the `Thread.run` method of the child thread object. Thus,

we could instrument the body of `Thread.run` to call a callback and then potentially call `Thread.run$instr` to start executing instrumented code. However, `Thread.run` is a virtual method and so may be overridden, in which case it would not get executed and we would not intercept execution of the child thread. Thus, we could also instrument every `run` method that overrides `Thread.run`. However, these methods (and even `Thread.run`) are public methods and so are not necessarily only invoked as a thread entry point. Thus, in the callback, we would somehow have to check whether the call is due to a thread starting. We came up with an elegant and more straightforward solution. We rename all `run` methods (with the same signature as `Thread.run`) to `run$orig`; calls to `run` are also updated to call `run$orig`. Note that these run methods are also doubled, resulting in additional methods named `run$orig$instr`. At this point, there are no `run` methods (not even `Thread.run`); thus, the JVM would be unable to start threads. We then add our own version of `Thread.run` which is guaranteed to be the only thread entry point. Our `Thread.run` method calls `run$orig` or `run$orig$instr`, depending on the value of the `Thread.instrumented` field of the current thread. We also call a callback just before calling `run$orig$instr`. The `Thread.instrumented` field of the child thread is set to true in the `Thread.start$instr` callback.

5.2.6 Limitations

Our approach for finding the invocation of `super` from within a constructor is a heuristic and may not hold for all bytecode, such as bytecode that was generated from a language other than Java. This could be solved using a dataflow analysis.

5.3 Implementing systematic concurrency testing for Java

In this section, we describe the design of the SCT components of JESS, which use the instrumentation described in the previous section. We believe the design elements described could be applied to any SCT tool, not just to a Java tool. We proceed as follows:

- §5.3.1: we describe two key classes in the design of JESS, `Executor` and `ExecutionManager`.
- §5.3.2: we describe how we store information about each thread using the `ThreadData` class and how we use this to serialise execution.
- §5.3.3: we describe the `schedule` method which implements scheduling points.

- §5.3.4: we introduce the notion of a scheduling strategy which gives an algorithm for how to explore the schedule-space. We describe the random and DFS scheduling strategies, and give an example of how the DFS strategy explores the schedule-space.
- §5.3.5: we describe how to implement entering (locking) a monitor as an example of how synchronisation operations are implemented using `ThreadData` and the `schedule` method.

In §5.4, we cover more advanced features related to POR.

5.3.1 Executor and ExecutionManager

There are two key classes that implement SCT in JESS: `Executor` and `ExecutionManager`.

A fresh instance of `Executor` is created for each schedule. It stores all data needed for the current schedule, such as the list of threads and the map of shadow objects and shadow arrays. It contains public methods that are called from the instrumentation callbacks, such as when a monitor is locked/unlocked, a field is accessed, etc. It also stores the scheduling strategy, which is an object that implements the `SchedulingStrategy` interface (see §5.3.4). The `Executor` class queries the scheduling strategy to ask which thread should be scheduled at each scheduling point.

The `ExecutionManager` class is used by the test harness (see §5.1.2) to start performing SCT. It stores the `ConcurrencyTestCase` and allows controlled execution to occur via the `doExecution` method, which invokes the instrumented `run$instr` method of the `ConcurrencyTestCase`. It creates and stores the current `Executor` for each execution and stores the set of execution hashes (see §4.5).

5.3.2 `ThreadData` objects and thread serialisation

As explained in §5.2.4, we use a weak identity hash map to map each Java object to a `SyncObjectData` object; the hash map is necessary because we cannot modify the `Object` class without crashing the JVM. This map is stored in a field of `Executor`. In a similar fashion, we store a `ThreadData` object for each Java thread that has been started. We add a `threadData` field to the `Thread` class to efficiently map each `Thread` to its corresponding `ThreadData` object; a hash map in `Executor` is not needed, as we can modify the `Thread` class without crashing the JVM. We also store the `ThreadData` objects in a list in `Executor` because we wish to track the thread creation order and also efficiently access each thread given a unique integer thread id that we assign, starting from 0.

Recall that, in SCT, execution is serialised so that only one thread executes at a time. We call the only executing thread the *active* thread and `ThreadData` objects contain a boolean flag called `active` which is true iff the corresponding thread is the active thread. Furthermore, to achieve thread serialisation, the inactive threads will be blocked from within a JESS callback. Note that the threads would not necessarily be blocked in the original program. Thus, to track the enableness of a thread in the original program, each `ThreadData` object contains a boolean flag called `enabled` which is true iff the corresponding thread would be enabled in the original program. When a thread is "blocked" (according to JESS), for example trying to lock a monitor that is already locked, we set the `enabled` field of its `ThreadData` object to false. The thread continues to execute and will eventually reach a scheduling point (see the `schedule` method described below) where its `enabled` flag will be read; the thread will then be descheduled according to JESS (it will be blocked and another thread will be released). The `ThreadData` object is used as a monitor in conjunction with the `active` flag to allow a descheduled thread and a scheduled thread to synchronise. This is described in more detail below. An example of entering a monitor is described in §5.3.5.

5.3.3 The `schedule` method

The `schedule` method is integral to the design of JESS. The schedule method represents a scheduling point (the start of a visible operation) and is invoked from within callbacks, such as from a field access callback or a monitor enter callback. Thus, the schedule method is the only place where the active thread can be descheduled so that another thread can become active and continue executing. The parameters of the schedule method provide information about the *next* visible operation. The idea of having a method that receives information about the next visible operation and that represents a scheduling point was inspired by the similar `Controller::Schedule` function in Maple [YNPP12] (as was the meaning of `enabled` and `active`), although we did not base our implementation of the method on any existing code.

A simplified version of the `schedule` method is shown in Listing 5.7. The method receives information about the current thread: its `ThreadData` object, the `SyncObjectData` object that its next visible operation accesses (corresponding to e.g. a field or monitor) and the operation type (`OpType`) of its next visible operation. The method updates the next operation of the current thread (by modifying the `ThreadData` object on lines 8 and 9). The details of the next sync object being accessed and the operation type are used in POR, which is explained in Chapter 4 and §5.4; updating this now ensures that the scheduling

```
1  public final void schedule(final ThreadData currThreadData,
2                             final SyncObjectData syncObject,
3                             final OpType opType) {
4    final Op prevOp = currThreadData.getCurrOp();
5    final SyncObjectData prevOpSyncObjectData = currThreadData.currOpSyncObjectData;
6
7    // Set the next operation of the current thread.
8    currThreadData.currOpSyncObjectData = syncObject;
9    currThreadData.currOpType = opType;
10
11   // Query the scheduling strategy for the next thread id to schedule.
12   final int nextTid = strategy.getNextThread(
13       currThreadData,
14       syncObject,
15       opType,
16       threadList,
17       prevOp,
18       prevOpSyncObjectData);
19
20   // Handle the deadlock case.
21   if (nextTid == SchedulingStrategy.THREAD_ID_DEADLOCK) {
22     handleDeadlock();
23     return;
24   }
25
26   final ThreadData nextThreadData = threadList.get(nextTid);
27
28   // Release the chosen thread and then block current thread.
29   if (nextThreadData != currThreadData) {
30     currThreadData.active = false;
31     synchronized (nextThreadData) {
32       nextThreadData.active = true;
33       nextThreadData.notifyAll();
34     }
35     synchronized (currThreadData) {
36       while (!currThreadData.active) {
37         currThreadData.wait();
38       }
39     }
40   }
41 }
```

Listing 5.7: A simplified version of the `schedule` method, which implements a scheduling point in JESS.

strategy can make use of the information. Next, the method queries the scheduling strategy to get the next thread id that needs to be scheduled (line 12). Note that the next thread id to be scheduled is a function of the state of the scheduling strategy, which can persist between executions, as well as the parameters to `getNextThread`. We show the implementation of the random scheduling strategy in §5.3.4. If no thread can be chosen because all threads are disabled, the scheduling strategy returns `THREAD_ID_DEADLOCK`, which is handled on line 21; we will not go into the details of this but, essentially, this execution ends via all threads throwing an exception. Finally, the chosen thread is released and the current thread is blocked (lines 29–40). Note that if the chosen thread is the current thread, then the method simply returns. Otherwise, the current thread is set to inactive and the next thread is set to active. The current thread then waits until it is made active (i.e. scheduled). Thus, this method shows how serialisation of threads is achieved; each thread blocks itself and releases the next scheduled thread. We give an example of using the `schedule` method below (§5.3.5).

Scheduling point for new thread. It is worth noting that a newly started thread must immediately block until it is scheduled and this can only occur when the active thread calls `schedule`. Thus, "thread start" is the thread's first scheduling point, but the thread cannot call `schedule` to achieve this, as `schedule` can only be called by the active thread. Thus, a newly started thread must instead execute the code in the `executedRun` method, shown in Listing 5.8, to block until it becomes active (line 7).

```
1  public final void executedRun() {
2    final Thread currThread = Thread.currentThread();
3    final ThreadData currThreadData = getThreadData(currThread);
4
5    synchronized (currThreadData) {
6      while (!currThreadData.active) {
7        currThreadData.wait();
8      }
9    }
10 }
```

Listing 5.8: A simplified version of the `executedRun` method, which is called by a newly started thread.

As explained in §5.2.5, when the `executedRun` callback returns the thread will continue executing its instrumented entry point. The `ThreadData` object of the newly started thread is added to the thread list by the parent thread and the thread is also marked as enabled. Thus, when the active thread reaches the next scheduling point, it may choose

to schedule the newly started thread. At this time, the newly started thread will become active and the thread will either be unblocked from line 7 or the loop (at line 6) will not be entered at all. Either way, the newly started thread does not continue executing until it becomes active.

5.3.4 Scheduling strategy

The scheduling strategy is an object that determines which thread to schedule next. A scheduling strategy must implement the `SchedulingStrategy` interface. The key method of this interface is `getNextThread`, which is invoked from the `schedule` method (see line 12 of Listing 5.7), and returns the next thread id that should be scheduled. Recall that, during SCT, we perform many executions, each of which must execute the target program from the start. Thus, when performing a DFS of the schedule-space, we would expect two successive executions to share a common prefix and so a DFS scheduling strategy object must store information about the previous executions in order to replay the common prefix. In contrast, a controlled random scheduling strategy does not need to store any information about previous executions. The `SchedulingStrategy` interface contains one other method, `prepareForNextExecution`, which is invoked before executing the next schedule. A DFS strategy will implement `prepareForNextExecution` to prepare its stack data structure for the next schedule, as we show below.

A random scheduling strategy A `getNextThread` method that implements the controlled random scheduler algorithm described in §3.2.4 is shown in Listing 5.9. The method starts by constructing a list of the enabled threads (line 11). If the list is empty, then there are no enabled threads and we return `THREAD_ID_DEADLOCK` (line 19). Otherwise, we choose a random thread from the list and return its thread id (line 24). Note that this random strategy does not store any information about the previous scheduling points or previous executions; the only state that is maintained across executions and across invocations of `getNextThread` is the state of the random number generator (`this.rng`).

DFS example We describe our DFS scheduling strategy using an example. Note that the DFS strategy object stores the unexplored schedules using a stack data structure. Figure 5.2 shows the stack data structure at three different points for some program. At each call to `getNextThread`, a list of enabled threads is created (similar to in the random scheduling strategy); each entry in the list can be marked as *selected* and/or *done*. After the first execution, the stack is in the state shown by (a). At line 3, thread 2 (t2) has been created (or at least became enabled). At each call to `getNextThreadId`, the first

```java
@Override
public int getNextThread(
    final ThreadData currThreadData,
    final SyncObjectData syncObject,
    final OpType opType,
    final List<ThreadData> threadList,
    final Op prevOp,
    final SyncObjectData prevOpSyncObjectData) {

  // Create a list of the enabled threads.
  final List<ThreadData> enabledThreadList = new ArrayList<>();
  for(ThreadData td : threadList) {
    if(td.enabled) {
      enabledThreadList.add(td);
    }
  }

  // No enabled threads => deadlock.
  if(enabledThreadList.size() == 0) {
    return SchedulingStrategy.THREAD_ID_DEADLOCK;
  }

  // Return a random enabled thread.
  final int randomIndex = this.rng.nextInt(enabledThreadList.size());
  final ThreadData nextThread = enabledThreadList.get(randomIndex);
  return nextThread.threadId;
}
```

Listing 5.9: A simplified `getNextThread` method that implements a straightforward random scheduling strategy.

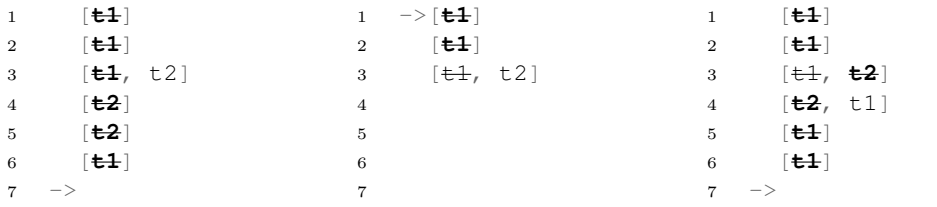


Figure 5.2: Examples of the stack from the DFS scheduling strategy. The stack grows downwards and the arrow indicates the top of stack pointer. The stack elements are shown as lists of thread ids (t1 and t2); **bold** indicates that the entry is *selected* while ~~strikeout~~ indicates that the entry is *done*. The stack is shown at three points: (a) after completing the first execution; (b) after invoking `prepareForNextExecution` after the first execution; and (c) after completing the second execution.

thread id in the list of enabled threads was returned and so the corresponding entry was selected and marked as done. Figure (b) shows the state of the stack after the call to `prepareForNextExecution` which is called before starting another schedule. Notice that entries have been popped until the top of stack contains an entry that is not done. Furthermore, the previously selected entry in the top of stack is deselected. Finally, the top of stack pointer is reset to the first entry; this pointer allows the entries on the stack to be replayed. During the second execution, the selected entries will be chosen (replayed) until the real top of stack is reached. Figure (c) shows the state of the stack after the second execution; the new entries in the stack were observed when executing the previously seen schedule prefix (lines 1–2), followed by the first previously unexplored thread (t2 on line 3), and then continuing with other unexplored transitions until reaching a terminal state. Note that t2 is now the first entry (on line 4) because we order threads in thread creation order, starting from the most recently executed thread and wrapping in a round-robin fashion. The next call to `prepareForNextExecution` will pop the top two elements off the stack and, in the next execution, t1 will be chosen at line 4.

5.3.5 Schedule example: enter monitor

We now describe the implementation of the `onEnterMonitor` method in `Executor`, which implements entering (locking) a monitor. The `onEnterMonitor` method is shown in Listing 5.10. The parameter `o` is the object/monitor that is being entered. We first check whether `o` is `null` and, if so, throw an exception (line 2), to mimic the behaviour of the `ENTERMONITOR` instruction. We obtain the `QueueData` object from `o` and store it in `oQueueDate` (line 6). The `QueueData` class contains bookkeeping information for monitors; the key pieces of bookkeeping information used are: the *entry set* (`entrySet`), which stores threads that are trying to enter the monitor; the *owner* (`owner`), which stores the thread that has entered (owns) the monitor; and `recursiveEntered`, which is used to implement recursive monitors. We check if no other thread owns the monitor (line 8) and, if so, we add the current thread to the entry set (line 10). This may seem counterintuitive since we could instead immediately update the owner thread to the current thread. However, we wish to ensure that there is a scheduling point immediately before the enter monitor operation; conceptually, the enter monitor operation occurs after the call to `schedule` (line 21). Thus, we add the current thread to the entry set so that, if the current thread t is preempted by another thread u (during the call to `schedule`), then thread u can set all threads in the entry set (including t) to be disabled (which occurs on line 34). If the current thread owns the monitor (line 11), we increment `oQueueData.recursiveEntered`

and return. This implements Java's recursive monitors; the owner thread of a monitor can re-enter the monitor many times. The `oQueueData.recursiveEntered` field is a counter that tracks how many times the owner thread has entered the monitor minus the number of times the thread has exited the monitor, excluding the first enter and last exit. There is no scheduling point on recursive enters and exits of a monitor; it is as if the operations never occurred in terms of scheduling. If another thread, u, owns the monitor (line 15), we add the current thread t to the entry set but also disable the current thread t. This ensures that another thread will be scheduled at the call to `schedule` (line 21) which is what we desire, since the current thread t cannot enter the monitor while thread u owns it. When the owner thread, u, is scheduled and exits the monitor, u will re-enable thread t. We then call `schedule` (line 21) which represents a scheduling point. When `schedule` returns, the current thread is going to become the owner of the monitor. Thus, there are several conditions that should hold, which we check using assertions (lines 23–26). Whenever any call to `schedule` returns, the current thread should be marked as enabled (checked on line 23). The entry set of the monitor should still contain the current thread (checked on line 24); we remove the current thread in this method on line 31. The monitor must not have an owner (checked on line 25). Finally, the `oQueueData.recursiveEntered` counter must be 0 (checked on line 26); this must be true whenever the monitor does not have an owner. We then set the owner of the monitor to the current thread (line 29), remove the current thread from the entry set (line 31) and, finally, disable all threads in the entry set (line 34).

5.4 Advanced SCT details

In this section, we describe SCT implementation details that relate to partial-order reduction (POR), including race detection and state-caching. We proceed as follows:

- §5.4.1: we motivate the approach of unifying all synchronisation operations as reads and writes on synchronisation objects, which greatly simplifies the implementation of POR techniques.
- §5.4.2: we describe how, in practice, transitions are represented as ops and how the dependency relation and happens-before relation can be defined over ops. This covers the prerequisite information to be able to describe our vector clock algorithms.
- §5.4.3: we describe our efficient vector clock algorithms for use with DPOR.

```
1  public final void onEnterMonitor(final Object o) {
2    if (o == null) { throw new NullPointerException(); }
3
4    final ThreadData currThreadData = getThreadData(Thread.currentThread());
5    final SyncObjectData oData = getSyncObjectData(o, OpType.ENTER_MONITOR);
6    final QueueData oQueueData = oData.getQueueData();
7
8    if (oQueueData.ownerThread == null) {
9      // No thread owns the monitor.
10     oQueueData.entrySet.add(currThreadData);
11   } else if (oQueueData.ownerThread == currThreadData) {
12     // The current thread owns the monitor.
13     oQueueData.recursiveEntered++;
14     return;
15   } else {
16     // Another thread owns the monitor.
17     oQueueData.entrySet.add(currThreadData);
18     currThreadData.enabled = false;
19   }
20
21   schedule(currThreadData, oData, OpType.ENTER_MONITOR);
22
23   assert currThreadData.enabled;
24   assert oQueueData.entrySet.contains(currThreadData);
25   assert oQueueData.ownerThread == null;
26   assert oQueueData.recursiveEntered == 0;
27
28   // Set the owner to be the current thread.
29   oQueueData.ownerThread = currThreadData;
30   // Remove the current thread from the entry set.
31   oQueueData.entrySet.remove(currThreadData);
32   // Disable all threads in the entry set.
33   for (final ThreadData td : oQueueData.entrySet) {
34     td.enabled = false;
35   }
36 }
```

Listing 5.10: The `onEnterMonitor` method which implements entering (i.e. locking) a monitor.

- §5.4.4: we describe ops (the `Op` class) in full detail, including how we represent thread and sync object ids, capture and hash the happens-before relation (HBR), and handle global objects and field offsets.
- §5.4.5: we describe how we implement barriers that support the *barrier wait* operation using read and write ops. We use three different types of ops and an additional barrier thread for each barrier.

5.4.1 Unified synchronisation operations

Various POR techniques require information about the HBR (see §4.2): sleep sets require the dependency relation; DPOR requires race detection, which we achieve using vector clocks; and HBR caching requires a concise representation of the entire HBR. This potentially means that every visible operation (henceforth, *op*) needs to be handled in a different way. In fact, each op could, in theory, need to be handled differently for each technique. This can be avoided by classifying every op type (such as entering and exiting a monitor, starting and joining a thread, barrier operations, memory read and writes, etc.) as either a read or a write on a shared object (sync object). This approach was used in CHESS [MQB+08] but we feel that some of the advantages were not emphasised or were omitted entirely and, as such, could be missed by future researchers, hence we detail them here. The advantages of the approach are as follows.

First, implementing/instrumenting additional synchronising operations is greatly simplified, as it is mainly a case of expressing these operations in terms of reads and writes on sync objects. Note that there is still some complexity in expressing which threads become enabled/disabled. We describe how we implemented the barrier operation in §5.4.5. Second, ops can conservatively be classified as writes to ensure soundness; POR techniques will explore all interleavings of writes to the same sync object. At a later point, certain ops can be reclassified as reads to gain a greater reduction. The two above advantages were described in [MQB+08].

Third, the definitions of read and write operations can be exploited to achieve optimised vector clock operations. We believe that state-of-the-art tools like CHESS do not take full advantage of this. We believe our vector clock operations (described in §5.4.3) improve upon the state-of-the-art in terms of space and time overhead. Vector clocks are essential for performing efficient race detection, as needed for DPOR, and they need to be updated after every op which is expensive and thus important to optimise.

Finally, the HBR relation of an execution (after every scheduling point) can be canonically represented as a *set* of ops without explicitly storing edges. Furthermore, an incre-

mental hash function can be used to efficiently maintain a hash of the HBR after every op without maintaining an actual set and without requiring the use of vector clocks. This allows for *HBR caching*, a form of state-caching, where every visited HBR hash is cached to avoid redundant execution from already visited HBRs (see §4.2). Thus, as long as ops are only implemented as reads and writes, HBR caching is very straightforward to implement. Yet, we believe that the simplicity and effectiveness of HBR may not be obvious to researchers. HBR caching was used by CHESS [MQB+08] although hashing is not mentioned in [MQB+08]; the hash-based approach is only described in a referenced technical report [MQ07a], where ops are introduced as reads and writes on variables, without explaining that these are abstract concepts and not just shared memory accesses. Furthermore, the hash-based approach is introduced to work around the complex soundness issues of combining partial-order reduction and preemption bounding. It may not be clear to other researchers that the HBR hashes can be used more generally, such as for estimating the number of states explored during concurrency testing. We describe the approach in §5.4.4.

5.4.2 Transitions as ops

As described in §4.2 (where ops were called *events*), transitions are not stored in SCT tools. Instead, a schedule:

$$E = \langle op_1, op_2, \ldots, op_k \rangle$$

is represented as a list of ops. The full representation of an op is revealed later in §5.4.4. For now, we assume that an op contains at least the following elements:

$$op = (tid, obj, opType, pti)$$

where *tid* is the thread id, *obj* is the sync object being accessed (e.g. a monitor or shared memory location), *opType* is the operation type (e.g. monitor enter, monitor exit, create thread), and *pti* is the per-thread index which denotes that this is the *pti*-th op executed by thread *tid*.

Each op type is classified as either a read or a write. Consequently, we refer to an op as being either a read or write depending on its op type. Let $IsWrite(op)$ be true iff *op* is a write op. While shared memory reads and writes are classified as reads and writes, respectively, an op that is a read/write is not necessarily a shared memory read/write; we present our approach for implementing the barrier wait operation in §5.4.5 which uses multiple read and write ops where the read ops update our barrier object.

In Definition 1, we gave the definition of a valid dependency relation D over transitions. In practice, we define a dependency relation, over ops (Ops) as follows:

Definition 8 (A practical dependency relation)**.** *A pair of operations are dependent,* $(\mathrm{op}_1, \mathrm{op}_2) \in D$*, iff either:*

1. $\mathrm{op}_1.tid = \mathrm{op}_2.tid$*, or*
2. $\mathrm{op}_1.obj = \mathrm{op}_2.obj \wedge (IsWrite(\mathrm{op}_1) \vee IsWrite(\mathrm{op}_2))$

In other words, a pair of ops are dependent iff they are from the same thread, or they access the same sync object and at least one is a write. The happens-before relation $\rightarrow_E$ is defined over the ops in E according to Definition 3, using the above dependency relation.

5.4.3 Efficient vector clock operations

Vector clocks are used to encode the happens-before relation and perform efficient race detection.[6] A vector clock is a map from thread ids to clocks (integers), typically implemented as a list of integers, $\langle c_1, \ldots, c_n \rangle$, where c_t stores the clock for thread t. In traditional data race detection, a clock value c_t represents the c_tth op of thread t. In SCT, we can instead follow the approach given in [FG05] and let c_t be the *global clock* of an op by thread t; that is, c_t is the index of an op within the schedule, E. This allows us to efficiently find the index of an op within E as needed for DPOR. Conceptually, each op $E(j)$ in a schedule is associated with a vector clock $E(j).VC = \langle c_1, \ldots, c_n \rangle$, such that thread t's last op that happens-before $E(j)$ is $E(c_t)$. Thus, the happens-before relation is encoded in the vector clocks:

Definition 9 (Happens-before relation using vector clocks)**.**

$$E(i) \rightarrow_E E(j) \text{ iff } i \leq E(j).\mathrm{VC}(E(i).tid)$$

We also ensure that $E(i).VC(E(i).tid) = i$. Thus, the global clock of an op can always be found in its vector clock at the position of the thread that executed the op and so ops from the same thread will always be totally-ordered. Figure 5.3 shows an example schedule with the vector clock of each op. Notice that the global clock of each op can be found within its vector clock at the position of the thread that executed the op. Observe that, according to Definition 9, $E(1)$ and $E(2)$ are unordered with each other and both happen-before $E(3)$, as required.

[6] Recall from §4.4.1 that we refer to pairs of ops from different threads that are directly-related in the HBR as races; this differs from the traditional definition of a data race.

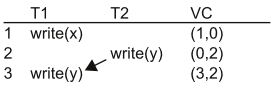


Figure 5.3: An example schedule showing the vector clock of each operation. The arrow indicates the single race.

Note that Definition 9 technically makes the happens-before relation reflexive, which contradicts our earlier definition (Definition 3) and implies that there are no races (according to Definition 7). To solve this, we conceptually ignore the reflexive edges in the HBR when detecting races.

Let:

$$join(VC_1, VC_2) = \langle max(\, VC_1(1),\, VC_2(1)), max(\, VC_1(2),\, VC_2(2)), \ldots, max(\, VC_1(n),\, VC_2(n))\rangle$$

denote the pointwise maximum of two vector clocks. Intuitively, we will obtain the vector clock of some op, *op*, using the *join* of the vector clocks of previous ops that race with *op*. In particular, note that, the pointwise maximum of $op.VC$ and the vector clocks of all ops that happen-before *op* is equal to $op.VC$.

In practice, it is not necessary to store a vector clock for each op. Instead, each thread t is associated with a vector clock $t.VC$, which stores the vector clock of the last op of thread t. Also, each sync object o is associated with a read vector clock, $o.readVC$, and a write vector clock, $o.writeVC$. The read vector clock stores the pointwise maximum of the vector clocks of all read ops that accessed o. The write vector clock stores the vector clock of the most recent write op that accessed o; note that this is equivalent to storing the pointwise maximum of the vector clocks of all write ops that accessed o because all writes to the same object are totally-ordered in the HBR, but the former description leads to a simpler and more efficient implementation. We describe how these two vector clocks are used in the context of race detection by considering the different types of races. Let $E(j)$ be an operation in the execution E; we wish to find the set of global clocks of ops that race with and occur before $E(j)$. That is, we wish to find $I = \{i \mid i < j$ and $E(i)$ races with $E(j)\}$. In the following, it is assumed that we only consider ops that occur before $E(j)$.

Write-read and write-write races If $E(j)$ is a read op that accesses o, then $E(j)$ can only race with the most recent write to o. Similarly, if $E(j)$ is a write op that accesses o, where there have been no reads from o since the last write, $E(j)$ can only race with the most recent write to o. In these cases, I will either be the empty set or the singleton set containing the global clock of the most recent write to o. Note that this global clock

is one of the clocks in $o.writeVC$; specifically, the one with the largest value. To avoid iterating over every clock in $o.writeVC$, we store the thread id of the last write, denoted as $o.lastWriteTid$. Thus, the global clock of the last write to o is $o.writeVC(o.lastWriteTid)$.

Read-write races If $E(j)$ is a write op that accesses o, and there has been at least one read of o since the last write to o (before $E(j)$), then $E(j)$ can only race with the reads from o since the last write. Note that we only need to consider the most recent read of o by each thread; an earlier read from a thread must happen-before the later read from the same thread, and so cannot race with a subsequent write. Furthermore, a pair of reads from *different* threads can also be ordered in the happens-before relation (via transitivity) and so only the most recent read of the pair may race with $E(j)$. Crucially, the "interesting" global clocks—those that correspond to all reads of o that occurred since the last write to o and that are not obscured by (related transitively in the HBR to) a later read of o—will be contained in $o.readVC$. Not every clock in $o.readVC$ necessarily corresponds to a read from o (or even a read op), but these other clocks are still required in order to track the combined happens-before information of all reads of o. In order to track which clocks in $o.readVC$ are interesting (as defined above) for race detection, we use a list of boolean values $o.readsMask = \langle b_1, b_2, \ldots, b_n \rangle$, implemented as a bitmask, where $o.readsMask(i)$ is true iff $o.readVC(i)$ is an interesting global clock that may race with $E(j)$. Thus, I will contain a subset of the interesting global clocks.

In the above, we have described the set of global clocks that *may* be in I (and that our algorithms will consider) because they access o. However, the clocks described may still happen-before $E(j)$ transitively (via operations that do not access o). We now present the algorithms that incorporate the above ideas.

Updating vector clocks The algorithm for updating vector clocks is shown in Listing 5.11. This `updateVectorClocks` method is invoked after each op that is executed. The thread data of the thread that executed the op is `t`; the global clock of the op is `globalClockOfSyncOp`; the sync object accessed by the op is `o`; and `write` is true iff the op is a write. We use the same notation for accessing the vector clocks, *lastWriteTid* and *readsMask* as introduced above. We use `t.threadId` to get the thread id of `t`. We access a vector clock like an array of `int`s (with indices starting at 0) for clarity, but note that Java does not actually support operator overloading and so in JESS vector clocks are accessed and manipulated via method calls.

We first update `t`'s vector clock to be equal to the conceptual vector clock of the op. The first step is on line 8 where we set `t.VC[t.threadId]` to be the global clock of the

```
public static void updateVectorClocks(
    final ThreadData t,
    final int globalClockOfSyncOp) {

  final SyncObjectData o = t.currOpSyncObjectData;
  final boolean write = t.getCurrOp().getOpType().isWrite();

  t.VC[t.threadId] = globalClockOfSyncOp;

  if (write) {
    if(o.readsMask.isEmpty()) {
      t.VC.join(o.writeVC);
    } else {
      t.VC.join(o.readVC);
    }
    // Equivalent to: for all i: o.writeVC[i] = t.VC[i];
    o.writeVC.set(t.VC);

    o.lastWriteTid = t.threadId;
    o.readsMask.clear();
  } else {
    t.VC.join(o.writeVC);
    // The following loop is similar to o.readVC.join(t.VC)
    // but also updates o.readsMask as necessary.
    for(int i=0; i < o.readVC.length; ++i) {
      if(o.readsMask[i] && t.VC[i] >= o.readVC[i]) {
        o.readsMask[i] = false;
      }
      o.readVC[i] = t.VC[i];
    }
    o.readsMask[t.threadId] = true;
  }
}
```

Listing 5.11: The `updateVectorClocks` method.

op because we wish to ensure that an op at least always happens-after itself. However, `t`'s vector clock must be updated further. In Listing 5.11, we use `VC1.join(VC2)` to denote mutating vector clock `VC1` to become the pointwise maximum of `VC1` and `VC2` (i.e. *join*(*VC1*, *VC2*)). We use the join operation to conceptually add edges to the happens-before relation. Or, to put it another way, we use join to capture HBR edges in a particular vector clock.

Returning to Listing 5.11, if the op is a write, the current op happens-after all previous accesses of o (reads and writes). As explained earlier, $o.readVC$ and $o.writeVC$ captures the happens-before information for all previous reads and writes of o. As also noted earlier, if there have been no reads of o since the last write, then the op only races with the last write of o (the previous accesses of o happen-before the current op via transitivity). Thus, on line 10, we check if the op is a write and on line 11 we check if there have been no reads of o since the last write; if so, then we join `t`'s vector clock with $o.writeVC$ (line 12). Otherwise, there have been reads from o since the last write and so the op only races with these reads. Thus, we join `t`'s vector clock with $o.readVC$ (line 14); the reads since the most recent write are guaranteed to happen-after the most recent write, which is why there is no need to also join with $o.writeVC$. If the op is a read, then we join `t`'s vector clock with $o.writeVC$ (line 22), since a read happens-after the most recent write.

This covers how `t`'s vector clock is updated, but the vector clocks of `o` must also be updated. If the op is a write, then $o.writeVC$ is simply set to $t.VC$ (line 17) because $o.writeVC$ must contain the vector clock of the last write op, which is what $t.VC$ currently represents. We also update $o.lastWriteTid$ appropriately (line 19) and clear $o.readsMask$; that is, we set every element in $o.readsMask$ to be `false` indicating that there have been no reads since the last write to o. If the op is a read, then we set $o.readVC$ to $join(o.readVC, t.VC)$ (line 25) because $o.readVC$ must contain the pointwise maximum of all reads and the op that is being processed is a read; furthermore, $t.VC$ currently represents the vector clock of this read op. However, note that instead of calling `o.readVC.join`, we perform the join using a for-loop because we also need to update `o.readsMask`; on line 27, we change `o.readsMask[i]` to `false` if we are about to update `o.readVC[i]` to the same or a larger value. If `o.readsMask[i]` was not already `false`, then `o.readVC[i]` was a global clock of a read of o that occurred since the last write. However, updating `o.readVC[i]` to a larger or equal value indicates that the current read op happens-after the previous read. Note that if the new value of `o.readVC[i]` remains the same then `o.readVC[i]` still corresponds to the same read but this read now happens-before the current op and so is no longer relevant, as it cannot race with any future write to o. Finally, we set `o.readsMask[t.threadId]` to `true` since thread t just performed a read of o and `o.readVC[t.threadId]` is the global

clock this read (`globalClockOfSyncOp`).

```
1  public static void getRaces(
2      final ThreadData t,
3      final Set<Integer> raceClocks) {
4
5    final SyncObjectData o = t.currOpSyncObjectData;
6    final Op op = t.getCurrOp();
7    final boolean write = op.getOpType().isWrite();
8
9    // Write-read or write-write race.
10   // Next sync op of t may race with previous write to o.
11   if (o.lastWriteTid >= 0 && (!write || o.readsMask.isEmpty())) {
12     // Is last write concurrent with this thread?
13     if (o.writeVC[o.lastWriteTid] > t.VC[o.lastWriteTid]) {
14       raceClocks.add(o.writeVC[o.lastWriteTid]);
15     }
16   }
17   // Read-write race.
18   // Next sync op of t may race with one or more previous reads from o.
19   else if (write && !o.readsMask.isEmpty()) {
20
21     for (int i = 0; i < o.readVC.length; ++i) {
22       // Does o.readVC[i] correspond to a read from o
23       // and is it concurrent with this thread?
24       if (o.readsMask[i] && o.readVC[i] > t.VC[i]) {
25         raceClocks.add(o.readVC[i]);
26       }
27     }
28   } else {
29     // This is the first sync op on o. No races.
30   }
31 }
```

Listing 5.12: The `getRaces` method.

Get races The algorithm for calculating the set of races, I, is shown in Listing 5.12. The algorithm forms part of the DPOR algorithm (see Algorithm 5) where we find all ops that race with the *next* op of each thread. The `getRaces` method checks the next op of thread `t`. Note that the next op of `t` has not actually been executed yet and may not be the next op in the execution. Thus, although this algorithm is essentially a race detection algorithm, what we are really calculating here is the backtracking points needed by DPOR. Also, note that `t.VC` is not the vector clock of the next op of `t`, but is instead the vector clock of the most recent op executed by `t`. Nevertheless, we can use `t.VC` to check if various ops in the execution so far are *concurrent* with the last op of `t`; if there exist ops that are both concurrent with the last op of `t` and dependent with the next op

of `t`, then these ops would race with the next op of `t` if this op was executed next. The `write` variable is true iff the next op of thread `t` is a write; `o` is the sync object accessed.

On line 11, we check if there has been at least one write to `o` (`o.lastWriteTid >= 0`); we then check whether the next op of `t` is a read *or* whether there have been no reads from `o` since the last write. If so, the next op of `t` may only race with the previous write to `o`; this implies a write-read or a write-write race as described earlier. The global clock of the previous write to `o` is `o.writeVC[o.lastWriteTid]`; we check whether the previous write is concurrent with the last op of `t` using Definition 9 (line 13). That is, we check if the previous write does *not* happen-before the last op of `t`; if so, then the last write would race with the next op of `t` and so we add the clock of the last write to the set of races (line 14).

On line 19, we check if the next op of `t` is a write and if there has been at least one read of `o` since the last write (or since the start of the execution). If this condition holds, then the next op of `t` can only race with these reads; we consider each clock `o.readVC[i]` (line 24) but only if the clock corresponds to a read (i.e. if `o.readsMask[i]` is `true`). For each clock that *is* a read, we test whether the read is concurrent with the last op of `t` using Definition 9 and, if so, the read would race with the next op of `t` and so we add the read clock to the set of races (line 25).

5.4.4 Op class

We now reveal the fields of the `Op` class (which represents an op) and discuss several challenges that influenced the design. Note that we will obtain the hash of the HBR of an execution by hashing the set of ops in the execution (by XORing the hashes of the ops) and so equivalent HBRs should have the same hash. The fields of the `Op` class are shown in Listing 5.13.

Identity The first four fields are those described in our earlier definition of an op. However, note that the `tid` and `obj` fields are of type `Op`. We could have stored the `tid` as an `int` where the ith thread created has a tid of i. However, it is possible for two or more threads to be created concurrently (by two or more threads) and, thus, two different executions with the same HBR could create the threads in a different order; this could lead to different thread ids and different HBR hashes. Thus, we instead represent the thread id of a thread via the op that created the thread. Similarly, we represent the identity of a sync object via the op that created it. For example, creating a new Java object is an op with an `opType` of `OpType.OBJ_INIT` and with the `obj` field set to `null`; let us call this a

```
1  public final class Op {
2
3    private final Op tid;
4    private final Op obj;
5    private final OpType opType;
6    private final int pti;
7
8    private final int numWrites;
9
10   private final int objAddr;
11   private final int objOffset;
12
13   // methods omitted
14
15 }
```

Listing 5.13: The fields of the `Op` class, which represents an op.

creation op. An op that locks the object (with `opType OpType.ENTER_MONITOR`) will have its `obj` field set to the creation op. We describe below how fields and array elements are handled, using the `objOffset` field.

Capturing the HBR The HBR of an execution can be implicitly represented by the set of ops in the execution. To do this, we must be able to (conceptually) recreate the HBR from the set of ops. We described this briefly in §4.5 but we now give more detail. We use the approach given in [MQ07a]. Note that the total-order between ops from the same thread is captured by the `pti` field; given any two ops, op_1 and op_2, if $op_1.tid = op_2.tid$ and $op_1.pti > op_2.pti$, then op_1 happens-before op_2. The addition of the `numWrites` field allows us to capture the inter-thread orderings. Given an execution E containing an op $E(i)$, $E(i).numWrites$ is the number of ops in $E[1:i]$ that access $E(i).obj$ and are writes. Formally:

$$E(i).numWrites = |\{j \mid 1 \leq j \leq i \wedge E(j).obj = E(i).obj \wedge IsWrite(E(j))\}|$$

Given two ops, op_1 and op_2, that access the same object ($op_1.obj = op_2.obj$) then op_1 happens-before op_2 if: $op_1.numWrites < op_2.numWrites$, or $op_1.numWrites = op_2.numWrites$ and op_1 is a write. Thus, the addition of the `numWrites` field allows the HBR of an execution to be canonically represented as the set of ops in the execution, without any edges or vector clocks being stored. Note that we never actually reconstruct the happens-before relation from the set of ops. We use this observation to construct the *hash* of the HBR by XORing the hashes of the ops. We let the hash of an execution be

defined in this way. That is, $hash(E) = hash(E(1)) \oplus hash(E(2)) \oplus \ldots \oplus hash(E(|E|))$. Furthermore, as an execution grows, one op at a time, the next execution (and HBR) hash can be obtained incrementally by XORing the hash of the next op with the previous execution hash. In other words, given $E_1 \cdot \langle op \rangle = E_2$, then $hash(E_2) = hash(E_1) \oplus hash(op)$. Of course, hash collisions are possible, which will lead to incorrectly treating two distinct HBRs as being equal. In JESS, we use the MurmurHash3 128-bit hash function[7] from Google's Guava Java library[8] to reduce the chance of collisions.

Global objects Recall from §5.1.1 that we test a `ConcurrencyTestCase` object by repeatedly executing the `execute` method. The `execute` method may access objects that were created before the first execution of the `execute` method, such as objects created during the warm up executions. Thus, these sync objects will not have a creation op within any execution and so we cannot identify them via a creation op. To solve this, if a sync object is first accessed within an execution by a non-creation op then we assume this is a *global* sync object (one that was created outside of the execution) and so we set the `objAddr` field of the op to the identity hash code of the object (from applying `System.identityHashCode`). Note that these hash codes will not change across executions as the JVM is not restarted. The hash codes are not guaranteed to be unique, but collisions are unlikely and there is no alternative that guarantees uniqueness. Furthermore, we do not use the identity hash codes of objects created during executions so the chance of a collision is greatly reduced. Thus, the identify of an object is now the combination of the `obj` and `objAddr` fields. Note that the identity of an object is added (lazily) to the `SyncObjectData` object that it is mapped to, so the identity hash code only has to be calculated once.

Field and array offset We initially included (somewhat artificial) ops that represented the creation of every field and array element, so that these elements have a unique creation op. For example, in every constructor callback we added a write op (with `opType FIELD_INIT`) for every field in the class. Similarly, in an array creation callback we added a write op for every element in the array. We avoid doing this (for greater elegance and efficiency) using the `objOffset` field. We let an object creation op represent creating the object and all fields and an array creation op represent creating the array object and all array elements. When a field or array element is accessed, we let the op refer to the cre-

[7] `https://github.com/google/guava/blob/master/guava/src/com/google/common/hash/Murmur3_128HashFunction.java`

[8] `https://github.com/google/guava`

ation op (or the global address) of the object or array, but with the `objOffset` field set to the offset of the field or array element (both of which start at 1). Note that `ClassManager` (§5.2.2) ensures that every field in a class (and its super-classes) is assigned a unique `int` index which is used as the value for `objOffset` when accessing fields. Thus, the identity of a sync object is now the combination of the `obj`, `objAddr` and `objOffset` fields, where the `objOffset` field is 0 if we are accessing the object itself such as when entering (locking) a monitor. Therefore, every object, field and array element has a unique identity.

Note that our description of how to construct the HBR of an execution given the set of ops must change slightly; two ops only access the same object if the objects have the same identity (according to the `obj`, `objAddr` and `objOffset` fields), with one exception: the `objOffset` is ignored if one of the ops is a creation op because the creation op represents accessing *all* array elements of an array or fields of an object.

Execution id The final issue relates to global objects and the `SyncObjectData` class that is used to derive `Op` objects. Recall from §5.2.4 that each field is mapped to a `SyncObjectData` object stored in a shadow field. The `SyncObjectData` class includes fields `id`, `addr`, `offset` and `numWrites`, which store the values that will be used in op fields `obj`, `objAddr`, `objOffset` and `numWrites`, respectively. Consider that shadow fields of global objects persist across executions. In particular, the `numWrites` field on a `SyncObjectData` object, which stores the number of writes to the field so far, may be from a previous execution. Similarly, the state of the vector clocks may also be old. In contrast, the `SyncObjectData` objects for objects and array elements are stored in weak identity hash maps that can be cleared before each execution, and non-global objects will be recreated in each execution with uninitialised shadow fields. For global objects, we need a mechanism to detect shadow fields that contain data from a previous execution. To achieve this, we give the `ExecutionManager` an integer *execution id* that is incremented after each execution and we give the `SyncObjectData` class an `executionId` field. When a `SyncObjectData` object is initialised, its `executionId` field is set to the current execution id. On subsequent executions, if the `executionId` field of a `SyncObjectData` object does not match the current execution id then we reinitialise the `SyncObjectData` object.

5.4.5 Implementing barriers using read and write ops

Recall that every op is either a read or write. We now describe how we implemented a barrier using only read and write ops. Assume a barrier object for use with k threads. A barrier object supports only one operation, a *barrier wait* operation, that blocks the

calling thread until k threads in total are blocked waiting on the barrier object; thus, the kth thread that reaches the barrier is in fact not blocked and releases all other $k-1$ threads blocked at the barrier object. The kth thread also causes the barrier object to be reset so that it will block the next $k-1$ threads that try to wait on the barrier and the kth thread to wait will again not be blocked and will release the other threads, and so on. We refer to the set of k barrier wait operations and k threads involved in a single use of the barrier (where the kth wait operation resets the barrier) as a *round* of operations/threads. In terms of the HBR, a barrier wait operation executed by a thread happens-before all other barrier wait operations in this round. Notice that this implies a symmetric relation between barrier wait operations which is not possible in our asymmetric HBR; our HBR is a subset of a total-order over the ops, which means we cannot have cycles. Thus, we use two ops per barrier wait operation: a `BARRIER_PRE` op followed by a `BARRIER_POST` op. The `BARRIER_POST` op is blocking except for the kth thread in a round which does not get blocked, releases the other threads at the barrier and resets the barrier. We can now describe the HBR between these ops: a `BARRIER_PRE` op happens-before all `BARRIER_POST` ops in the same round. Crucially, `BARRIER_PRE` ops are unordered with each other, and `BARRIER_POST` ops are unordered with each other. This is important because the order in which a set of threads reach a barrier in an execution does not change the state reached by the execution. Thus, two executions in which threads reach the barrier in different orders but are otherwise identical should have the same HBRs.

Unfortunately, it is not possible to directly implement this description of the HBR using read and write ops; letting `BARRIER_PRE` be a write and/or `BARRIER_POST` be a write causes these ops types to be ordered with respect to each other, which is too strong of an ordering. Letting them both be reads causes all ops to be unordered, which is too weak. Thus, we introduce a `BARRIER_MID` op such that all threads in a round proceed as follows: all threads execute a *read* `BARRIER_PRE` op, then one thread will execute a *write* `BARRIER_MID` op, and then all threads will execute a *read* `BARRIER_POST` op. This achieves the ordering we require. However, it is not clear which thread should execute the `BARRIER_MID` op; it cannot be the last thread to reach the barrier as this will change depending on the order that threads reach the barrier (leading to different HBRs). It cannot be the thread that created the barrier as this thread may not be participating in the round. Thus, when a barrier is created, we start a *barrier thread* that blocks on a `BARRIER_MID` op and will be released once k threads have executed the `BARRIER_PRE` op in a round. This ensures that the `BARRIER_MID` op is always executed by the same thread and the HBR will be the same, regardless of the order in which threads reach the barrier.

5.5 Related work

Concuerror [GCS11, CGS13, AAJS14] is a systematic concurrency testing tool for Erlang programs. Implementation details of Concuerror are described in [CGS13]. Concuerror uses a source-to-source transformation (of the core Erlang language) to instrument the target program and libraries. Because Erlang uses actor-style programming, the implementation details are quite Erlang-specific, or at least specific to the actor-style. Thus, the contribution is mostly orthogonal to ours.

Several implementation details of the CHESS SCT tool are briefly described in [MQB+08, MM07], including how standard Win32 API functions are instrumented using a wrapper library and how blocking threads are detected without necessarily having to reimplement synchronising operations (as we do in JESS). Unlike in CHESS, in JESS we decouple the scheduling strategy (and its data structures) from the rest of the tool (see §5.3.4), such that the stack data structure used in a DFS need not exist when using a scheduling strategy that does not require it (such as the random scheduling strategy); we recommend that SCT tool authors follow this approach in the future for greater flexibility.

Java bytecode instrumentation We have already mentioned the TCH [FSS04] and method doubling [PSE07] approaches from prior work. Our approach is very similar to the method doubling of [PSE07], although that abstract has only high-level details; we describe some additional details, including handling of constructors, instrumenting methods of `java.lang.Object` using method body doubling and instrumenting thread creation which we have not seen in prior work. Unlike these prior works, we also add shadow fields and shadow arrays, similar to those used in RoadRuner [FF10] (discussed below), and discuss some of the issues in finding efficient ways of storing data. The source code for our method doubling approach is also available,[9] in contrast to the method doubling of [PSE07].

RoadRunner [FF10] is a dynamic analysis framework built using ASM for concurrent Java programs. In contrast to JESS, RoadRunner does not instrument standard libraries; it does not use method doubling or any similar approach. This was the main reason for not using RoadRunner in our work. Nevertheless, our shadow fields and shadow arrays were heavily inspired by the description in the RoadRunner paper [FF10].

[9] `https://github.com/mc-imperial/jtool-sct`

Vector clock operations The original DPOR algorithm included an implementation description using vector clocks [FG05], but this only supports write operations; thus, it uses one vector clock per thread and one "last write clock" per object. Prior work [SKH12] gives an improved version of DPOR that handles both read and write operations. Per object, the approach stores: one write vector clock, one access vector clock, one last write clock and a *list* of last read clocks (vs. our bitmask). As described in [SKH12], the list potentially causes bad worst case performance:

> A trivial worst case for finding a backtracking point is $O(|E|)$. This happens, for example, when a program consists of a single process [i.e. a single thread] executing multiple successive reads from a single communication object followed by a write to the same communication object

Note that the program could have many threads, but that a pattern that leads to particularly poor performance is when one thread performs many reads to the same object followed by a write, at which point all previous reads since the last write are checked using the happens-before relation. In contrast, our approach stores at most one read clock per thread (so at most n read clocks in total, where n is the number of threads); we track which of the read clocks in $o.readVC$ are worth considering in a bitmask which prevents unnecessary happens-before tests.

The CHESS tool [MQB+08] includes an implementation of DPOR that considers read and write ops. Note that this DPOR implementation is not described in the original project and was most likely added as part of later work [CBM10]. Thus, there is no published description of this implementation. From studying the source code,[10] we believe that, per sync object, the DPOR component stores one "last write" clock and one "accesses" vector clock. However, this is not sufficient to perform happens-before tests; the CHESS happens-before monitor component stores two additional vector clocks per sync object. Despite this inefficiency, the DPOR component stores at most one read clock per thread per sync object, which is similar to our approach. However, the "accesses" vector clock is an additional vector clock, whereas our approach uses an existing vector clock plus a bitmask. Furthermore, a read clock for thread i is never discarded due to reads from other threads (as in our approach), so some redundant read clocks may remain which could lead to redundant happens-before tests. Additionally, at the time of writing, we believe the CHESS DPOR source code contains a bug; when determining the backtracking points due to the next read op of a thread that accesses o, only the most recent access to o is considered. This access may turn out not to race and so no backtracking points will be added, but in this

[10] `http://chesstool.codeplex.com/SourceControl/latest#Chess/Dpor.cpp`

case, the other (less recent) accesses to o should then be considered, but they are not. This incorrect approach can lead to unsoundness due to missed backtracking points. We have received a confirmation from the author that the bug appears to be genuine [Coo16]. We hope that our description of vector clock operations and race detection will serve as an efficient and sound reference for future researchers.

In contrast to prior work, a key insight of our approach is that the clocks for previous reads and writes can always be found in $o.writeVC$ and $o.readVC$ which means no additional vector clocks or lists are needed. Furthermore, $o.lastWriteTid$ must be less than or equal to n and so can likely be stored as an 8- or 16-bit integer. The $o.readsMask$ is implemented as a bitmask which is also very space-efficient. Although this does not give a better big O space complexity compared to prior work (we still need $O(n)$ clocks per object), the improved space efficiency is arguably still useful in the context of race detection because of the potentially large number of sync objects (i.e. in the extreme case, every byte in the heap could be a sync object). We also believe that our approach is efficient in terms of run time complexity compared to prior approaches, as we track the minimal number of previous read clocks using a reads bitmask to identify the necessary clocks; some additional work is needed to maintain this bitmask but this work is performed (efficiently) during the update to the read vector clock and is justified since it prevents redundant happens-before tests from occurring later.

5.6 Conclusion

We have presented implementation details of JESS, our systematic concurrency testing tool for Java programs, as a reference for future researchers. We covered instrumenting Java programs using bytecode instrumentation, key design details of the tool, and some more advanced details relating to POR.

6 Case study: applying SCT to Azure Service Fabric distributed systems

In our empirical study (Chapter 3), we described some difficulties of applying SCT in practice (§3.4.2). In particular, programs that use network communication are problematic, as they require a large engineering effort to model the various networking functions/libraries and to handle inter-process communication. Additionally, ensuring that programs are deterministic (modulo scheduling nondeterminism) can be nontrivial. Motivated by these challenges and the importance of distributed systems, in this chapter, we consider applying SCT to distributed systems written for *Azure Service Fabric* [Fam15] (or *Fabric* for short), a platform and API for writing reliable services. On the one hand, distributed systems represent an extreme challenge for SCT due to the networked, inter-process communication. On the other hand, Fabric provides a stable C# API, which allows any modelling and engineering effort to be reused. We attack the challenge of ensuring determinism in this highly nondeterministic setting by using actors [HBS73, HO09, DDK+15]. We focus on actors that are restricted to basic send and receive operations; this allows SCT to be applied with minimal user effort, as actors only communicate via message-passing, which is mediated by the actor runtime. In short, we aim to allow users to write and test actor-based services for the Fabric platform so they can reap the benefits of SCT. To this end, we created a model of Fabric using actors that can be used to test actor-based Fabric services within a single process, removing inter-process communication entirely. The main results of this chapter are:

- A description of our Fabric model version 1, written using the P# actor framework.We describe its architecture and how replication (a key operation in Fabric) is achieved.
- A description of our Adara actors framework and the benefits it provides over other frameworks like P#. The main issue we found when using P# is that it uses dynamically-typed actors; it is not possible to statically determine the type of an actor reference and the messages that it can receive. Adara actors provides portable,

statically-typed actors that are defined by C# interfaces. This provides type safety and allows existing compiler/IDE features and static analyses to work as expected.

- A description of our Fabric model version 2 that uses Adara actors. We describe the key changes including how we were able to reduce the amount of asynchrony in our model (reducing its complexity) while still providing enough asynchrony to explore interesting interleavings of operations. We describe how replication is achieved to show the differences from version 1.

- An experimental evaluation showing that we can apply SCT to Fabric services and find bugs. We constructed a test system with 15 bugs that can be individually enabled: 11 real bugs that we found during development and 4 injected bugs that we believe are representative of subtle mistakes that developers are likely to make when writing Fabric services. We test for each bug using the controlled random scheduler (described in §3.2.4) and the PCT d=3 scheduler (described in §3.2.5). For each scheduler and for each bug, we execute 10,000 schedules. We found 14 of the 15 bugs using SCT, including all of the 4 injected bugs, showing that our Fabric model includes enough behaviours/asynchrony to expose these subtle pitfalls. We note that the controlled random scheduler performed poorly in comparison to its performance in our previous empirical study (§3.7) and we comment on the relative ease of applying SCT to actor-based systems compared to in Chapters 3 and 4.

We give an introduction to Azure Service Fabric (§6.1), an introduction to actor programming (§6.2), a description of our first Fabric model (§6.3), a description of our Adara actors framework (§6.4), a description of Fabric model V2 that uses Adara actors (§6.5), and a description of our experiments (§6.6). We conclude in §6.8.

Relation to published work Details of our first Fabric model were described in [DMT+16]. The P# framework was described and evaluated in [DDK+15].

6.1 Introduction to Azure Service Fabric

Azure Service Fabric [Fam15] (or Fabric for short) is a platform and API for writing services that are replicated for reliability. It is used by Microsoft for their own cloud services such as SQL Database, DocumentDb, Bing Cortana, Halo Online, Skype for Business, and many others [Fam15]. Our focus in this work is the lowest-level C# API provided by Fabric which is not yet publicly documented. To use Fabric, the developer

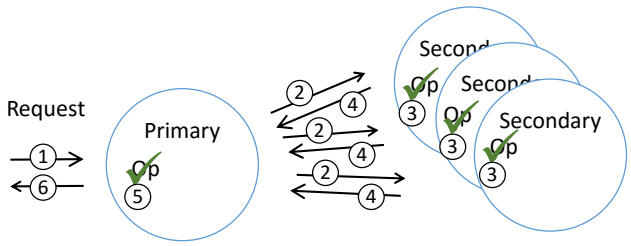


Figure 6.1: A diagram showing replication in Fabric.

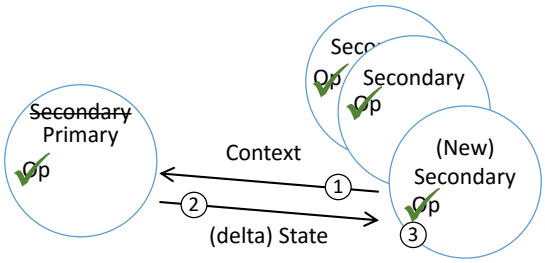


Figure 6.2: A diagram showing state copying in Fabric.

writes a C# service using the Fabric API that receives requests (e.g. HTTP requests) from clients and mutates its state based on these requests. To make the service reliable, Fabric launches several replicas of the service, where each replica is a separate process that typically runs on a different node. One replica is selected to be the *primary* which serves client requests; the remaining replicas are *secondaries* that redundantly store the same data as the primary and can optionally be used to serve read-only client requests. Although the Fabric API helps developers in writing reliable services, there are still many subtle cases that must be handled by the developer to ensure that state is correctly replicated between replicas. The nondeterminism due to the asynchronous and distributed nature of services further increases the likelihood of subtle concurrency bugs being introduced. Thus, applying SCT to Fabric services would be extremely useful for finding and reproducing such bugs in a deterministic manner.

We give a brief overview of two key Fabric processes, replication and copying, in order to give an impression of the complexity of Fabric and introduce some terminology. As a running example, consider a shopping list service that simply stores a list of strings. Clients can request the list of items from the primary or secondary replicas (i.e. read the list) or request to add a string to the list at the primary replica.

Figure 6.1 shows the replication process in Fabric: (1) The primary receives a request from a client that changes the state of the replica. For example, in the shopping list service the request would be to add an item to the shopping list. The primary does not

yet change its state but instead creates an object that captures the operation that will mutate the state of the replica. We refer to this object as the *replication operation*. (2) The primary informs Fabric (via an API call) of the replication operation, which Fabric then forwards to all secondaries. (3) Each secondary updates its state based on the replication operation. Note that the code that updates the state of the secondary is written by the developer. For example, in the shopping list service a secondary will add an item to its shopping list. (4) Each secondary informs Fabric that its state has been updated. Fabric acknowledges the replication operation with the primary. (5) Once at least half of the secondaries have acknowledged the replication operation, Fabric invokes a callback (written by the developer) at the primary. The callback applies the replication operation to the primary's state and (6) acknowledges to the client that the request completed. Note that some secondaries may not have updated their state by the time the client request is acknowledged. If a secondary falls too far behind it can "catch up" by receiving a copy of the state from the primary. A particular instance of this state copying occurs when a primary fails, which we now describe.

When the primary fails, Fabric elects one of the secondaries to become the new primary and spawns a new secondary. The promoted secondary is informed by Fabric that it is now the primary. The new secondary initially has no state and so must catch up to the state of the primary. Fabric services that maintain state can optionally write state to a persistent data store, such as to disk or an external database server. Thus, note that the new secondary may be able to restore state from its persistent store, although this state may be slightly out-of-date. Figure 6.2 shows the state copying process in Fabric. (1) The new secondary sends its *copy context* data; that is, the secondary sends messages to the primary indicating what state it has managed to restore from its persistent store. (2) In the simple case, the new secondary has no state or is slightly behind the primary. The primary sends a copy of its state so that (3) the secondary updates its state. However, the primary need not send its full state. Based on the copy context data it received from the secondary, the primary can instead send the delta-state—messages that allow the secondary to mutate its restored state into the primary's current state. Note that various subtle issues can arise here. For example, a secondary may have updated its state (and written this state to its persistent store) due to replication operations that ended up not being applied at the primary. The new secondary may restore this state and the new primary must detect this "false-progress" from the copy context data and act appropriately; the simple solution is for the primary to send a full copy of its state to the secondary.

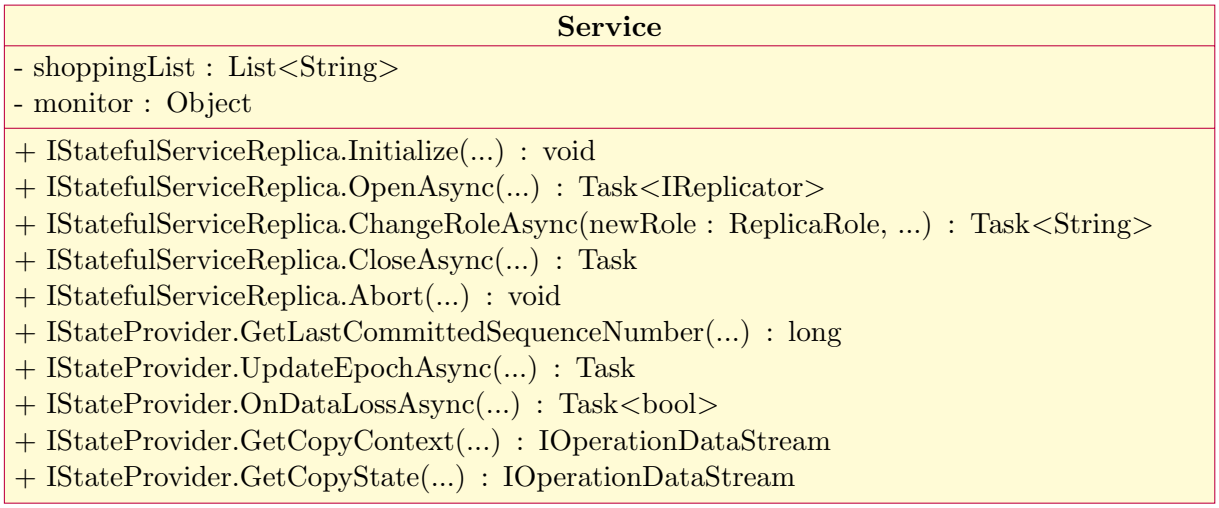


Figure 6.3: A class diagram of our C# Fabric shopping list service class. Many fields and methods are omitted so as to focus only on the key details. We use "..." to indicate omitted method parameters

6.1.1 The Fabric API

The Fabric API that we model for this work is publicly available, but not publicly documented, and has only been used internally at Microsoft. The public release of Azure Service Fabric[1] includes and documents two higher-level APIs that are built on top of the low-level API. Furthermore, there is limited code available that uses the low-level Fabric API. As such, we created our own straightforward shopping list service in C# that uses the low-level Fabric API. This allowed us to understand the Fabric API and provide an example of how to use it for future reference. We describe some elements of the Fabric API using our shopping list service as an example.

The class diagram of the shopping list `Service` class is shown in Figure 6.3. The class stores the shopping list in its `shoppingList` field. The object in the `monitor` field is used as a monitor to protect accesses to the shopping list, as certain methods and callbacks can be invoked by Fabric concurrently. A Fabric service has to provide implementations of the `IStatefulServiceReplica` and `IStateProvider` interfaces; our `Service` class implements both of the interfaces, although in general each interface could be implemented by a different class. In each replica process, Fabric creates an instance of the `Service` class and then invokes the `Initialize` method, followed by `OpenAsync` and then `ChangeRoleAsync`, where the `newRole` parameter will reveal whether this replica will be a primary or sec-

[1] `https://azure.microsoft.com/services/service-fabric/`

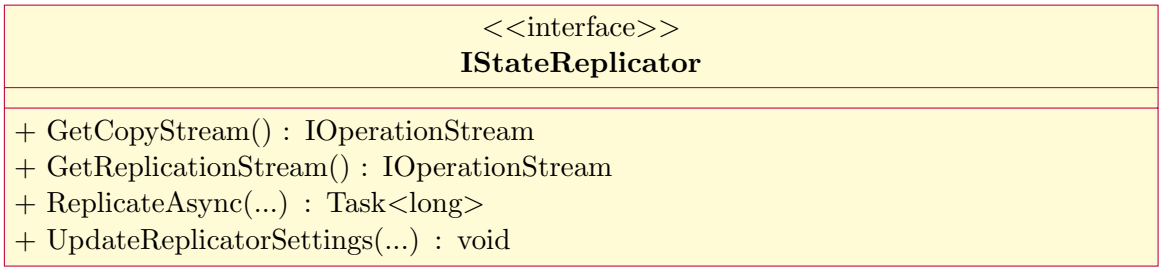


Figure 6.4: A class diagram of Fabric's `IStateReplicator` interface.

ondary. The `IStateProvider` interface methods will be invoked to get information about the state (i.e. the shopping list) of the replica. For example, `GetCopyContext` will be called on a new secondary in order to get the copy context information, as described earlier. The `GetCopyState` method will be called on the primary to get the state that will be sent to a new secondary (which is referred to as the copy state). These methods return and take *streams* (e.g. `IOperationDataStream`), which are objects on which Fabric or the service code repeatedly invokes `GetNextAsync` to get the next operation. Each operation is simply an array of bytes. Thus, service code must serialise/deserialise its copy context or copy state data in order to send/receive it to/from other replicas. Replication operations are handled in the same way.

The file structure of Fabric's `IStateReplicator` interface is shown in Figure 6.4. Fabric provides an `IStateReplicator` object to the service. The key methods are `ReplicateAsync`, which allows the primary to send replication operations to the secondaries, and `GetCopyStream`/`GetReplicationStream`, which allows secondaries to get the copy/replication stream on which to receive copy/replication operations from the primary.

6.2 Actor programming using P#

Our aim is to apply SCT to Fabric services. In order to apply SCT in this highly nondeterministic setting, we take advantage of actor-style programming [HBS73, Akk, HO09, DDK+15, Pon]. A key advantage of this approach is that actors only communicate via message-passing, which is mediated by the actor runtime; this allows SCT to be applied efficiently and with minimal user effort because we can simply provide an alternative actor runtime that controls when messages are sent and received. This is in contrast to Chapters 3 and 4, where we had to handle arbitrary synchronisation operations using complex and expensive instrumentation of the target program. To introduce actor-style programming, we describe the P# framework [DDK+15], which was used to implement

the initial version of our Fabric model. We note that (non-actor) C# programs typically use C# *tasks*[2], which are asynchronous operations that are scheduled on a thread pool. Actor-style programming is an alternative approach which we assume to be incompatible with task-based programming.

P# [DDK+15] is a framework for asynchronous, actor-style programming co-designed with SCT. It takes advantage of the message-passing approach to apply SCT with minimal user effort. A P# program consists of a set of actors that communicate by sending messages to each other. Note that the program is a single process; the actors are scheduled on a thread pool. Each actor has private fields (i.e. private state) and a FIFO queue in which received messages are placed. An actor processes its messages, one-at-a-time, by removing the message from the front of its queue. For each removed message, the appropriate *action* for the given message type is invoked. An action is just a sequential C# method; an action should not create C# threads/tasks, use C# synchronisation primitives, nor perform inter-actor communication via other means such as via shared memory accesses. Actions can create additional actors and send messages to other actors. Each P# actor conceptually has a *state* field; the mapping between message types and actions for an actor can change depending on the value of its state field. However, in this work, we focus on actors with a single state and thus a fixed mapping between message types and actions.

We can describe a P# program in terms of our abstract model of a concurrent program (§2.2): actors are analogous to threads, private fields are part of each actor's thread state, and each actor's FIFO queue is part of the shared state. The visible operations are: *create* (to create an actor), *start* (which is the first operation executed by every actor and thus this operation is disabled until the actor has been created), *end* (which is the last operation executed by every actor before the actor halts—the actor then blocks forever on a second *end* operation—see §2.2), *send* (which is a non-blocking operation that adds a given message to a given actor's queue), and *receive* (which blocks the calling actor until/unless the calling actor's queue contains at least one message at which point the first (oldest) message is removed and returned for processing). Note that details of action handlers and state fields are abstracted away as they are captured by transitions and thread states. Note that there is no *join* operation to wait for an actor to terminate. In Adara actors, which we introduce in §6.4, we add a *join* operation for convenience.

[2] `https://msdn.microsoft.com/en-us/library/dd537609(v=vs.110).aspx`

```
1  public class MHuman : Machine
2  {
3      public class EEat : Event
4      {
5          public string foodName;
6          public int nourishmentAmount;
7
8          public EEat(string foodName, int nourishmentAmount)
9          {
10             this.foodName = foodName;
11             this.nourishmentAmount = nourishmentAmount;
12         }
13     }
14
15     private int health = 100;
16
17     [Start]
18     [OnEventDoAction(typeof(EEat), nameof(OnEat))]
19     class SingleState : MachineState { }
20
21     void OnEat()
22     {
23         EEat e = (EEat) this.ReceivedEvent;
24         health += e.nourishmentAmount;
25         Console.WriteLine(
26             $"Just ate {e.foodName} and gained {e.nourishmentAmount} health.");
27     }
28 }
```

Listing 6.1: The definition of a P# `MHuman` actor.

Example P# actor An example of a P# actor is shown in Listing 6.1.[3] Note that in P#, actors are called *machines* and messages are called *events*, but we continue to use the terms actors and messages. As shown: an actor is a class that derives from P#'s `Machine` class (line 1); a message is a class that derives from P#'s `Event` class (line 3), which need not be nested as in this example (although we use this convention for messages that are unique to a particular actor); a field is an ordinary C# private field (line 15); a state is a nested class that derives from P#'s `State` class (line 19); and an action is a C# method with `void` return type (line 21). A mapping between a message type and an action is defined using a C# attribute (which is like a Java annotation) as seen on line 18. Thus, when this actor receives an `EEat` message, the `OnEat` action will be invoked, which in this case just updates the private field of the actor and prints a message for debugging. To create an instance of the `MHuman` actor and send it an `EEat` message, another actor can

[3] P# provides a domain specific language for writing P# programs, which is transpiled to C#. However, we did not use this in our work and instead write programs directly in C# as shown, using P# as a library.

execute the following code from inside an action:

```
1  MachineId m = CreateMachine(typeof(MHuman));
2  Send(m, new EEat("Pizza", 2));
```

The `CreateMachine` and `Send` methods are defined by P#'s `Machine` class.

P# execution A P# program can be executed on P#'s production runtime or SCT runtime. The production runtime uses C# tasks to schedule actors to execute on a thread pool. Thus, note that an actor need not be implemented as a thread but it is possible to think of an actor as a thread when considering the semantics of a P# program. The SCT runtime performs SCT, letting only one actor execute at a time in order to explore the different interleavings. The SCT runtime supports various schedulers including a straightforward depth-first search, the controlled random scheduler, and a version of the PCT scheduler; we described these schedulers in our empirical study in §3.2.

6.3 P# Fabric model

In this section, we describe our P# Fabric model. Recall that our aim is to apply SCT to distributed systems but that trying to handle arbitrary network communication functions and inter-process communication is extremely challenging. Thus, we chose to target Fabric services specifically by implementing a model of Fabric and testing the services against this model within a single process. Our initial aim was to produce a P# Fabric model on which we could run unmodified C# Fabric services (i.e. the services would not be written using P#) using P#'s production runtime. This would give us confidence that our model is accurate enough to run Fabric services but we would not be able to perform SCT because we cannot control task-based C# code. We would then be able to port services to P# so that SCT can be applied.

We describe our approach (§6.3.1), the architecture (§6.3.2), an example of how replication occurs in our model (§6.3.3), how we created a test harness for running Fabric services on our model (§6.3.4), and how we were able to prototype P# Fabric services on our model to perform SCT (§6.3.5).

6.3.1 Approach

A key design choice that we made upfront was that we should *not* try to model all internal details of Fabric, and we should under- and over-approximate in our model where reasonable, with the goal of allowing interesting interleavings and without introducing

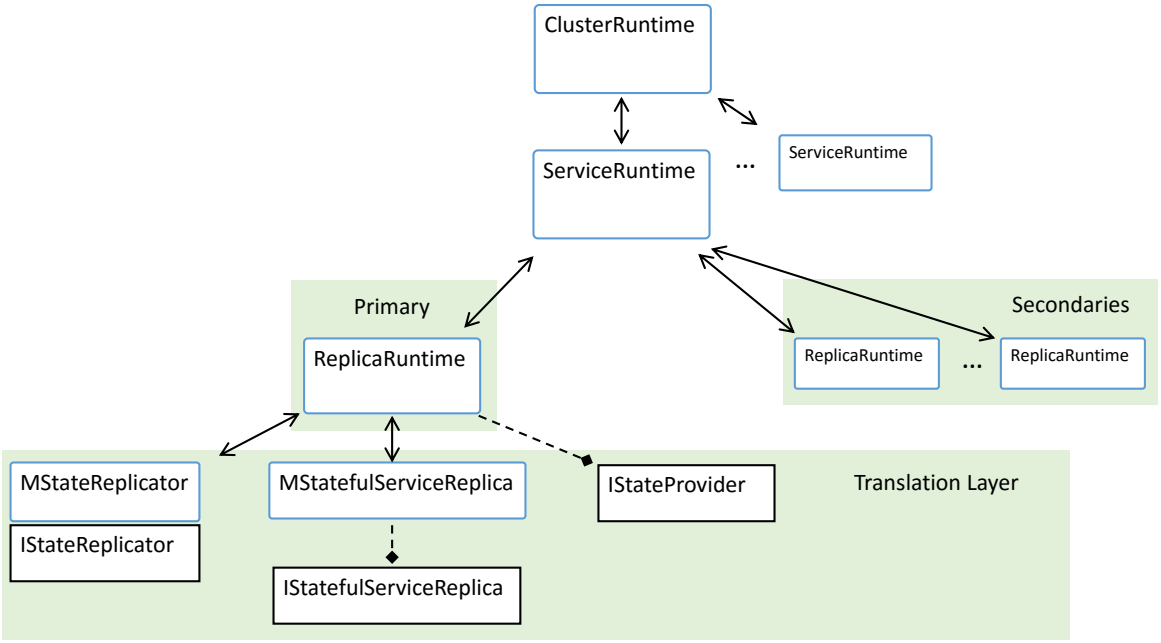


Figure 6.5: A diagram showing the design of the P# Fabric model. The rounded rectangles with a blue outline represent P# actors, while the rectangles with a black outline are C# classes that implement the interface given by the associated label. The solid lines indicate communication via messages. The dashed lines show composition, e.g. a `ReplicaRuntime` actor invokes methods on its `IStateProvider` object. Note that an `MStateReplicator` actor and its `IStateReplicator` object interact via shared-memory communication.

false-errors. For example, Fabric uses a distributed leader election algorithm to pick which secondary to promote. Implementing this in the model would greatly increase the schedule-space of the system without any benefit. Indeed, this algorithm is not even described in the Fabric documentation as it is not relevant to the service code. Instead, our model picks the first viable secondary. We believe this is sufficient to expose interesting interleavings. Nevertheless, we could change our model to nondeterministically pick one of the viable secondaries if we later find that more nondeterminism is needed to expose bugs. Regardless, we do not model the election algorithm; our goal was to model the *observable behaviours* of the Fabric API so that we can expose interesting interleavings of the service code.

6.3.2 Architecture

Figure 6.5 shows the design of our Fabric model. We use the term *real Fabric* to refer to the current implementation of Fabric. In the real Fabric, multiple services can be running and communicating with each other; in our model, the `ClusterRuntime` actor manages all Fabric services that exist in our system and allows for the creation of new services. A `ServiceRuntime` actor is created for each instance of a service that is running. In the real Fabric, each service is made up of multiple replica processes that typically run on different nodes. Furthermore, there is no central "manager" process like the `ServiceRuntime` actor. This is another example of how our model does not need to follow the design of the real Fabric, as these details do not affect the execution of the service code that we want to test. We use the `ServiceRuntime` actor for convenience to store information about the service and to manage all replicas that are part of the service. Additionally, the replicas all communicate with each other via the `ServiceRuntime` actor (unlike in the real Fabric). Each replica process is represented as a `ReplicaRuntime` actor, where each service has one primary replica and a configurable number of secondary replicas. Figure 6.5 only shows the actors and objects managed by a primary `ReplicaRuntime` actor, but the secondary `ReplicaRuntime` actors are identical. We refer to the actors and objects shown below the `ReplicaRuntime` as the *translation layer*, as these actors and objects are only used to allow the C# service code to interact seamlessly with the P# `ReplicaRuntime` actor, without modification.

The `IStateProvider` and `IStatefulServiceReplica` objects are provided by the Fabric service code, as described in §6.1.1. The `ReplicaRuntime` creates a new instance of the `IStatefulServiceReplica` implementation and creates an `MStateServiceReplica` actor that wraps the object. The `ReplicaRuntime` thus makes "calls" on the `IStatefulServiceReplica` object indirectly by sending and receiving messages (i.e. asynchronously). The reason for this approach is that we needed the `ReplicaRuntime` actor to be able to process other messages while invoking methods of `IStatefulServiceReplica`. For example, when invoking `ChangeRoleAsync` on the `IStatefulServiceReplica` object (but before the call has returned), we still want the `ReplicaRuntime` to be able to send and receive messages relating to replication or copying, as this may reveal subtle bugs in the service code. If the `ReplicaRuntime` actor made the call directly then the actor would be blocked until the call returns. In contrast, the methods of the `IStateProvider` interface are less complex and so the `ReplicaRuntime` invokes these methods directly, for simplicity. Recall that the `IStateReplicator` object is provided to the service code by Fabric. Thus, our model provides its own implementation of this interface. At the

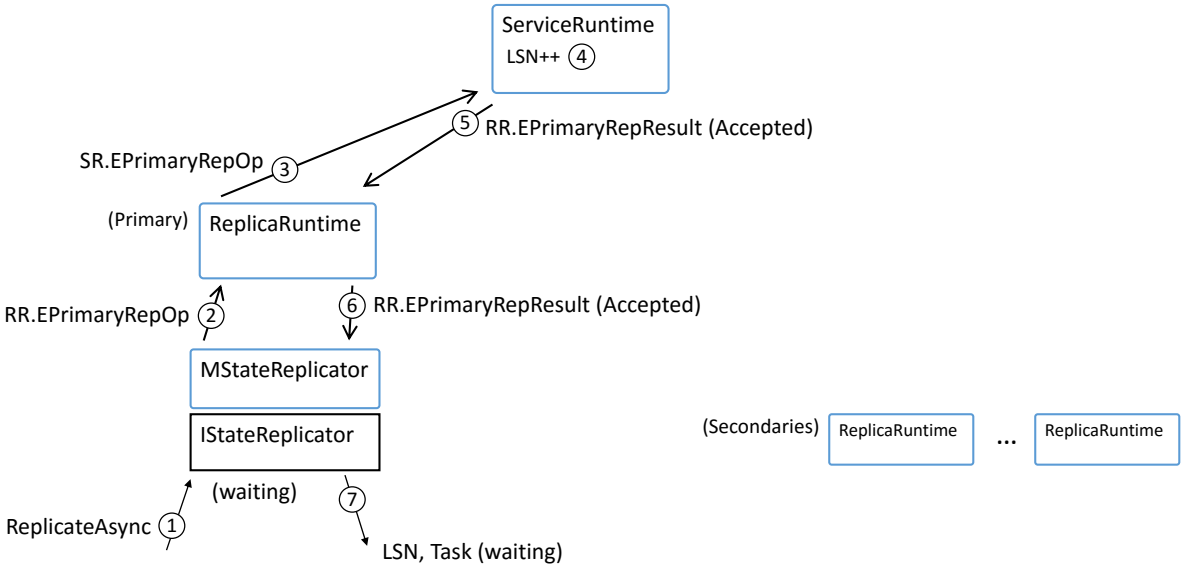


Figure 6.6: A diagram showing the first stage of replication in our Fabric model. The left `ReplicaRuntime` is the primary replica.

time of creating the model, P#did not support waiting to receive a P# message from an arbitrary task/thread. This posed a problem, as the service code can invoke methods of `IStateReplicator` from some task/thread that it has created, yet we wanted to be able to send a message to `ReplicaRuntime` and wait for a response. To solve this, we use an `MStateReplicator` actor to wait for messages via inter-task synchronisation; this violates the rule that actions should not synchronise with other tasks (as explained in §6.2), as the actions of `MStateReplicator` use synchronisation to notify other tasks about the message. However, the main consequence of breaking this rule (in general) is that we cannot perform SCT on the code, which is already the case, since we are interfacing with C# task-based code. We addressed this issue in version 2 of our Fabric model (§6.5).

6.3.3 Replication example

Having introduced the architecture of our model, we now show how we modelled replication (introduced in §6.1) which is a key process in Fabric. As as result, we also introduce more Fabric terminology.

State 1: getting the logical sequence number (LSN) for the replication operation Figure 6.6 shows the first stage of replication in our model. We assume the primary

has already received a client request and now wants to send the replication operation to the secondaries so they can update their state. The C# service code at the primary invokes `ReplicateAsync` (1) on the `IStateReplicator` object; our implementation of this method sends an `RR.EPrimaryRepOp` message (2) to the `ReplicaRuntime` actor, which includes the serialised replication operation data. For example, in our shopping list service, the operation data is an instance of an add operation class that contains a string that will be added to the shopping list. The `ReplicateAsync` method then blocks until it receives a message, using the `MStateReplicator` actor.

In the `ServiceRuntime` actor, we store the current *logical sequence number* (LSN). In the Fabric API, each replication operation is assigned a consecutive integer LSN, such that all operations have a well-defined total-order. We speculate that the real Fabric stores this in the primary replica process and secondaries must also mirror this in case the primary goes down. In our model, we chose to store the current LSN in the `ServiceRuntime` actor along with a buffer that stores all replication operations that have not yet been acknowledged by a majority of secondaries. Our intention was to simplify our model; by keeping complex logic in the `ServiceRuntime` actor along with the required data, we reduced the communication needed between the `ServiceRuntime` actor and `ReplicaRuntime` actors. The replication operation is given the current LSN. The current LSN is then incremented (4). The `ServiceRuntime` actor then sends an `RR.EPrimaryRepResult` message to the `ReplicaRuntime` actor (5) that includes the LSN and indicates that the replication operation has been accepted (but has not yet been acknowledged by a majority of secondaries). This is forwarded by the `ReplicaRuntime` actor (6). When the `MStateReplicator` actor receives the message, the call to `ReplicateAsync` returns the LSN assigned to the operation as well as a task that is not yet complete (7); the task will complete later, when the `MStateReplicator` actor receives a second `RR.EPrimaryRepResult` message indicating that the replication operation has been acknowledged by a majority of secondaries.

Stage 2: applying the replication operation at secondaries Figure 6.7 shows the second stage of replication in our model and we now detail the steps of this process. (1) The `ServiceRuntime` actor is still executing the action for the `EPrimaryRepOp` message. The action sends k `ESendReplicate` messages to the `ServiceRuntime` actor, where k is the number of secondaries; the action for this message sends an `ESecondaryRepOp` message to one of the secondaries. We chose to have the `ServiceRuntime` actor send messages to itself so that the sending of messages to the secondaries is split up across multiple actions and so can be "interrupted" by other messages. In particular, we model primary failure (i.e. simulating the primary's node crashing, or becoming unresponsive due to network

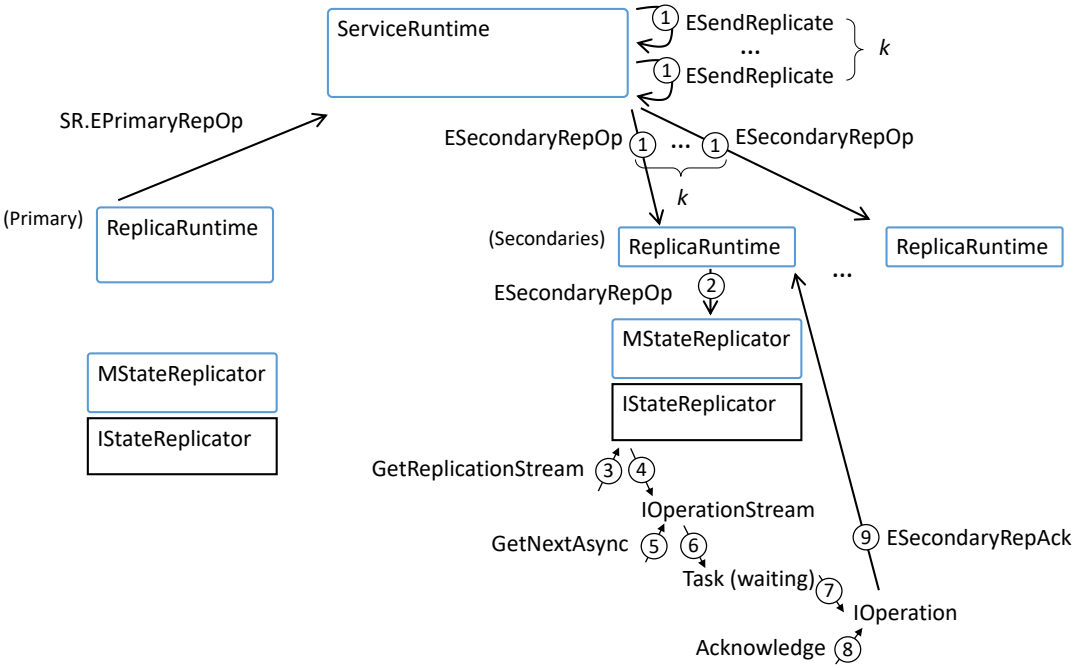


Figure 6.7: A diagram showing the second stage of replication in our Fabric model.

failure) by sending an `EKillPrimary` message to the `ServiceRuntime` actor; thus, the primary can be "killed" at a point where only *some* of the secondaries have received the replication operation. Figure 6.7 shows one secondary `ReplicaRuntime` actor in detail. Note that earlier, the service code of the secondary invoked `GetReplicationStream` (3) on its `IStateReplicator` object, which returns a stream (4). The service code starts a task to get the messages from the stream by calling `GetNextAsync` (5), which yields a task (6) that only completes once the `MStateReplicator` actor receives an `ESecondaryRepOp` message (2). Now that the message has been received, the task yields an `IOperation` object (7), containing the serialised operation data. The secondary service code deserialises the operation and applies it. For example, for the shopping list service, the service code adds the deserialised string to its shopping list. Once the operation is applied, the service code invokes `Acknowledge` (8) on the `IOperation` object, which we implemented to send an `ESecondaryRepAck` to the `ReplicaRuntime` actor (9).

Stage 3: acknowledging the replication operation Figure 6.8 shows the third stage of replication in our model. The secondary `ReplicaRuntime` actor forwards the `ESecondaryRepAck` message to the `ServiceRuntime` actor (1). In the `ServiceRuntime` actor, we store the set of acknowledged replication LSNs for each `ReplicaRuntime` actor. (2)

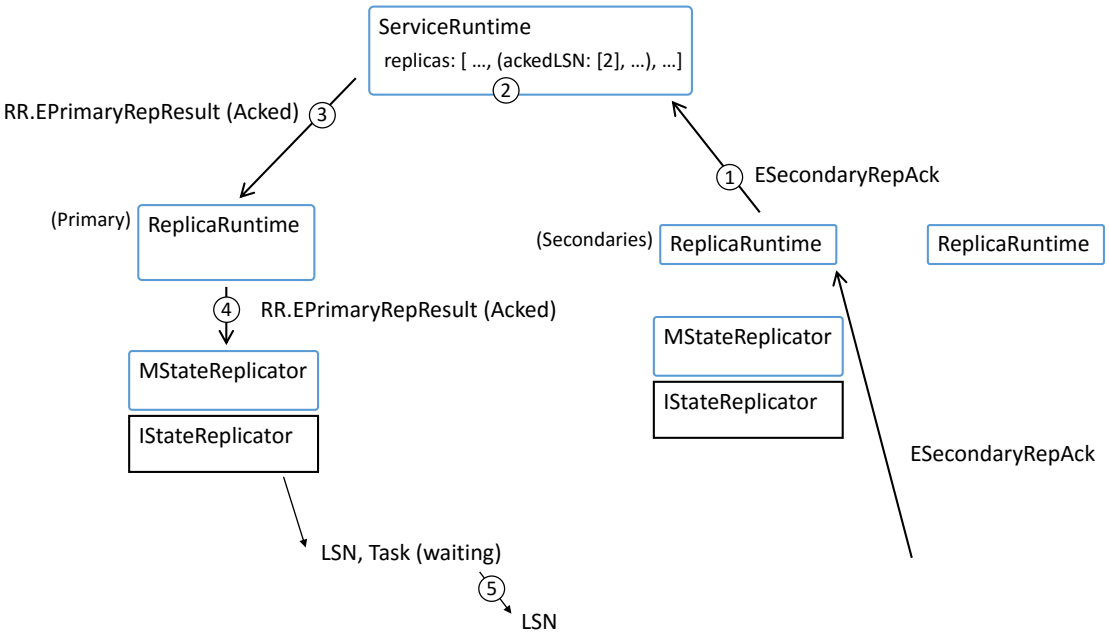


Figure 6.8: A diagram showing the third stage of replication in our Fabric model.

The acknowledgement message from the secondary causes the set of acknowledged LSNs for the secondary to be updated. Once a majority of secondaries have acknowledged the replication operation, the `ServiceRuntime` actor sends the second `RR.EPrimaryRepResult` (3) message to the primary `ReplicaRuntime`, indicating that the replication operation has been acknowledged. This message is forwarded to the `MStateReplicator` actor (4) and the task that was previously returned to the service code completes (5), yielding (for convenience) the LSN of the acknowledged replication operation. The Fabric documentation notes that, at the primary, replication operations are not necessarily acknowledged in order. Our model captures this scenario since the `ESecondaryRepAck` messages (for each replication operation) can be sent in any order.

6.3.4 Test harness

To execute a C# service on our Fabric model using the P# production runtime, we used a test harness similar to the one shown in Listing 6.2. On line 5, we create the `ClusterRuntime` actor, yielding the id of the actor. On line 7, we create the service runtime parameters object that defines various parameters for our Fabric service, such as how many secondary replicas there should be. In particular, we set the `CSharpFactory` parameter to a factory object that will be used to create a shopping list service object for each `ReplicaRuntime` actor. On line 14, we send an `EAddServiceRuntime` message to

```
1  class Program
2  {
3    static void Main(string[] args)
4    {
5      MachineId id = PSharpRuntime.CreateMachine(typeof (ClusterRuntime));
6
7      var serviceRuntimeParams = new ServiceRuntimeParams
8      {
9        // ...
10       CSharpFactory = new ShoppingListServiceFactory()
11     };
12
13     var e = new ClusterRuntime.EAddServiceRuntime(serviceRuntimeParams);
14     PSharpRuntime.SendEvent(id, e);
15
16     var eKill = new ClusterRuntime.EKillPrimary(serviceRuntimeParams.ServiceName);
17     PSharpRuntime.SendEvent(id, eKill);
18
19     Console.WriteLine("[Press enter to exit]");
20     Console.ReadLine();
21   }
22 }
```

Listing 6.2: Test harness for executing our C# shopping list service on our Fabric model.

the `ClusterRuntime` actor. This message will cause a `ServiceRuntime` actor and several `ReplicaRuntime` actors to be created, thus creating several replicas of our shopping list service. On line 17, we send an `EKillPrimary` message to the `ClusterRuntime` actor, which will cause the primary replica of our shopping list service to fail; this allows us to test secondary promotion and the state copying process. The test harness does not check that a new primary was elected and that all state was copied successfully. However, our model *does* include many assertions to check expected properties which can be used to reveal bugs in our model and the service. We describe how we created a more thorough test harness for SCT that checks the state of all replicas in §6.6.1.

6.3.5 P# services

Our long-term goal was to allow Fabric services to be written in P# to be executed on the real Fabric and also on which we could perform SCT using our Fabric model. In this work, we do not address the creation of a P# Fabric API nor the challenge of running P# services on the real Fabric. However, we were able to *prototype* running P# services on our model by porting our shopping list service to P#. Intuitively, we replaced the translation layer in Figure 6.5 with a single `MShoppingList` actor. We modified the `ReplicaRuntime` actor so that, when hosting a P# service, calls to `IStateProvider` are replaced with sending and

receiving messages. In other words, our initial attempt at a P# Fabric API was essentially made up of the message types used for communication between the `ReplicaRuntime` and translation layer. Thus, we were able to perform SCT on our service; this revealed many subtle bugs in our Fabric model that had remained hidden when testing C# services without SCT. Indeed, the numerous subtle bugs we encountered in our model was a key motivation for creating version 2 of our Fabric model (§6.5) with an improved design. We defer discussion of bugs to §6.6.3 where we perform SCT on our Fabric model V2.

6.4 Adara actors

In this section, we describe our *Adara actors* framework, which provides statically-typed actors for C#. While developing the P# Fabric model, described in §6.3, we identified several shortcomings of using P# in practice. Our solution was to create Adara actors. We then created the Fabric model V2 (which we will describe in §6.5) using Adara actors. Note that we created Adara actors from scratch (without using P#). The Adara actors framework provides the following benefits:

- Statically-typed actors using C# interfaces; we discuss the benefits of statically-typed actors below (§6.4.2).
- Actors defined using Adara actors depend only on a small set of C# interfaces and so are decoupled from the underlying runtime. Thus, they are portable as they could, in theory, be executed on other actor frameworks, such as P#, Akka.NET, etc. by implementing an alternative runtime that defers to the target framework.
- Better integration with C#'s task-based concurrency; Adara actors can treat any task as an actor and wait on these tasks (as is common in task-based code). SCT can be performed on these task-based actors.

Note that the approach used to implement statically-typed actors is not limited to C#; it would be possible to implement the approach in other languages that support interfaces, such as Java. In this section, we describe some of the issues of using P# that motivated the creation of Adara actors (§6.4.1), we describe the Adara actors framework using examples and discuss its advantages (§6.4.2), and describe the code generation used in Adara actors (§6.4.3).

6.4.1 Motivation

The main issue we encountered when using P# is that it uses dynamically-typed actors. That is, it is not possible to statically determine the type of an actor reference (a `MachineId` object) and the messages that it can receive. This can make the code less readable (as it can be unclear what kind of actor an actor reference refers to) and can lead to run-time errors that could be detected at compile-time (when a message is sent to an actor that cannot handle the message); C# developers are used to the benefits of static-typing provided by C# classes and interfaces. Also, defining actors requires several, tedious manual steps and conventions that are not understood by existing tools, compilers and IDEs. This can hurt developer productivity; again, C# developers are used to having strong tool-support, such as being able to automatically generate and refactor code.

Consider the following statements that create an `MHuman` actor and send it an `EEat` message:

```
1  MachineId m = CreateMachine(typeof(MHuman));
2  Send(m, new EEat("Pizza", 2));
```

The `MachineId` object stored in variable `m` is not specialised to denote that the actor is of type `MHuman`. Thus, the compiler and IDE cannot provide any assistance or static checks. For example, there is no way to know what messages can be sent to `m`. There is no way to discover the "interface" of `m` nor use IDE features to autocomplete messages or jump to the `MHuman` class. On line 2, any object can be passed as the second parameter to the `Send` method; there is no compile-time check that an `EEat` message can be received by `m`. Also consider that `MachineId` objects can be sent to other actors; in such scenarios, it can be very difficult to keep track of which messages an actor can receive.

Consider, once again, the definition of the `MHuman` actor in Listing 6.1. Notice that in order to add a new message and corresponding action, a new nested class must be defined (with fields and possibly a constructor), one or more mappings must be added, and an action must be added. Although the example only has one message type (`EEat`) and so may seem relatively straightforward, in practice we found that prototyping additional messages was tedious due to the required manual steps; the IDE provides little assistance since it is not aware of this workflow. Also, notice that in the `OnEat` action (§21), the event that triggered the action is obtained via the `this.ReceivedEvent` property and must be cast (unsafely) to the `EEat` type (line 23). This is another manual, unsafe step. Also note that, even if the `MachineId` class included `MHuman` as a generic type parameter, the mappings are specified in the `MHuman` class using C# attributes, which, again, are not understood by existing tools and so provide no information to the compiler and IDE.

```
public interface IHuman : ITypedActor
{
    void Eat(string foodName, int nourishmentAmount);
    void Run(int distanceInMeters);
}
```

Listing 6.3: The `IHuman` actor interface.

```
public class Human : IHuman
{
    private int health = 100;

    public void Eat(string foodName, int nourishmentAmount)
    {
        health += nourishmentAmount;
        Console.WriteLine(
            $"Just ate {foodName} and gained {nourishmentAmount} health.");
    }

    // ...
}
```

Listing 6.4: The `Human` class implementation of the `IHuman` actor.

A further potential issue is that P# code is highly-dependent on the P# library and runtime. As such, P# actors are not portable. The effort that we have spent in modelling Fabric might be useful in the future in contexts where P# is not appropriate or not needed. Thus, it would be ideal if we could write our Fabric model using portable actors that are independent of the P# runtime, and so could be run on other C# actor frameworks in the future (e.g. P#, Akka.NET [Akk], etc.) without changes.

6.4.2 Adara actors

Our solution to these issues is *Adara actors*—a library for defining portable, statically-typed actors using C# interfaces. We now explain how actors are defined using an example. Listing 6.3 shows how we define an `IHuman` actor interface. Actor interfaces implement the `ITypedActor` interface (which is an empty interface) to distinguish them from normal interfaces. The method signatures in `IHuman` define the message types that it can receive; in this case, it can receive `Eat` and `Run` messages. The method parameters define the fields in the respective message. Thus, the `Eat` message has fields `foodName` and `nourishmentAmount`, and the `Run` message has a `distanceInMeters` field. Listing 6.4 shows the `Human` class that implements `IHuman`. This class provides an implementation of the `IHuman` actor interface and contains the private state (i.e. the `health` field) and

the actions (i.e. the methods) for the actor. Listing 6.5 shows how to create an `IHuman` actor and send it a message. On line 2, an `IHuman` actor is created, where the private

```
1 // A new instance of Human provides the handlers and private state of the actor.
2 IHuman humanProxy = typedRuntime.Create<IHuman>(new Human());
3 // Send messages to the IHuman actor by invoking methods.
4 humanProxy.Eat("Pizza", 2);
```

Listing 6.5: Creating a typed actor in Adara actors and sending a message.

state and actions of the actor are given by a new instance of the `Human` class. However, the returned `IHuman` object is *not* the instance of `Human`. Instead, it is an *actor proxy* object that allows messages to be sent to the new actor by simply invoking methods of the proxy object. Thus, on line 4, the call to `humanProxy.Eat` is not directly invoking the `Human.Eat` method; instead, a message is sent to the `IHuman` actor containing the method name ("Eat") and the parameter values ("Pizza" and 2) and the call immediately returns, regardless of when the message is processed. Thus, it is equivalent to `Send` in P#. When the message is processed by the `IHuman` actor, the Adara runtime invokes the `Human.Eat` method on the `Human` object (created on line 2), passing the parameter values ("Pizza" and 2) from the message.

We note the following advantages of Adara's statically-typed actors (or just *typed actors* for short) over P#'s dynamically-typed actors (*untyped actors* for short):

- Statically-typed actors are type-safe. This improves code readability as it is always possible to determine the interface of an actor reference (actor proxy) and, thus, the messages that can be sent to the actor. It is only possible to a send a message to a typed actor if the typed actor has defined an action for that message type. This eliminates run-time errors caused by unhandled messages. Additionally, the receiving actor does not need to cast the incoming message like in P#.
- We immediately have IDE and compiler integration due to the use of interfaces. For example:
  - Method names and parameters are auto-completed and invoking a non-existent method gives an IDE/compiler error.
  - It is fast and straightforward to define messages because they are generated from method signatures. IDE features, such as refactoring and code generation, all work as expected. For example, action stubs (i.e. empty actions) can be automatically generated (because empty interface method implementations can be generated by the IDE).

```
1  public interface IA  : ITypedActor { void A(); }
2  public interface IB  : ITypedActor { void B(); }
3  public interface IAB : IA, IB {}
4
5  public class AB : IAB
6  {
7      public void A() {}
8      public void B() {}
9  }
10
11 public class Main
12 {
13     static void Main()
14     {
15         // ...
16         IAB ab = typedRuntime.Create<IAB>(new AB());
17         IA a = (IA) ab;
18         // ...
19     }
20 }
```

Listing 6.6: An example of hiding message types in Adara actors by exploiting interfaces.

  - It is easy to find all actions for a given message type (i.e. method) as this simply requires searching for all implementing methods which is supported by IDEs.

- Describing and understanding the interface of an actor simply involves looking at the C# interface, which is familiar and well-understood. Furthermore, the benefits of normal C# interfaces apply; an actor can implement multiple actor interfaces and an actor proxy object can be cast to one of the implementing interfaces in order to "hide" methods of the other interfaces, simplifying the interface of the proxy object. This casted proxy object can be sent to other actors and these actors will only be able to send messages that are part of the casted interface. For example, consider Listing 6.6. The `ab` proxy object is cast to an `IA` object (line 17), which hides the `B` method, and this casted proxy object could be sent to another actor that expects an `IA` actor (as opposed to an `IAB` actor).

Architecture and portability An additional benefit of Adara actors is that actors are portable; actor code (such as our Fabric model V2 (§6.5)) only depends on a set of interfaces, and not our implementations of these interfaces. The assembly dependency diagram for Adara actors and the Fabric model V2 is shown in Figure 6.9. An assembly is the smallest unit of deployment for .NET applications and typically contains a set of precompiled classes/interfaces.

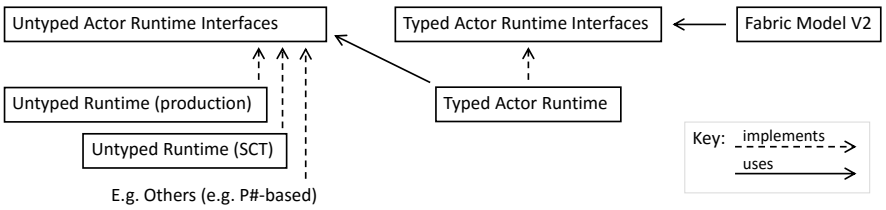


Figure 6.9: The assembly dependency diagram for our Adara actors framework and our Fabric model V2.

As shown in Figure 6.9, Adara actors consists of untyped actor runtime interfaces (which provide the ability to create dynamically-typed actors) and typed actor runtime interfaces (which provide the ability to created statically-typed actors). Notice that the Fabric model V2 does not depend on our typed actor runtime nor any of our untyped runtimes; instead, it depends on the typed actor runtime interfaces assembly. Similarly, our typed actor runtime requires an untyped actor runtime that provides basic, dynamically-typed actors and can be implemented in any appropriate way. We provide two untyped runtimes (like in P#): a production runtime and an SCT runtime. Note that these do not use P#; they were implemented from scratch using C# tasks. In theory, one could also implement the untyped actor runtime interface using P# or another untyped actors library. In order to run our Fabric model using specific typed and untyped runtimes, we need another assembly (not shown) that depends on the Fabric model and the concrete runtimes (e.g. our production untyped runtime and our typed actor runtime) and creates the initial actors from the Fabric model using the concrete runtimes.

6.4.3 Code generation for actor proxies

We now describe the code generation that occurs for actor proxies by considering our earlier example. The idea of automatically generating proxy objects is not new; such objects are generated in remote procedure call frameworks like Java's Remote Method Invocation (RMI) [Ora], the Windows Communication Foundation (WCF) [Mic], and Apache Thrift [Apa]. Our approach uses run-time code generation via the `System.Reflection.Emit` package to emit bytecode, but compile-time code generation could be used instead.

Consider Listing 6.5 once again. When the `Create` method is called (line 2) for the first time for a given actor interface (e.g. `IHuman`), the typed actor runtime generates a single proxy class and potentially several message classes. For example, for the `IHuman` class, the runtime generates a proxy class similar to that shown in Listing 6.7. The proxy

```
1  public class HumanProxy : IHuman
2  {
3      public IMailbox<object> mailbox;
4
5      #region Implementation of IHuman
6      public void Eat(string foodName, int nourishmentAmount)
7      {
8          var msg = new EatMsg
9          {
10             foodName = foodName,
11             nourishmentAmount = nourishmentAmount
12         };
13         mailbox.Send(msg);
14     }
15
16     // ...
17
18     #endregion
19 }
```

Listing 6.7: The generated `HumanProxy` class.

```
1  public class EatMsg : ICallable
2  {
3      public string foodName;
4      public int nourishmentAmount;
5
6      // Overrides ICallable.Call
7      public void Call(ITypyedActor t)
8      {
9          ((IHuman) t).Eat(foodName, nourishmentAmount);
10     }
11 }
```

Listing 6.8: The generated `EatMsg` class.

contains a field for the mailbox of the target actor, which is initialised by the typed actor runtime to refer to an untyped actor that we will describe below. A mailbox is like a `MachineId` in P# in that it is used to send untyped messages to an actor. We show the implementation of the `Eat` method on line 6; the method stores the method parameters in a new `EatMsg` object and sends the object to the target actor. The `EatMsg` class is one of the generated message classes; the runtime generates one such class for each method in the actor interface (e.g. each method in `IHuman`). For the `IHuman.Eat` method, the typed actor runtime generates an `EatMsg` message class similar to that shown in Listing 6.8. For each parameter of the `Eat` method there is a corresponding field in the `EatMsg` class. The `EatMsg.Call` method is used to invoke the `Eat` method on a given object that implements

`IHuman`, passing in the field values as the parameters.

Once the proxy and message classes have been generated, the typed actor runtime creates an untyped actor and sends it an object that implements the actor interface. For example, in Listing 6.5, the runtime sends the instance of the `Human` class (which implements the `IHuman` actor interface). The untyped actor then simply invokes `ICallable.Call` on every received message, passing in the object (e.g. the `Human` object). Once the untyped actor has been sent the object that implements the actor interface, a new instance of the proxy class (e.g. `HumanProxy`) that sends messages the untyped actor is returned (line 2 of Listing 6.5).

6.5 Fabric model V2

In this section, we describe our Fabric model V2, which uses an alternative design and Adara actors (§6.4). We encountered several issues with our first Fabric model (§6.3). First, the shortcomings of using P# described in §6.4.1 made development difficult. Second, the mixture of tasks and P# actors in our translation layer was inelegant. In Adara actors, we ensured that our untyped actors runtime could support treating tasks as actors, allowing them to send and receive messages without resorting to shared-memory synchronisation. Third, we encountered many subtle bugs in our original model. We decided that creating a `ReplicaRuntime` actor for each replica led to an unnecessarily complex model. We also learned more about Fabric while developing the model and we felt that several other aspects of our model would benefit from a different design. We describe several aspects of the new design below, followed by an example of replication (§6.5.1).

Removal of `ReplicaRuntime` actors and shared-memory communication Figure 6.10 shows the design of our Fabric model V2. Unlike in the original (Figure 6.5), the `IServiceRuntime` actor (similar to the `ServiceRuntime` actor in the original) directly communicates with wrapper actors for each replica. In other words, there are no `ReplicaRuntime` actors. The `IServiceRuntime` actor contains a list of `ReplicaInfo` objects that stores all information about the replicas. This information was previously split between the `ServiceRuntime` and `ReplicaRuntime` actors. This removes a large amount of complexity from the model. However, communication with the replicas is still asynchronous due to the wrapper actors: `IStatefulServiceReplicaWrapper` and `IStateProviderWrapper`. Also, our implementation of `IStateReplicator` (which we provide to the service code) now communicates with the `IServiceRuntime` actor via messages instead of using shared-memory communication; this is enabled by our Adara actors

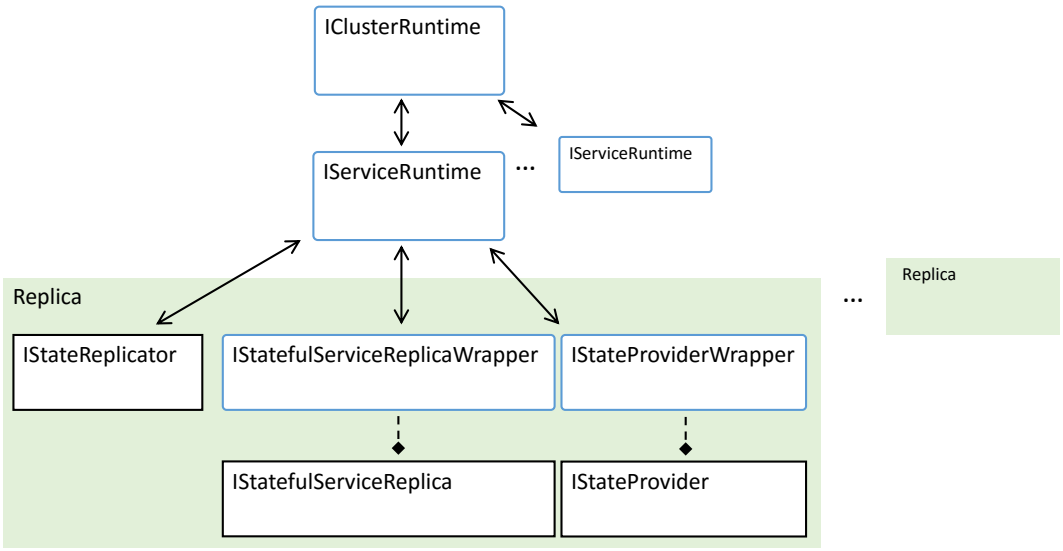


Figure 6.10: A diagram showing the design of our Fabric model V2. The rounded rectangles with a blue outline represent actors, while the rectangles with a black outline are C# objects that implement the interface given by the associated label. The solid lines indicate communication via messages. The dashed lines show composition (e.g. the `IStateProviderWrapper` actor invokes methods on its `IStateProvider` object).

untyped actor runtime being able to obtain a mailbox for the current task on which messages can be received.

Primary and secondaries In our first Fabric model, we created several `ReplicaRuntime` actors at once in order to maintain a set of replicas for a service. These were then changed into primaries or secondaries, state copying occurred, etc. concurrently. By running and modifying our shopping list service on the real Fabric, we later realised that Fabric will first try to establish a single primary replica before creating/updating the secondaries. Furthermore, if the primary changes before a secondary has become an active secondary, the idle secondary will be destroyed and restarted. These behaviours were not described in the documentation (although this is not surprising, as the documentation was minimal) but we believe it is safe to rely on them in the future. This leads to a simpler system with less asynchrony; all secondaries can be ignored until the primary replica is established, at which point all secondaries can be handled concurrently. This benefits us in two ways: (1) our model is simpler and less likely to have bugs; (2) SCT is more efficient as the size of the schedule-space is reduced.

Request-response buffers In the real Fabric, when replicas perform state copying and replication they request a message from a stream (by calling `GetNextAsync`), which can yield different results depending on the state of the replica set. In our original Fabric model, we implemented streams using the provided messaging system of P#. Thus, replicas just waited for a message, instead of requesting a message. In version 2 of our model, replicas instead send a message to the `IServiceRuntime` actor *requesting* a replication or copy operation from the stream and the `IServiceRuntime` actor responds with a replication or copy operation message (or a *closed* message). This allows us to implement the precise behaviour of streams. We noticed a recurring pattern in these scenarios which led us to create *request-response* buffers. A request-response buffer contains two buffers: one for the the requests for an operation and one for the responses (the operation data messages). Requests can arrive before responses and vice versa. For example, a secondary may request a replication operation before a primary has sent one or the primary may send one or more replication operations before a secondary has requested one. When adding to the request-response buffer, the newly added request or response may match with a buffered response or request, respectively. A function object is passed to the `Add` method that will be executed when a request and response are matched. There are also cases where multiple request-response pairs exist in the buffer but cannot be processed and removed immediately; we give of an example of this when we describe replication (§6.5.1). As such, requests and responses continue to be buffered until the `ProcessMatching` method is invoked, at which point all request-response pairs are processed using the provided function object. The function objects can return different values to indicate whether the request, response, both, or neither, should be removed from the buffers. For example, in our function objects, a null response is never removed as it indicates the end of the stream; all future requests from the stream are removed and cause the null operation to be sent to the replica, as in the real Fabric. We believe request-response buffers may be a useful data structure in general for actor systems.

6.5.1 Replication version 2

We now describe replication in our Fabric model V2 to contrast it with replication in our original Fabric model (§6.3.3). Figure 6.11 shows the first and second stages of replication in our Fabric model V2. As before, we assume that the primary replica has received a client request that it has to replicate to the secondaries. The service code at the primary invokes `ReplicateAsync` (1) on the `IStateReplicator` object, providing the replication operation, which sends a `PrimaryRepOp` message (2) containing the replication opera-

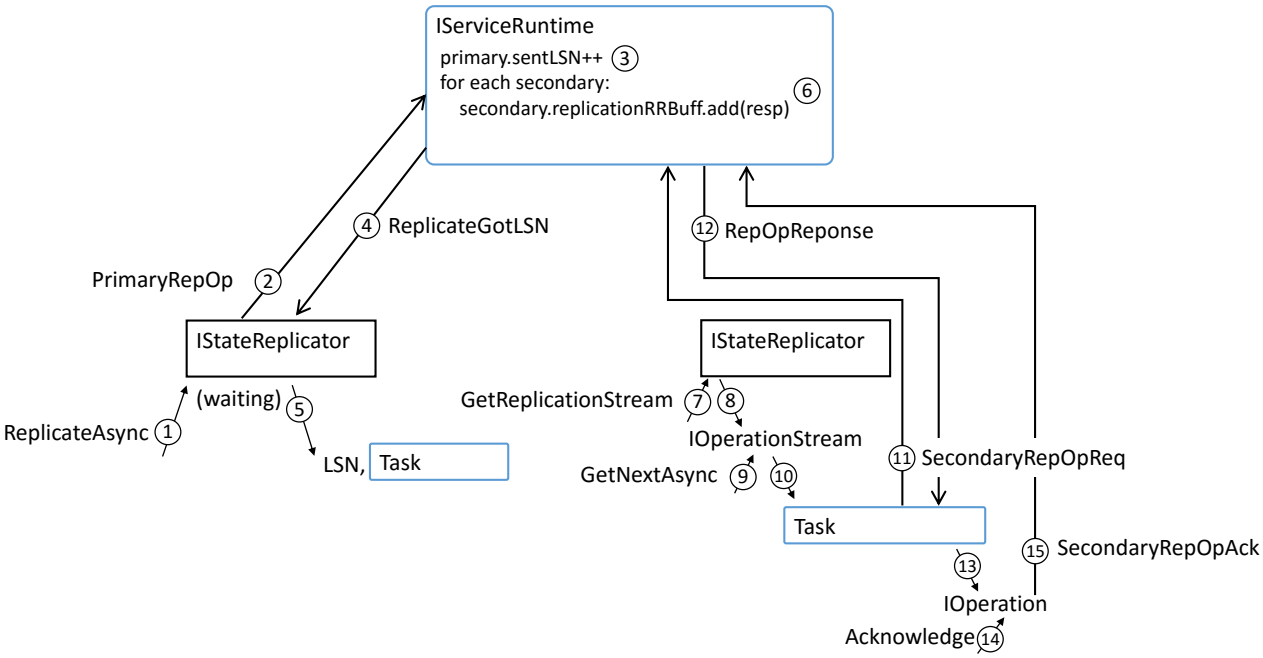


Figure 6.11: A diagram showing stage one and two of replication in version 2 of our Fabric model. The left `IStateReplicator` is the primary replica.

tion to the `IServiceRuntime` actor. The `IServiceRuntime` actor no longer contains a single `currentLSN` field. Instead, the current LSN is stored in the `sentLSN` field of the `ReplicaInfo` object that corresponds the primary replica. We believe this more accurately represents how the real Fabric stores LSNs; if the primary fails, the current LSN may be lost with it. Similar to before, the `IServiceRuntime` actor increments the primary's LSN and responds with a `ReplicateGotLSN` message (4) which contains the new LSN for the replication operation. As before, the `ReplicateAsync` method returns (5) the LSN and a task. We show the task as an actor to highlight the fact that Adara actors allows tasks to behave like an actor by waiting for messages. The action at the `IServiceRuntime` is still executing; the replication operations are added as a *response* to the replication request-response buffer (`replicationRRBuff`) of each secondary (6). Recall that a request that is added to the buffer will be matched with this response and the request-response pair will be processed. Alternatively, a request for a replication operation may already be in the buffer. In this example, we assume the former. Note that the `IServiceRuntime` actor no longer needs to send messages to itself; this approach was used in the old model so that we could achieve the scenario where the primary fails and only a subset of secondaries had received the replication operation. Since replication operations are now requested via messages, this scenario can occur in a schedule where some secondaries have not yet made

a request for the replication operation.

As before, the service code at a secondary replica will acquire the replication stream (7–8) and invoke `GetNextAsync` (9) to get a task (10) that will yield the next replication operation. This time, our implementation of the task sends a `SecondaryRepOpReq` message (11) to the `IServiceRuntime`. This request will be matched with the response in the secondary's replication request-response buffer. As hinted earlier, the request-response pair may not always be processed and removed immediately. For example, if the primary has changed since the last successful replication operation, the secondary must be notified about the change *before* receiving the next replication operation; in this case, the match function sends a message to the secondary's `IStateProviderWrapper` actor and leaves the request-response pair in the buffer to be processed later. In this example, we assume the pair can be processed immediately; the match function sends a `RepOpResponse` message (12) and the pair is removed from the buffer. This message allows the task to complete and yield an `IOperation` (13). The secondary deserialises and applies the operation to update its state and then invokes `Acknowledge` (14) which we implement to send a `SecondaryRepOpAck` message to the `IServiceRuntime` actor. Once a majority of secondaries have acknowledged the replication operation, the `IServiceRuntime` sends a message to the primary to acknowledge the operation; this process is very similar in the original Fabric model (§6.3.3) and so we omit a description and diagram.

6.6 Experiments

We aim to show that we can apply SCT to distributed systems written for the Fabric platform. Thus, we aim to show that our actor-based Fabric model is complete enough to find bugs in Fabric services that use the actor-style approach. We also wish to briefly contrast the effort needed to apply SCT when using actors with the effort needed in Chapter 3 and Chapter 4, where we used C/C++ and Java benchmarks, respectively. To this end, we created a test system consisting of: our Fabric model V2, our shopping list service modified to use Adara actors, a client actor that sends requests to the shopping list service, and a test harness that initialises the above and waits for quiescence (i.e. for all actors to block because there are no more messages to process). Our test system contains 15 bugs: 11 real bugs that were found using SCT,[4] and 4 injected bugs based on errors that we thought developers would be likely to make when writing Fabric services. We guarded each bug/fix with its own boolean variable so that we can enable each bug

[4]One real bug was actually found via inspection while fixing another bug (found via SCT) and has not yet been reproduced using SCT.

in isolation. We test each bug using the controlled random scheduler (§3.2.4) and the PCT d=3 scheduler (§3.2.5). Recall that these schedulers both use randomisation and were used in the empirical study in Chapter 3. In particular, recall that PCT d=3 uses a priority based scheduler with 2 randomly chosen priority change points and that this scheduler found the most bugs at various different schedule limits (§3.7.2). We execute each scheduler on each bug for 10,000 schedules. We report the number of buggy schedules detected as well as other data. We use this approach to answer the following research questions (RQs):

RQ1 Can we find bugs in our test system using SCT, thus showing that we can apply SCT to actor-style distributed systems written for the Azure Service Fabric platform?

RQ2 Does our Fabric model include enough asynchrony/nondeterminism to expose the injected bugs in our system? (We believe the injected bugs highlight several tricky and subtle aspects of developing Fabric services that developers are likely to get wrong at some point.)

RQ3 How do the controlled random scheduler and PCT d=3 scheduler compare in terms of number of bugs found, and how do these results compare to our results of our empirical study in Chapter 3?

RQ4 How easy was it to apply SCT to this actor-based system compared to applying SCT to our C/C++ benchmarks in Chapter 3 and our Java benchmarks in Chapter 4?

We describe our test system (§6.6.1), our scheduler implementations (§6.6.2), our results (§6.6.3), and our main findings (§6.6.4).

6.6.1 Test system

As described above, our test system consists of our Fabric model V2, our shopping list service modified to use Adara actors, a client actor that sends requests to the shopping list service, and a test harness (a method) that we repeatedly execute. We now describe our shopping list service, how we ported the service to Adara actors, our client, and our test harness.

Shopping list service Our shopping list service stores a list of strings (items). Clients can request the list of items from the primary or secondary replicas (i.e. read the list), or request to add a string to the list at the primary replica (i.e. modify the list). A request to add an item is replicated to the secondary replicas before being applied at the primary and

confirmed with the client. Although the service sounds simple, Fabric services are subtly complex; developers must implement services very carefully, following the pseudocode in the Fabric API documentation, in order to ensure that data is copied and replicated correctly. For example, the primary replica must be very careful in tracking and applying its replication operations as they may be acknowledged out-of-order (but typically must be applied in order).

Porting our service to Adara actors Our shopping list service uses C# tasks and a monitor (i.e. a mutex with signalling capabilities) to perform operations asynchronously as suggested by the Fabric documentation. We could have reimplemented our service using a single actor (without a mutex). This would greatly reduce the concurrency in the system and perhaps simplify the implementation of services at the expense of performance. However, we wanted to capture the complexity and concurrency of Fabric services as described in the Fabric documentation as accurately as possible, so we decided to make minimal changes to the service. Thus, we changed the service to create and wait for tasks using the untyped actors runtime so that these tasks can be treated as actors and controlled during SCT. We also implemented a monitor actor that allows actors to lock, unlock, wait and pulse the monitor (notify waiting actors) by sending and receiving messages. Although the actors share objects, the ownership of the shared objects is conceptually transferred via the messages being sent to and received from the monitor actor. Thus, there are no data races. After these changes, we were able to run the service (deterministically) on the Adara actors SCT runtime. As stated earlier, we leave the creation of an elegant, actor-based Fabric API for future work.

Client We created a client actor that attempts to add three unique items to the shopping list service. In our Fabric model, we modelled parts of the Fabric client API which allows client code to query a Fabric service by name and retrieve a list of endpoints (URLs) for the replicas of that service. Each endpoint is marked as referring to either the primary replica or one of the secondary replicas. Additionally, we added extra methods to the Fabric model so that actors can add and retrieve mappings from URLs to actor mailboxes (to model network communication via sending messages). Thus, each shopping list replica: (1) creates an actor that "listens" for client requests; (2) adds a mapping between some chosen URL and the listening actor's mailbox; (3) informs Fabric of the URL so that clients can retrieve it. The client actor: (1) uses the Fabric client API to query the shopping list service by name, receiving a URL for each replica; (2) looks up the mapping for the primary URL to get a mailbox to which it can send messages.

Our client actor repeatedly tries to resolve (i.e. get the mailbox of) the primary's listening actor; this can fail because the primary has not yet initialised or because the primary has been killed. Once the primary is resolved, the client sends three add requests to the primary in order to add the three unique items to the shopping list. The client keeps track of pending requests and, on receiving a response, re-resolves the primary replica, requests the current state of the shopping list, and re-sends add requests if necessary. Note that any add request could fail if the primary is killed at the right moment and the item may or may not have been replicated to the secondaries. Our client can also request the shopping list from every replica and assert that the shopping lists are equal and contain only the three unique items (with no duplicates).

Test harness Our test harness creates an instance of the shopping list service configured to use three replicas (i.e. one primary and two secondaries). It also immediately creates an actor that sends a "kill primary" message so that the primary will be killed at some point in the schedule. The client actor is also created. The test harness then waits for quiescence (i.e. for all actors to block because there are no more messages to process). It then tells the client to check that all replicas have the expected shopping list. Note that the client will stop sending messages once it sees that the primary's shopping list contains the three items; from this point, it is not possible for the shopping list to be lost unless two replicas are killed. Thus, the test harness should always see that all replicas have the expected shopping list. The test harness can therefore detect bugs in the client, service or Fabric model that prevent this expected final state from being reached, such as incorrect handling of replication/copy operations or secondary promotion. The client, service and Fabric model also all contain additional assertions to check various expected properties. We found bugs from both categories; i.e. some bugs cause more immediate assertion failures, while others only cause the final state assertion to fail.

6.6.2 Random schedulers

We implemented the controlled random scheduler (§3.2.4) and the PCT scheduler (§3.2.5) for our Adara actors SCT runtime. Note that we treat actors analogously to threads and so our previous descriptions of these schedulers still apply, although we used a modified version of the PCT scheduler. We now describe our modified version of the PCT scheduler, how we applied POR, how we handled yielding and liveness issues, and how we generated our random seeds.

Modified PCT algorithm We use a modified version of the PCT scheduler (different to the version described in §3.2.5) that does not require an upfront estimated number of steps k nor an estimated number of threads/actors n. This is useful as our system creates many, short-lived actors and so the number of actors created can vary e.g. depending on when the primary is killed. Our modified PCT algorithm proceeds as follows, where k is initially 1:

1. Insert the initial actor into a list L. The total-order of the actors in L yields the relative actor priorities; the first actor in the list has the highest priority while the last actor in the list has the lowest priority.

2. Randomly pick integers $k_1, \ldots, k_{d-1}$ from $\{1, \ldots, k\}$. These will be priority change points. Thus, initially, $k_1 = k_2 = \ldots = k_d = 1$.

3. Schedule actors strictly according to their priorities; always execute the actor with the highest priority that is enabled. After executing the k_i-th step, move the actor that executed the step to the end of the list giving it the lowest priority.

4. When an actor a is created, randomly choose j from $\{1, \ldots, |L| + 1\}$ and insert a into L before the jth element. Thus, if $j = 1$ then a becomes the first actor in L and if $j = |L| + 1$ then a is appended to the end of L.

5. Once the schedule reaches a terminal state, update k to be the maximum of the current value of k and number of steps in the schedule. The algorithm repeats from step 1 for the next schedule with the updated value of k. Thus, k is the maximum number of steps in a terminal schedule executed so far.

Note that k is always initialised to 1 before testing each bug for 10,000 schedules; i.e. the updated value of k is not reused across different bugs.

POR In an actor program, it is only necessary to consider the order in which send operations are interleaved; swapping the order of other operations does not change the terminal state reached. This is a form of POR, which we introduced in general in §4.2. In Adara actors, we take a different approach to applying POR that is compatible with random schedulers and takes advantage of the message-passing style. Consider a scheduler that executes only non-send operations until the only enabled actors are all about to execute send operations. The non-send operations would have been executed eventually with any scheduler (assuming an acyclic state space) and would have the same behaviour; they are independent of the not-yet-executed send operations. In particular, any receive

operation that will be dependent with a not-yet-executed send operation must be blocked. The order of other operation types (creating an actor, waiting for an actor, etc.) is similarly uninteresting. We can also explain this in terms of the HBR (§4.2): the only operations ordered in the HBR that can be reversed (i.e. by scheduling different enabled actors to get a different HBR and thus reach a different state) are pairs of send operations.

Thus, to apply POR we modified our controlled random scheduler and PCT scheduler to always execute the first enabled actor (in order of actor creation) that is executing a non-send operation until the only enabled actors are those about to execute sends. At this point, the scheduler chooses to execute one of the actors that is about to execute a send operation. Thus, a send operation is a single a step (transition); all non-send operations (except yield—see below), even those from other actors, are (conceptually) combined into the preceding send operation.

Yielding and liveness Our test system contains actors that retry operations an infinite number of times until they succeed e.g. resolving the primary replica or performing a replication operation. To help prevent unfair schedulers from executing an infinite schedule, we ensure that actors perform a yield operation before retrying. The controlled random scheduler ignores yield operations as its random choices typically avoid infinite schedules. However, the PCT scheduler is typically very unfair and is likely to execute infinite schedules. We used the same approach as in our empirical study (see §3.3); when the PCT scheduler executes a yield operation, the current actor is moved to the end of the list giving it the lowest priority. Note that, for both the random scheduler and PCT scheduler, we make yield count as a step (transition).

There are 3 bugs that we test that can lead to infinite executions. We detect these using a straightforward approach: we apply a step limit of 5,000 steps; if a schedule reaches this step limit, we stop exploring the the schedule and increment the number of observed "livelock schedules" (see Table 6.1). Note that the maximum number of steps observed so far (k) is not updated when the step limit is reached.

Random seeds We use the same approach for exploring random schedules as in our empirical study (see §3.6). Thus, we generate 10,000 random seeds using a random number generator. We reuse these same random seeds for each bug and both schedulers, where each seed is used to seed the random number generator of the scheduler before executing each schedule. Thus, for each seed, bug and scheduler, we execute a single schedule. Note that, unlike in our original empirical study, the schedules produced by our modified PCT scheduler *are* technically dependent on the order in which we execute the schedules

id	name	injected bug?	Rand							PCT d=3						
			# max actors	# max enabled actors	# max steps (k)	# schedules to first bug	# schedules to first livelock	# buggy schedules	# livelock schedules	# max actors	# max enabled actors	# max steps (k)	# schedules to first bug	# schedules to first livelock	# buggy schedules	# livelock schedules
0	BadAssertInPrimaryReplicateOp		104	11	465	-	-	0	0	120	17	456	186	-	118	0
1	UpdateServiceEndpointTooEarly		104	11	465	-	-	0	0	121	17	432	-	19	0	1036
2	PromotePrimaryImmediately		104	11	465	-	-	0	0	119	17	458	124	-	129	0
3	PromotePrimaryImmediatelyBadFix		104	11	465	-	-	0	0	120	17	456	6568	-	1	0
4	UpdateSecondaryEpochTooEarly		104	11	465	-	-	0	0	120	17	456	-	-	0	0
5	BlockForNextId		103	8	436	1	-	9999	0	121	14	459	6	-	5522	0
6	GetCopyStreamNPE		91	10	436	6	-	1427	0	120	17	456	6	-	1558	0
7	ActiveSecondaryCopyStreamDoneButCancelled		104	11	465	-	-	0	0	121	17	461	1477	-	44	0
8	SecondaryCopyStreamDoneButCancelled		104	11	465	-	-	0	0	120	17	456	124	914	21	3
9	DontClearCreateReplicasList		109	11	485	-	-	0	0	120	17	456	244	-	3	0
10	DoReplicateWithoutMutex		109	11	485	7	-	1877	0	120	17	450	36	-	103	0
11	DontWaitWhenSendingCopyState	✓	109	11	485	-	-	0	0	120	17	456	244	-	20	0
12	CommitAckedOpImmediately	✓	109	11	485	15	-	91	0	120	17	456	5	-	1083	0
13	CommitAckedOpImmediatelySkipAssertions	✓	109	11	485	15	-	91	0	120	17	456	5	66	1027	34
14	SendDuplicateRequests	✓	112	11	533	9	-	1491	0	134	17	548	5	-	2353	0

Table 6.1: Experimental SCT results for our Fabric test system. For the controlled random scheduler (Rand) and PCT $d = 3$ scheduler, and for each of the 15 bugs, we execute 10,000 schedules.

because the maximum number of steps seen so far (k) is updated based on the previously executed schedules. However, k is typically updated to close to its final value after just 1 or 2 schedules so we believe the number of buggy schedules found would be similar regardless of the order in which schedules are executed.

6.6.3 Results

We executed our experiments on a MacBook Pro (Retina, 13-inch, Mid 2014) with an Intel Core i7-4578U CPU and 16GB of RAM running Windows 10 64-bit. The C# projects were compiled using Visual Studio Enterprise 2015 Update 2 with .NET 4.6. The full set of data gathered for our testing is shown Table 6.1. Each bug has an id and name for reference, which are shown in the first two columns. For both the controlled random scheduler (Rand) and the PCT d=3 scheduler, and for each bug, we report (in order of the table columns): the maximum number of actors observed at a scheduling point, the maximum number of enabled actors observed at a scheduling point, the maximum number of steps (sends and yields) observed in a single schedule, the number of schedules up to

and including the first buggy schedule (where a buggy schedule is one where an assertion failed), the number of schedules up to and including the first livelock schedule (where a livelock schedule is one where the step limit of 5,000 was reached), the total number of buggy schedules explored, and the total number of livelock schedules explored. If no buggy/livelock schedules were found for a particular bug and scheduler, then we write "-" in the "# schedules to first to bug/livelock" column.

Of the 15 bugs, the random scheduler found 6 while the PCT d=3 scheduler found 14. Thus, PCT was more effective at finding bugs compared to the random scheduler. Furthermore, we note that the random scheduler is not as effective at finding bugs in this test system compared with the benchmarks in our empirical study (§3.4.1) (where the random scheduler found the majority of the bugs). We consider the results for individual bugs below.

Bug overview The list of 15 bugs that we used are shown in our results table (see Table 6.1). We provide the "# max (enabled) actors" and "# max steps (k)" columns for both schedulers to give an indication of the complexity of the test system with each bug enabled. Bugs 0–9 (inclusive) are bugs in our Fabric model, bugs 10–13 are bugs in our shopping list service, and bug 14 is a bug in our client actor.

Fabric model bugs The bugs in our Fabric model are all real (i.e. unintentional) bugs that we found; these were caused by coding or design mistakes that we made when implementing our Fabric model V2. For example, bug 1 (`UpdateServiceEndpointTooEarly`) causes a secondary that is promoted to a primary to continue to be listed as a secondary in the list of endpoints of the service, even though it should be listed as a primary. This can lead to a livelock where the client actor infinitely and unsuccessfully tries to resolve the primary endpoint for the service. As seen in Table 6.1, PCT found livelock schedules due to this bug, while the random scheduler did not; the random scheduler schedules the actor that kills the primary almost immediately, before any secondaries can be initialised. Thus, it completely misses this important scenario. This observation explains why the random scheduler performed poorly on our test system; the eagerness with which it schedules the actor that kills the primary causes it to miss most interesting scenarios.

Interestingly, bug 5 (`BlockForNextId`) was found by the random scheduler on almost all schedules (9999), while PCT only exposed the bug on just over half of the schedules (5522). The bug causes the `ServiceRuntime` and `ClusterRuntime` actors to deadlock, due to the `ServiceRuntime` actor waiting for a message from the `ClusterRuntime` actor (and vice versa) *from within an action*, which, although possible in Adara actors, is arguably

not idiomatic actor programming as it can lead to deadlock. The bug can occur within just a few steps, such as in the following scenario: first, the `ClusterRuntime` actor creates the `ServiceRuntime` actor; second, the `ClusterRuntime` actor receives a request from the client actor to resolve the primary replica, and so sends a message to the `ServiceRuntime` actor and then blocks until it receives a response; the `ServiceRuntime` actor sends a request to the `ClusterRuntime` actor asking for an id for the initial primary replica and also blocks until it receives a response. We believe that the random scheduler performs well because the bug is very likely to occur if the `ServiceRuntime` actor is preempted before sending a request for a replica id (and the random scheduler is very likely to preempt actors). Furthermore, the client actor often repeatedly tries to resolve the replicas, which means it is very likely that a resolve request will be interleaved with the id request, which causes the deadlock. In contrast, there are probably only a few depths at which PCT can insert a change point for the bug to be exposed, and the relative actor priorities of the client actor, `ServiceRuntime` and `ClusterRuntime` actors will also need to allow the required interleaving.

Notice that a few of the bugs in our Fabric model were exposed via a small number of buggy schedules (i.e. < 30) and so can perhaps be considered "difficult to find" bugs. In particular, bug 3 (`PromotePrimaryImmediatelyBadFix`) was exposed by only a single schedule using the PCT scheduler. This bug is related to bug 2, which occurs when a secondary is under consideration for being promoted to a primary while it is still applying replication operations. Given a very specific interleaving of operations, this bug can cause our Fabric model to lose track of which replication operations have been applied at the new primary resulting in the Fabric model re-issuing already-used LSNs. Bug 3 is a fix that we applied for bug 2 which turned out to also be buggy, but requires an even more specific interleaving of operations to occur, which explains why only one buggy schedule was found.

We note that bug 4 (`UpdateSecondaryEpochTooEarly`) was not found by either scheduler. This bug was found by inspection; we believe this bug could be exposed using the test harness given the right schedule but we were not able to reproduce it yet.

Real shopping list service bug Bug 10 (`DoReplicateWithoutMutex`) is the only real bug we found in our shopping list service. It was caused by an actor calling `ReplicateAsync` at the primary (to send a replication operation to the secondaries) without first locking the monitor. This goes against the advice of the Fabric documentation. We attempted this approach because we thought it was valid and would maximise the concurrency of replication operations and so better test our Fabric model. We imagine

developers might (incorrectly) attempt this to achieve additional parallelism. However, if the actor calling `ReplicateAsync` is interrupted by another actor (also trying to send a replication operation) before acquiring the monitor, *and* both replication operations are confirmed by a majority of secondaries before the interrupted actor can acquire the monitor, the primary can end up applying replication operations in the wrong order. As seen in Table 6.1, this bug was found by both schedulers.

Injected shopping list service bugs Bugs 11–13 represent bugs that we believe developers are likely to introduce when attempting to write Fabric services. We injected these bugs into our shopping list service to test whether our Fabric model contains enough asynchrony to expose these bugs.

Bug 11 (`DontWaitWhenSendingCopyState`) is a bug in the primary code that provides the state of the shopping list to new secondaries; Fabric asks the primary to send the state that would be observed after applying a particular replication operation (and all those before it). However, the primary may not have actually applied the specified replication operation yet and so must wait until it has been applied (the operation is applied by another actor) so that the correct state is copied to the secondary. We achieve this by waiting on the monitor until the specified replication operation has been applied. When bug 11 is enabled, we send the current state of the shopping list without waiting, causing the secondary to potentially receive an old version of the shopping list. It would be very unlikely to observe this on the real Fabric because of the small window of time between Fabric notifying the the primary that a replication operation has been acknowledged and the primary applying the replication operation. As seen in Table 6.1, this bug was only exposed by the PCT scheduler, and on just 20 schedules.

Bug 12 (`CommitAckedOpImmediately`) affects the way the primary applies replication operations after they have been acknowledged by a majority of secondaries. Fabric may notify the primary of acknowledged replication operations in the wrong order, and so the code at the primary must carefully store a list of unapplied replication operations so that it only applies consecutive, acknowledged operations, starting from the front of the list. When bug 12 is enabled, the primary applies acknowledged replication operations immediately without considering their order, potentially resulting in the shopping list items ending up in the wrong order at the primary. Bug 13 is the same as bug 12, but skips some assertions in our primary code so that the error is not immediately detected. As seen in Table 6.1, bugs 12 and 13 were detected by both schedulers. Interestingly, bug 13 sometimes results in livelock, as found by PCT. We believe this occurs when the primary gets blocked trying to send a copy of the shopping list to a secondary when waiting for

the latest replication operation to be applied; the actor is never unblocked because only the most recently applied replication operation is checked which (due to the bug) is not the replication operation with the highest (and the required) LSN. This causes the client actor's add operations to repeatedly fail (until the secondary is brought up-to-date, which never happens).

Injected client actor bug Finally, bug 14 is a bug in the client actor that causes it to sometimes send duplicate add operations; this is caused by the actor reading the state of the shopping list at the primary and requesting all missing items to be added without taking into account previous add operations that are still in-progress (but not yet visible at the primary). This bug is based on a real error that we made when implementing the client actor, but we bounded the maximum number of in-progress operations that the client would send so that the bug would not exceed the step limit of 5,000. As seen in Table 6.1, this bug was detected by both schedulers.

6.6.4 Main findings

We now report our main findings by answering our research questions (RQs) given at the beginning of this section.

RQ1: Can we find bugs in our test system using SCT, thus showing that we can apply SCT to actor-style distributed systems written for the Azure Service Fabric platform? Yes. We found 5 bugs in our shopping list and client, and a further 9 in our Fabric model. We have not yet reproduced 1 bug that we believe exists in our Fabric model but we think it could be reproduced given the appropriate schedule.

RQ2: Does our Fabric model include enough asynchrony/nondeterminism to expose the injected bugs in our system? (We believe the injected bugs highlight several tricky and subtle aspects of developing Fabric services that developers are likely to get wrong at some point.) Yes. We found all 4 injected bugs. Bug 11 (`DontWaitWhenSendingCopyState`) was only found using the PCT scheduler while the other 3 were found by both the PCT and random schedulers.

RQ3: How do the controlled random scheduler and PCT d=3 scheduler compare in terms of number of bugs found, and how do these results compare to our results of our empirical study in Chapter 3? Of the 15 bugs, the random scheduler found 6 while the PCT scheduler found 14 (including all those found by the

random scheduler). Thus, PCT was more effective at finding bugs in this system. The random scheduler performed poorly compared to our previous study (where it found the majority of bugs). We believe this is because it eagerly schedules the actor that kills the primary replica which causes it to miss many interesting scenarios. There was one exception: bug 5 (`BlockForNextId`) was found on almost 100% of schedules using the random scheduler, and on just over 50% of schedules when using PCT. Of course, we only used one test system (with multiple bugs) which may be a mostly adversarial system for the random scheduler. However, we believe that actor-based test systems that wish to simulate failure (such as killing the primary replica) are likely to include actors that trigger this failure when scheduled; the random scheduler is likely to always perform poorly in such systems. Future work could investigate schedulers that take these failure-triggering actors into account.

RQ4: How easy was it to apply SCT to this actor-based system compared to applying SCT to our C/C++ benchmarks in Chapter 3 and our Java benchmarks in Chapter 4? Applying SCT was much easier than in previous chapters as we did not have to instrument binaries/classes, track various different visible operations, nor detect data races (although data races may still exist in our system). The modelling effort was, of course, much greater. We hope that this cost can be amortized over testing multiple Fabric services. Also, testing other distributed systems that use a less-complex platform (such as the higher-level Fabric APIs) might require less modelling effort. However, we note that porting an existing non-actor-based system to an idiomatic, actor-based equivalent is a significant design and engineering challenge that we did not attempt (outside of modelling Fabric). Furthermore, naïvely porting a system to use nonidiomatic actors (as we did with the shopping list service) is not ideal and would not always be as easy as it was in our case; our shopping list service was fairly straightforward and did not use many synchronisation operations. Thus, the benefits of actor-based systems are not immediately available to existing code without re-writing. Overall, we conclude that actor-based systems are ideally suited for SCT due to actors only communicating via message-passing, which is mediated by the actor runtime and thus easy to control. Furthermore, message-passing enables a straightforward POR approach that we were able to apply to our random schedulers (§6.6.2).

6.7 Related work

Our work is not the first example of applying SCT to actor-based systems. P# was inspired by the P language [DGJ+13], a domain specific language for actor-style programming that compiles to C for production and to the Zinger model checking language for SCT. P was used to implement and verify the USB device driver stack that ships with Microsoft Windows 8. P# was introduced in [DDK+15] and SCT was performed on several benchmarks. A number of P# case studies (including our first Fabric model) were described in [DMT+16]. Concuerror [GCS11, CGS13, AAJS14] (as discussed in §5.5) is a systematic concurrency testing tool for Erlang programs (the Erlang language uses the actor model [HBS73]). Concuerror supports several techniques, including preemption bounding [MQ07b] and optimal DPOR [AAJS14]. We note that Concuerror supports a technique referred to as *blocking avoidance* that we believe is similar in spirit to our straightforward POR approach of combining execution of non-send operations with the previous send operation (see §6.6.2).

6.8 Conclusion

We have presented a large case study in which we created a model of Azure Service Fabric so we can apply SCT to distributed systems written for Fabric. We described our Adara actors framework that provides portable, statically typed actors, which we used to create the final version of our Fabric model. We found 14 of the 15 bugs in our test system using SCT, including the 4 injected bugs that we believe are representative of mistakes that developers are likely to make when developing Fabric services. Thus, our Fabric model contains enough behaviours/asynchrony to expose these subtle pitfalls. We note that applying SCT to actor-based systems is more straightforward than in shared-memory systems. As in our prior empirical study (Chapter 3), the PCT d=3 scheduler performed well, finding 14 of the bugs, while the random scheduler was less effective than before, finding only 6 of the bugs.

7 Conclusions and future work

7.1 Contributions

This thesis has made the following contributions to the field of SCT:

- In Chapter 3, we presented an empirical study of existing SCT techniques over 48 concurrent benchmarks. Our most surprising result was that the "naïve" controlled random scheduler, which randomly chooses one thread to execute at each scheduling point, performed well, finding more bugs than preemption bounding. PCT (with parameter d=3) performed the best and only missed one bug which was also missed by all other techniques. The results call into question whether schedule bounding is an effective technique for finding bugs *and/or* whether the concurrency benchmarks used in research are useful. We report that several benchmarks are arguably trivial. For example, the bugs in 18 benchmarks were exposed at least 50% of the time when using the random scheduler.

- In Chapter 4, we introduced the *lazy happens-before relation* (lazy HBR) which provides reduction *beyond* partial-order reduction for programs that use mutexes. We proved that schedules with identical lazy HBRs are guaranteed to reach identical states and presented two reduction techniques backed by the lazy HBR: *lazy HBR caching* and *lazy dynamic partial-order reduction*. Our experimental results showed the significant potential and practical reduction of using the lazy HBR.

- In Chapter 5, we described implementation details of our SCT tool, JESS, which we believe to be an important contribution because such implementation details are rarely discussed in prior work. We described our race detection algorithm which we believe to be more efficient than any prior work.

- In Chapter 6, we applied SCT in the context of distributed systems written for *Azure Service Fabric* (Fabric)—a platform and API for reliable services. We introduced our Adara actors framework for writing *portable*, *statically-typed actors*. We evaluated our Fabric model on a system containing 11 real bugs and 4 API-related injected

bugs. We found 14 of the 15 bugs using SCT, including all of the injected bugs (that we believe are representative of mistakes that developers are likely to make when developing Fabric services) showing that our Fabric model includes enough behaviours/asynchrony to expose these subtle pitfalls. We note that applying SCT to actor-based systems, where actors are restricted to just sending and receiving messages, is more straightforward than in shared-memory systems where one has to consider the memory model, instrumenting every memory access, and a larger set of concurrency functions (versus just *send* and *receive*).

In summary, in this thesis we have evaluated, improved, described, and applied practical systematic concurrency testing.

7.2 Future work

Future SCT studies We believe that further studies that compare additional SCT techniques would greatly benefit the field. Our study in Chapter 3 is limited by the set of techniques and benchmarks that we consider. Partial-order reduction techniques (which we only consider in isolation in Chapter 4) are obvious candidates for future studies. In particular, it would be interesting to test whether sound reduction techniques (like optimal DPOR [AAJS14]) provide enough reduction to allow exhaustive exploration; in §4.6.3, we encountered a number of benchmarks that could not be exhaustively explored within our schedule limit by DPOR nor lazy DPOR, although neither of these techniques is optimal. If sound reduction approaches cannot explore all terminal states then it would be useful to test to what extent they are worthwhile for bug finding. We would also like to create and test other straightforward randomisation techniques as these seem to be surprisingly effective. However, perhaps the most challenging aspect of these types of studies is obtaining concurrent benchmarks that (a) are amenable to SCT and (b) represent code that is used in practice. Applying SCT to complex programs like web browsers or servers is probably infeasible; instead, we can consider testing certain libraries and modules in isolation. However, as noted in our study, even this is often nontrivial due to the complexity and poor testability of codebases. Indeed, a study that focuses on the challenges of applying SCT to one or two complex open source projects could be useful in itself. We believe it would be useful to include multiple tools in future studies so as to increase the types of benchmarks that can be considered; our study was limited to C/C++ programs that use pthreads as this is what the Maple tool targets.

Other concurrency testing techniques Some concurrency testing techniques do not serialise execution and/or do not rely on a deterministic target program [EFN+02, PLZ09, YNPP12, BKMN10, NBMM12]. As such, they can be applied to a greater number of benchmarks with less effort, including complex programs like web browsers. It would be useful to compare these techniques with each other and with SCT techniques; it would be useful to test whether the additional control provided by SCT is actually advantageous for bug finding compared to the most effective non-SCT approaches.

Sound reduction algorithms As suggested in Chapter 4, it would be useful to improve our lazy DPOR algorithm so that it is sound for programs that do not contain mutex-deadlock states. It would be useful to compare our lazy DPOR algorithm with optimal DPOR [AAJS14], and to consider creating a lazy version of optimal DPOR. However, perhaps the most interesting direction for reduction is demonstrated by the *maximal causality reduction* (MCR) [Hua15] algorithm that is optimal and reduces beyond POR (and thus beyond optimal DPOR) using an SMT solver. It would be useful to evaluate the overhead of using an SMT solver; MCR uses parallelism to overcome the solver bottleneck but the additional CPU time is still a cost and DPOR could also be parallelised in theory. Nevertheless, the additional CPU time needed for the reduction may be justified by the savings from not exploring redundant schedules.

Relaxed memory models We assumed sequential consistency throughout this thesis; as described in §2.1, this is common in prior work and is not a limitation for programs that are *intended* to be free from data races (and free from weak memory operations) and thus only exhibit sequentially consistent behaviours. Chronological traces [AAA+15] allow optimal DPOR to check programs for the TSO and PSO memory models. The MCR [Hua15] has also been extended [HH16] to support TSO and PSO. Applying such techniques to programs that have only sequentially consistent behaviours should not increase the number of schedules explored. However, enabling exploration of schedules that exhibit the effects of weak memory (i.e. forcing delayed visibility of writes) has a cost in terms of time and engineering effort; the work of [AAA+15] uses an LLVM interpreter to explore schedules, which is slower than native execution. This is just one approach, but this requirement for additional control is yet another challenge which may make it less straightforward to apply SCT in practice and, as described in §3.4.2, we believe this is already challenging. Thus, perhaps it is worthwhile performing this more challenging SCT for weak memory models on small test cases that test implementations of concurrency primitives (mutexes, concurrent containers, etc.) where the additional overhead and

restrictions are less troublesome, and then rely on regular sequentially consistent SCT and data race detection for testing large programs that use these primitives, assuming the primitives can "shield" the program from the effects of the weak memory model [GMY12]. Nevertheless, in future work, it would be useful to investigate if there are ways of exploring weak memory behaviours without significantly increasing execution time.

Bibliography

[AAA+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In *TACAS*, pages 353–367, 2015.

[AAJS14] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *POPL*, pages 373–384, 2014.

[Akk] Akka.NET. Akka.NET. `http://getakka.net/`. Online. Accessed: 2016-09.

[AKT13] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In *CAV*, pages 141–157, 2013.

[Apa] Apache Software Foundation. Apache thrift. `https://thrift.apache.org/`. Online. Accessed: 2016-09.

[B+07] Norris Boyd et al. Rhino: Javascript for Java. *Mozilla Foundation. Project Website: http://www.mozilla.org/rhino*, 2007.

[BAEFU06] Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Producing scheduling that causes concurrent programs to fail. In *PADTAD*, pages 37–40, 2006.

[BCD+12] Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, and Paul Thomson. Gpuverify: a verifier for GPU kernels. In *OOPSLA*, pages 113–132, 2012.

[BCD+15] Adam Betts, Nathan Chong, Alastair F. Donaldson, Jeroen Ketema, Shaz Qadeer, Paul Thomson, and John Wickerson. The design and implementation of a verification technique for GPU kernels. *ACM Trans. Program. Lang. Syst.*, 37(3):10:1–10:49, 2015.

[BH05] Saddek Bensalem and Klaus Havelund. Dynamic deadlock analysis of multi-threaded programs. In *HVC*, pages 208–223, 2005.

[Bie11] Christian Bienia. *Benchmarking Modern Multiprocessors*. PhD thesis, Princeton University, 2011.

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *ASPLOS*, pages 167–178, 2010.

[Boe11] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *HotPar*, pages 1–6, 2011.

[CBM10] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *PPoPP*, pages 15–24, 2010.

[CE81] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In *Logic of Programs*, pages 52–71, 1981.

[CF11] Lucas Cordeiro and Bernd Fischer. Verifying multi-threaded software using SMT-based context-bounded model checking. In *ICSE*, pages 331–340, 2011.

[CGS13] Maria Christakis, Alkis Gotovos, and Konstantinos F. Sagonas. Systematic testing for detecting concurrency errors in erlang programs. In *ICST*, pages 154–163, 2013.

[CMM13] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In *OOPSLA*, pages 833–848, 2013.

[Coo16] Katherine E. Coons. private communication, October 2016.

[CRKH05] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. 2005.

[CWY11] Qichang Chen, Liqiang Wang, and Zijiang Yang. SAM: self-adaptive dynamic analysis for multithreaded programs. In *HVC*, pages 115–129, 2011.

[DDK+15] Pantazis Deligiannis, Alastair F. Donaldson, Jeroen Ketema, Akash Lal, and Paul Thomson. Asynchronous programming, analysis and testing with state machines. In *PLDI*, pages 154–164, 2015.

[DGJ+13] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: safe asynchronous event-driven programming. In *PLDI*, pages 321–332, 2013.

[DMT+16] Pantazis Deligiannis, Matt McCutchen, Paul Thomson, Shuo Chen, Alastair F. Donaldson, John Erickson, Cheng Huang, Akash Lal, Rashmi Mudduluru, Shaz Qadeer, and Wolfram Schulte. Uncovering bugs in distributed storage systems during testing (not in production!). In *FAST*, pages 249–262, 2016.

[EFN+02] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded java program test generation. *IBM Syst. J.*, 41(1):111–125, 2002.

[ELC02] E.Bruneton, R. Lenglet, and T. Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *Adaptable and extensible component systems*, November 2002.

[EP14] Mahdi Eslamimehr and Jens Palsberg. Race directed scheduling of concurrent programs. In *PPoPP*, pages 301–314, 2014.

[EQR11] Michael Emmi, Shaz Qadeer, and Zvonimir Rakamarić. Delay-bounded scheduling. In *POPL*, pages 411–422, 2011.

[Fam15] Bob Familiar. *Microservices, IoT, and Azure: Leveraging DevOps and Microservice Architecture to Deliver SaaS Solutions*. Apress, 1st edition, 2015.

[FF09] Cormac Flanagan and Stephen N. Freund. FastTrack: efficient and precise dynamic race detection. In *PLDI*, pages 121–133, 2009.

[FF10] Cormac Flanagan and Stephen N. Freund. The RoadRunner dynamic analysis framework for concurrent programs. In *PASTE*, pages 1–8, 2010.

[FF13] Cormac Flanagan and Stephen N. Freund. RedCard: Redundant check elimination for dynamic race detectors. In *ECOOP*, pages 255–280, 2013.

[FG05] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *POPL*, pages 110–121, 2005.

[FG11] Cormac Flanagan and Patrice Godefroid. Addendum to dynamic partial-order reduction for model checking software. 2011.

[FLR98] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[FSS04] Michael Factor, Assaf Schuster, and Konstantin Shagin. Instrumentation of standard libraries in object-oriented languages: The twin class hierarchy approach. In *OOPSLA*, pages 288–300, 2004.

[GCS11] Alkis Gotovos, Maria Christakis, and Konstantinos F. Sagonas. Test-driven development of concurrent programs using concuerror. In *Erlang*, pages 51–61, 2011.

[GJS+13] James Gosling, Bill Joy, Guy L. Steele, Jr., Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 7 Edition*. Addison-Wesley Professional, 1st edition, 2013.

[GMY12] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Show no weakness: Sequentially consistent specifications of TSO libraries. In *DISC*, pages 31–45, 2012.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[God97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL*, pages 174–186, 1997.

[GP93] Patrice Godefroid and Didier Pirottin. Refining dependencies improves partial-order verification methods. In *CAV*, pages 438–449, 1993.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular ACTOR formalism for artificial intelligence. In *IJCAI*, pages 235–245, 1973.

[HF11] Gerard J. Holzmann and Mihai Florian. Model checking with bounded context switching. *Formal Asp. Comput.*, 23(3):365–389, 2011.

[HH16] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *OOPSLA*, 2016.

[HJG11] Gerard J. Holzmann, Rajeev Joshi, and Alex Groce. Swarm verification techniques. *IEEE Trans. Software Eng.*, 37(6):845–857, 2011.

[HO09] Philipp Haller and Martin Odersky. Scala actors: Unifying thread-based and event-based programming. *Theoretical Computer Science*, 410(2):202–220, 2009.

[Hol03] Gerard Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, first edition, 2003.

[Hua15] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *PLDI*, pages 165–174, 2015.

[HZ11] Jeff Huang and Charles Zhang. An efficient static trace simplification technique for debugging concurrent programs. In *SAS*, pages 163–179, 2011.

[JPPS11] Nicholas Jalbert, Cristiano Pereira, Gilles Pokam, and Koushik Sen. RADBench: a concurrency bug benchmark suite. In *HotPar*, pages 1–6, 2011.

[JS10] Nicholas Jalbert and Koushik Sen. A trace simplification technique for effective debugging of concurrent programs. In *FSE*, pages 57–66, 2010.

[KLVU10] Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, and Shmuel Ur. A platform for search-based testing of concurrent software. In *PADTAD*, pages 48–58, 2010.

[KP92] Shmuel Katz and Doron Peled. Defining conditional independence using collapses. *Theor. Comput. Sci.*, 101(2):337–359, July 1992.

[KZC12] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with Portend. In *ASPLOS*, pages 185–198, 2012.

[L+05] Chi-Keung Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, pages 190–200, 2005.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[LB98] Bil Lewis and Daniel J. Berg. *Multithreaded Programming with Pthreads*. Prentice-Hall, 1998.

[LPSZ08] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *ASPLOS*, pages 329–339, 2008.

[Mic] Microsoft. What is windows communication foundation. `https://msdn.microsoft.com/en-us/library/ms731082(v=vs.110).aspx`. Online. Accessed: 2016-09.

[MM07] Tom Ball Madan Musuvathi, Shaz Qadeer. Chess: A systematic testing tool for concurrent software. Technical report, November 2007.

[MQ07a] M. Musuvathi and S. Qadeer. Partial-order reduction for context-bounded state exploration. Technical Report MSR-TR-2007-12, Microsoft Research, 2007.

[MQ07b] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *PLDI*, pages 446–455, 2007.

[MQ08] Madanlal Musuvathi and Shaz Qadeer. Fair stateless model checking. In *PLDI*, pages 362–371, 2008.

[MQB+08] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing Heisenbugs in concurrent programs. In *OSDI*, pages 267–280, 2008.

[N+07] Satish Narayanasamy et al. Automatically classifying benign and harmful data races using replay analysis. In *PLDI*, pages 22–31, 2007.

[NBMM12] Santosh Nagarakatte, Sebastian Burckhardt, Milo M.K. Martin, and Madanlal Musuvathi. Multicore acceleration of priority-based schedulers for concurrency bug detection. In *PLDI*, pages 543–554, 2012.

[Nit14] Michael Nitschinger. Debugging concurrency issues with OpenJDK jcstress. *http://nitschinger.at/Debugging-Concurrency-Issues-with-Open-JDK-Jcstress*, 2014.

[Ora] Oracle. Java SE 7 Remote Method Invocation API. `http://docs.oracle.com/javase/7/docs/technotes/guides/rmi/index.html`. Online. Accessed: 2016-09.

[PHW07] Donald E. Porter, Owen S. Hofmann, and Emmett Witchel. Is the optimism in optimistic concurrency warranted? In *HotOS*, pages 1:1–1:6, 2007.

[PJ14] Pavel Parízek and Pavel Jančík. Approximating happens-before order: Interplay between static analysis and state space traversal. In *SPIN*, pages 1–10, 2014.

[PL11a] Pavel Parízek and Ondrej Lhoták. Identifying future field accesses in exhaustive state space traversal. In *ASE*, pages 93–102, 2011.

[PL11b] Pavel Parizek and Ondrej Lhoták. Randomized backtracking in state space traversal. In *SPIN*, pages 75–89, 2011.

[PLZ09] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing atomicity violation bugs from their hiding places. In *ASPLOS*, pages 25–36, 2009.

[Pon] Pony Developers. Pony. `http://www.ponylang.org/`. Online. Accessed: 2016-09.

[PSE07] Jeff H. Perkins, David Saff, and Michael D. Ernst. Instrumentation of standard libraries in Java. In *Research Abstracts*. CSAIL, 2007.

[QS82] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in cesar. In *Proceedings of the 5th Colloquium on International Symposium on Programming*, pages 337–351, 1982.

[RM09] Neha Rungta and Eric G. Mercer. Clash of the titans: Tools and techniques for hunting bugs in concurrent programs. In *PADTAD*, pages 9:1–9:10, 2009.

[SB01] Lorna A Smith and J Mark Bull. A multithreaded Java grande benchmark suite. In *Java for High Performance Computing*, 2001.

[SBGH12] Jirí Simsa, Randy Bryant, Garth A. Gibson, and Jason Hickey. Scalable dynamic partial order reduction. In *RV*, pages 19–34, 2012.

[Sen08] Koushik Sen. Race directed random testing of concurrent programs. In *PLDI*, pages 11–21, 2008.

[SES+12] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *POPL*, pages 387–400, 2012.

[SI09] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: data race detection in practice. In *WBIA*, pages 62–71, 2009.

[SKH12] Olli Saarikivi, Kari Kahkonen, and Keijo Heljanko. Improving dynamic partial order reductions for concolic testing. In *ACSD*, pages 132–141, 2012.

[SL05] Herb Sutter and James Larus. Software and the concurrency revolution. *ACM Queue*, 3(7):54–62, 2005.

[SSO+10] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-TSO: a rigorous and usable programmer's model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

[TD15] Paul Thomson and Alastair F. Donaldson. The lazy happens-before relation: better partial-order reduction for systematic concurrency testing. In *PPoPP*, pages 259–260, 2015.

[TDB14] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: An empirical study. In *PPoPP*, pages 15–28, 2014.

[TDB16] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using controlled schedulers: An empirical study. *ACM Trans. Parallel Comput.*, 2(4):23:1–23:37, February 2016.

[W+95] Steven Cameron Woo et al. The SPLASH-2 programs: Characterization and methodological considerations. In *ISCA*, pages 24–36, 1995.

[WSG11] Chao Wang, Mahmoud Said, and Aarti Gupta. Coverage guided systematic concurrency testing. In *ICSE*, pages 221–230, 2011.

[YCG08] Y. Yang, X. Chen, and G. Gopalakrishnan. Inspect: A runtime model checker for multithreaded C programs. Technical Report UUCS-08-004, University of Utah, 2008.

[YN09] Jie Yu and Satish Narayanasamy. A case for an interleaving constrained shared-memory multi-processor. In *ISCA*, pages 325–336, 2009.

[YNPP12] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *OOPSLA*, pages 485–502, 2012.

[ZKW15] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *PLDI*, 2015.