Practical Concurrency Testing

or: How I Learned to Stop Worrying and Love the Exponential Explosion

Ben Blum

Decembruary 2018

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Garth Gibson, Chair
David A. Eckhardt
Brandon Lucia
Haryadi Gunawi, University of Chicago

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy.

Keywords: landslide terminal, baggage claim, ground transportation, ticketing

For my family, my teachers, and my students.

Abstract

Concurrent programming presents a challenge to students and experts alike because of the complexity of multithreaded interactions and the difficulty to reproduce and reason about bugs. Stateless model checking is a concurrency testing approach which forces a program to interleave its threads in many different ways, checking for bugs each time. This technique is powerful, in principle capable of finding any nondeterministic bug in finite time, but suffers from exponential explosion as program size increases. Checking an exponential number of thread interleavings is not a practical or predictable approach for programmers to find concurrency bugs before their project deadlines.

In this thesis, I develop several new techniques to make stateless model checking more practical for human use. I have built Landslide, a stateless model checker specializing in undergraduate operating systems class projects. Landslide extends the traditional model checking algorithm with a new framework for automatically managing multiple state spaces according to their estimated completion times, which I show quickly finds bugs should they exist and also quickly verifies correctness otherwise. I evaluate Landslide's suitability for inexpert use by presenting the results of many semesters providing it to students in 15-410, CMU's Operating System Design and Implementation class, and more recently, students in similar classes at the University of Chicago and Penn State University. Finally, I extend Landslide with a new concurrency model for hardware transactional memory, and evaluate several real-world transactional benchmarks to show that stateless model checking can keep up with the developing concurrency demands of real-world programs.

Acknowledgments

I thank my dog, Louie.

Contents

1 Introduction **1**

2 Background **7**
2.1 Concurrency . . . 7
2.1.1 The Basics . . . 7
2.1.2 Identifying bugs . . . 8
2.1.3 Concurrency Primitives . . . 8
2.1.4 Transactional Memory . . . 9
2.2 Stateless Model Checking . . . 11
2.2.1 The state space . . . 11
2.2.2 On the size of state spaces . . . 14
2.3 Data Race Analysis . . . 15
2.3.1 Definition . . . 15
2.3.2 Happens-Before . . . 16
2.4 Education . . . 17
2.4.1 Pebbles . . . 17
2.4.2 Pintos . . . 21
2.5 Glossary . . . 22

3 Landslide **23**
3.1 User interface . . . 23
3.1.1 Setup . . . 24
3.1.2 Running Landslide through Quicksand . . . 24
3.1.3 Running Landslide directly . . . 26
3.1.4 Test cases . . . 28
3.1.5 Bug reports . . . 30
3.2 Kernel annotations . . . 30
3.2.1 config.landslide annotations . . . 30
3.2.2 In-kernel code annotations . . . 33
3.3 Architecture . . . 35
3.3.1 Execution tree . . . 36
3.3.2 Scheduler . . . 36
3.3.3 Memory analysis . . . 37
3.3.4 Machine state manipulation . . . 38

3.3.5 State space traversal . . . 39
3.3.6 Bug-finding output . . . 40
3.3.7 Pebbles-specific features . . . 42
3.3.8 Pintos-specific features . . . 43
3.3.9 Handy scripts . . . 44
3.4 Algorithms . . . 45
3.4.1 Preemption point identification . . . 45
3.4.2 Dynamic Partial Order Reduction . . . 47
3.4.3 State space estimation . . . 54
3.4.4 Data race analysis . . . 57
3.4.5 Iterative Context Bounding . . . 58
3.4.6 Heuristic loop, deadlock, and synchronization detection . . . 60

4 Quicksand **61**
4.1 Preemption points . . . 61
4.2 Iterative Deepening . . . 61
4.2.1 Soundness . . . 61
4.3 Evaluation . . . 61

5 Education **63**
5.1 Pebbles . . . 63
5.1.1 Recruiting . . . 64
5.1.2 Automatic instrumentation . . . 65
5.1.3 Test cases . . . 66
5.1.4 Survey . . . 67
5.2 Pintos . . . 68
5.2.1 Recruiting . . . 68
5.2.2 Automatic instrumentation . . . 68
5.2.3 Test cases . . . 69
5.2.4 Survey . . . 70
5.3 Evaluation . . . 71
5.3.1 Bug-finding . . . 71
5.3.2 Student submission quality . . . 75
5.3.3 Survey responses . . . 80
5.4 Discussion . . . 86
5.4.1 Bias . . . 87
5.4.2 Retrospect . . . 87
5.4.3 Future educational use . . . 89

6 Transactions **91**
6.1 Concurrency Model . . . 92
6.1.1 Example . . . 93
6.1.2 Modeling Transaction Failure . . . 93
6.1.3 Memory Access Analysis . . . 96

6.2 Implementation . . . 97
6.2.1 User Interface . . . 97
6.2.2 Failure Injection . . . 98
6.2.3 Data Race Analysis . . . 99
6.2.4 Retry independence . . . 100
6.3 Evaluation . . . 101
6.3.1 Bug-finding . . . 103
6.3.2 Verification . . . 107
6.4 Discussion . . . 112

7 Related Work **117**
7.1 Stateless Model Checking . . . 117
7.2 Data race analysis . . . 120
7.3 Concurrency in education . . . 121
7.4 Transactional memory . . . 123
7.5 Other concurrency verification approaches . . . 124

8 Future Work **127**

9 Conclusion **129**

Bibliography **131**

List of Figures

2.1 Example concurrent program in which simultaneous accesses to shared state may interleave to produce unexpected results. 9
2.2 Using a locking primitive to protect accesses to shared state. 10
2.3 The example `count` routine from Figure 2.2, rewritten to use HTM. If the transaction in the top branch aborts, whether from a memory conflict or random system interrupt, execution will revert to the return of `_xbegin`, `status` will be assigned an error code indicating the abort reason, and control will drop into the `else` branch. The programmer can then use explicit synchronization, such as a mutex, to resolve the conflict. 11
2.4 Visualization of interleaving state space for the program in Figure 2.1. Thread 1 is represented by purple ovals, thread 2 by yellow squares, and time flows from top to bottom. As the two threads execute the same code, without loss of generality thread 1 is fixed to run first – the full state space is twice the size, and the other half is symmetric to the one shown. 12
2.5 DPOR identifies independent transitions by different threads which can commute without affecting program behaviour. Here, if the transitions marked 1 and 2 have no shared memory conflicts, the states marked with the red arrow are guaranteed identical. Hence, only one of the subtrees need be explored. 15
2.6 Data-race analyses may be prone to either *false negatives* or *false positives*. Applying Happens-Before to program (a) will miss the potential race possible between A1/A2 in an alternate interleaving, while using Limited Happens-Before on (b) will produce a false alarm on B1/B2. 16

3.1 Example demonstrating the need for `misbehave` to force the kernel to yield during `thread_fork`. 29
3.2 Example preemption trace bug report. 31
3.3 Result of a single iteration of DPOR. 50
3.4 Result of 3 DPOR iterations, pruning 7 redundant branches. 51
3.5 DPOR's termination state, having reduced 20 interleavings to 6. 52
3.6 Motivating example for the sleep sets optimization. Three of Figure 3.5's interleavings are highlighted, with the always-independent `tmpN++`s omitted for brevity. 53

5.1 Distribution of project grades among Landslide users and non-users. Study participants (S'15 to S'18) are the experimental group; non-participants (S'15 to S'18) and students from semesters predating the study (F'13-F14) are the control groups. Compare Landslide users to non-users for within-semester differences, or users to pre-study students to mitigate selection bias. 76
5.2 Detailed statistics from student grade distributions. 77
5.3 Student survey responses. SD/D/N/A/SA stands for strongly disagree/ disagree/neutral/agree/strongly agree. 81

6.1 Example TSX program. 92
6.2 Variant of the program in Figure 6.1, with additional synchronization to protect the failure path from the transactional path. The optional line 0 serves to prevent a cascade of failure paths for the sake of performance by allowing threads to wait until transacting is safe again. 94
6.3 Motivating example for retry independence reduction. 100
6.4 Unmodified code from `htmavl.hpp` showing the previously-unknown bug Landslide found in `avl_insert`. The transaction path fails to check `_retry`, leading to data races and corruption just as in `htm1`. 105
6.5 Code from `spinlock-rtm.c`, modified only to remove unrelated logic for brevity, showing the performance bug Landslide found in `lock_fast(spinlock)`. The `isfree()` routine uses an atomic read-and-write operation where just a read would suffice, which leads to superfluous memory conflicts in the transactional path (seen at both of its callsites below). 106
6.6 Visualization of retry set state space reduction. On both `counter`(2,1) and `fig63`(2,1), baseline DPOR tests all 10 interleavings pictured, with the middle 2 arising from the data-race preemption point within the abort path. With the reduction enabled, after the 4th and 6th branches (i.e., when preempting to reorder threads), Landslide activates the retry set indicated at the top of the next upcoming subtree, allowing it to identify and skip 2 redundant branches in `counter` and 4 in `fig63`. 110
6.7 Abstraction reduction test cases. 112

List of Tables

2.1 The Pebbles specifcation defines 25 system calls. Students are not required to implement ones marked with an asterisk (*), though the reference kernel provides them. 18

5.1 Participation rates across semesters, among students who submitted thread libraries (CMU, PSU) or kernels (U. Chicago). 72

5.2 Landslide's bug-finding performance across all semesters of the CMU 15-410 study. $\leq$ marks possible overcounts on account of unidentified false positives; $\geq$ marks possible undercounts on account of students not necessarily re-running to verify bugfixes. 73

5.3 Correlation of student Landslide use with solving certain bugs in their final submission during Fall 2017 and Spring 2018 semesters. Note that the totals in the bottom row double-count students, which makes sense only if you believe the incidences of each bug in a given submission are independent. 78

6.1 Landslide's bug-finding performance on various test configurations. Iterative Deepening (§4.2), optimized for fast bug-finding, is compared against Maximal State Space mode (§3.1.2), optimized for fast verification. For each, I list the CPU-time and wall-clock time elapsed, plus the number of interleavings tested in the ultimately buggy state space until the bug was found. * marks the winning measurements between each series. Lastly, state space estimation (§3.4.3) confers a sense of the exponential explosion. 104

6.2 Transactional tests verified (or not) by Landslide. Run with `-M -X`, plus any additional reduction options listed. "Baseline DPOR" always tests every abort path, without distinguishing among failure reasons (i.e., injecting only `_XABORT_RETRY`). "Retry sets" skips equivalent success path and/or retry abort reorderings (§6.2.4); "STM" suppresses retry aborts and dynamically detects when to inject conflict aborts and so on (§6.2.2). State space estimates measured after a timeout of 10 hours (and include those 10 hours in the predicted total). 108

6.3 Number of concurrency events per iteration of each test case. Note that no test used any synchronization besides mutexes (the P2 thread API was annotated as trusted (§3.2.1) and so does not contribute to state space size). Also note that "#race" means the number of unique accesses identified as racy, rather than racing pairs (the other half of a pair might well be within a transaction, which cannot be preempted on). 109
6.4 Continuation of Table 6.2, demonstrating abstraction reduction [126] on the `avl_fixed` and `map_basic` tests by verifying HTM mutex implementations separately. Tested with STM semantics, as both lock implementations include a retry loop. 113

Chapter 1

Introduction

Motivation

Modern computer architectures have turned to increasing CPU core count, rather than clock speed, to improve processing power [99]. To take advantage of multiple cores for performance, programmers must write software to execute *concurrently* – using multiple *threads* which execute multiple parts of a program's logic simultaneously. However, when threads access the same shared data, they may interleave in unexpected ways which change the outcome of their execution. When an unexpected interleaving produces undesirable program behaviour, for example, by corrupting shared data structures, we call it a *concurrency bug*. Concurrency bugs are notoriously hard for programmers to find and debug because the specific thread interleaving required to trigger them arises at random during normal execution, and often with very low probability.

Most commonly, a programmer searches for concurrency bugs in her code by running it many times (in parallel, in serial, or both), hoping that eventually, it will run according to the particular interleaving required to expose a hypothetical bug. This technique, known as *stress testing*, is unreliable, providing no guarantee of finding the failing interleaving in any finite amount of time. It also provides no assurance of correctness: when finished, there is no way of knowing how many distinct thread interleavings were actually tested. Nevertheless, stress testing remains popular because of how easily a programmer can use it: she simply wraps her program in a loop, sets it to run overnight, and kills it if her patience runs out before it finds a bug.

Stateless model checking [50] is an alternative way to test for concurrency bugs, or to verify their absence, which provides more reliable coverage, progress, and verification than stress testing. A stateless model checker tests a program by forcing it to execute a new unique thread interleaving on each iteration of the test, capturing and controlling the randomness in a finite state space of all possible interleavings.

Unfortunately, the size of these state spaces is exponentially proportional to the size of the tested program. For even moderately-sized programs, there may be more possible ways to interleave every thread's every instruction than particles in the universe. Accordingly, a programmer who wants her test to make reasonable progress through the state space must choose a subset of ways that her threads could interleave, focusing on fully

testing that subset, while ignoring other possibilities she doesn't care about. However, it is difficult to choose a subset of thread interleavings that will produce a meaningful, yet feasible test. Until computers can automatically navigate this trade-off in some intelligent way, programmers will continue to fall back to the random approach of stress testing.

Another problem stateless model checking suffers is that certain types of programs cannot be tested without the programmer putting forth some manual instrumentation effort. For example, operating system kernels implement their own sources of concurrency and their own synchronization primitives, so the checker needs to be told how to identify and control the execution of each thread. Some expert concurrency research wizards may be willing to add manual annotations to their code, but required manual effort is a serious downside for anyone with a looming deadline, and especially so for students who are still learning basic concurrency principles.

Contribution

This thesis will solve both problems discussed above. My thesis statement is as follows:

Combining both theoretically-founded automatic reduction techniques and user-informed heuristic ones, stateless model checking can sufficiently mitigate exponential explosion to be a practical testing technique even for inexperienced users.

I have built Landslide [12], a stateless model checker for thread libraries and kernels, and I have developed some techniques for automatically choosing the best thread interleavings to test and for automatically instrumenting operating system kernels in an educational setting. This thesis will comprise three major contributions:

1. **Meaningful state spaces (Chapter 4).** I will present *Iterative Deepening*, a new algorithm for navigating the trade-off in how many preemption points to test at once. Iterative Deepening incorporates state space estimation [129] to decide on-the-fly whether each state space is worth pursuing, and uses data race analysis [125] to find new preemption point candidates based on a program's dynamic behaviour. This section will include a large evaluation of the technique, comparing its performance to three prior work approaches across 600+ unique tests. I will show that Iterative Deepening of preemption points outperforms prior work in terms both of finding bugs quickly and of completely verifying correctness when no bug exists.
2. **Educational use (Chapter 5).** For the past five semesters, I have offered a fully-automated version of Landslide to students in 15-410, CMU's undergraduate Operating System Design and Implementation class [35, 36], for use as a debugging aid during the thread library project. Recently I have also extended Landslide to handle Pintos kernel projects from other universities [114]. In the two most recent semesters, I collaborated with Operating Systems course staff at two such schools, the University of Chicago and University of California at Berkeley, to provide debugging feedback to their students.

At all three universities I then collected statistics on the numbers and types of bugs found, and surveyed students to understand the human experience, This section will present the study's results to evaluate the suitability of stateless model checking in an educational setting.

3. **Hardware Transactional Memory (HTM) (Chapter 6).** HTM is a relatively new concurrent programming technique [32, 59] which is not yet addressed by modern model checkers. I have extended Landslide's concurrency model to support HTM's execution semantics and several advanced features, and tested several "real-world" HTM programs and benchmarks. This section will discuss the theoretical techniques I used to model the new form of concurrency, present associated correctness proofs of my approach, and show the verification results.

Organization

The rest of this dissertation is organized as follows.

- **Background:** Chapter 2 will present the requisite background material on concurrent programming, stateless model checking, and the various types of programs targeted by Landslide.
- **Landslide:** Chapter 3 explains the design and implementation of Landslide and all the special features it's been equipped with over the years. It is the foundation upon which all three contributions above build.
- **Quicksand:** Chapter 4 presents the Iterative Deepening framework which more intelligently chooses which state spaces to test, corresponding to contribution 1 above.
- **Education:** Chapter 5 discusses my evaluation of Landslide in CMU's 15-410 class environment using the Pebbles kernel, and in the University of Chicago's and Berkeley's OS class environments using the Pintos kernel, corresponding to contribution 2 above.
- **Transactional Memory:** Chapter 6 presents my extension of Landslide's concurrency model to handle transactional concurrency and the evaluation thereof, corresponding to contribution 3 above.
- **Related Work:** Chapter 7 honors my neighbours and ancestors in research spirit.
- **Future Work:** Chapter 8
- **Conclusion:** Chapter 9 provides some thoughts on the future of the field.

Notes on reading this thesis

I have tried to make this document accessible to readers of all programming experience levels, although some of the research being theoretical and several layers of abstractions deep, I cannot promise all easy reading. Chapter 2, Background, provides what I hope are friendly concrete examples to help the reader feel comfortable with each level of intuition that upcoming algorithms will build upon. These should suffice for the experiments

and overall contributions, if not necessarily the details of each algorithm or soundness proof. In particular, Chapter 5, Education, may be approached with no knowledge of concurrency or model checking, taking it merely as a study of a magic new debugging tool in the classroom setting. The more ambitious reader may proceed to the Landslide chapter's algorithm walkthroughs (§3.4), which should equip them to understand every detail herein. Readers who are here only to skim and skip around should at least be aware of the glossary (§2.5) to help clarify any terminology confusion.

Color will be used in figures and graphs to add visual contrast and make the data easier to navigate at a glance, but only in redundant ways also signaled by symbols. I have made some effort to choose palettes friendly to color-blindness; should the reader find contrasts too low anyway, whether being color-blind or reading a physical copy printed in greyscale, they may be assured all important distinctions still render in monochrome. For example, ovals and rectangles typically depict different threads, and † distinguishes state space estimates from completed verifications.

Pronoun use will vary between more specific and more ambiguous to convey additional nuance. The singular "I" is associated with my own research contributions, while the royal "we" should be taken to include the reader, such as when surveying background material or related work, to which the author and reader share more similar relationships. The impersonal "the programmer" will be referred to as she/her to highlight her role as the intended user, separate from the underlying research, and also to challenge readers' unconscious bias about gender in computer science. Individual students who participated in the user studies will be given the more inclusive they/them. Gender-neutral pronouns will also be used on the author themself.

This document is, in a way, only half the work of the thesis, the other half being Landslide's implementation. While some readers may prefer to be taught in prose and/or mathematical notation how an algorithm works, others may find that disorienting and wish to see things in a way a compiler would understand. The beginning of Chapter 3 provides source code links, and the rest of it serves as a guide to browsing the repository. Later chapters will often make parenthetical references to specific files and/or functions therein which implement a feature under discussion.

Chapter 2

Background

This chapter will introduce the necessary background material on concurrency, stateless model checking, data-race analysis, and the relevant undergraduate operating systems classes.

2.1 Concurrency

2.1.1 The Basics

Modern software often turns to multithreading to improve performance. In a multithreaded program, multiple execution units (or *threads*) execute the same or different sections of code simultaneously. This can provide speedups up to a factor of the number of threads running in parallel, but may also provide surprising execution results.

Simultaneity

This simultaneity of threads is achieved either by executing each one on a separate CPU, or by interleaving them nondeterministically (as controlled by clock interrupts) on the same CPU. Because clock interrupts can occur at any instruction[1], we consider single-CPU multithreading to be simultaneous at the granularity of individual instructions. Likewise, when multiple CPUs access the same memory, hardware protocols generally ensure that the events of a single instruction are executed atomically from the perspective of all CPUs. Although there are some exceptions – unlocked memory-to-memory instructions, unaligned writes [89], and weak memory consistency models [5] – we model multicore concurrency the same way as above, deferring these exceptions beyond the scope of this work. We refer to an execution trace depicting the global sequence of events as a *thread interleaving* or *schedule*.

[1] With some exceptions in kernel-level programming, discussed later.

Shared state

When a programming language offers multithreaded parallelism but forbids access to any shared state between threads [95], the simultaneity of threads is largely irrelevant to the program's behaviour. However, "thread-unsafe" languages such as C, C++, Java, and so on remain popular, in which threads may access global or heap-allocated variables and data structures with no enforced access discipline. The behaviour of such programs is then subject to the manner in which these accesses interleave.

2.1.2 Identifying bugs

Even if a program's behaviour is nondeterministic, that does not necessarily mean it has a bug. After all, many programs use random number generation to intentionally generate different outputs. We say a *concurrency bug* occurs when one or more of a program's nondeterministic behaviours is both *unanticipated* and *undesired*. Most often, a concurrency novice who programs with shared state will consider the possible interleavings where one thread's access sequence occurs entirely before the other's, but neglect to consider intermediate outcomes in which the threads' access sequences are interleaved.

Consider the program in Figure 2.1: Any output between 2 and 2000 is possible[2], but whether this constitutes a bug is a matter of perspective. Was the program written to count to 2000, or was it written to compute a randomized distribution? In this thesis, we make no attempt to reason about the "intent" of programs, so we further restrict *concurrency bug* to denote a program behaviour which is mechanically identifiable, according to commonly-accepted notions of what programs behaviours are always bad. Bug conditions include assertion failures, memory access errors (i.e., segmentation fault or bus error), heap errors (i.e., use-after-free or overflow), deadlocks, and infinite loops (which must be identified heuristically [136]).

2.1.3 Concurrency Primitives

To prevent unexpected interleavings such as the example in Figure 2.1(b), most concurrent programs use *concurrency primitives* to control which interleavings are possible. Controlling nondeterminism is not typically provided by any features of programming languages themselves; rather, it is achieved via special atomicity mechanisms provided by the CPU and/or operating system – hence the term "primitive". For example, x86 CPUs provide the `xchg` instruction, which performs both a read and subsequent write to some shared memory, with no possibility for other logic to interleave in between. Using such atomic instructions as building blocks, concurrency libraries provide abstractions for controlling nondeterminism in several commonly-desired ways. These include *locks*, *descheduling*, *condition variables*, *semaphores*, *reader-writer locks*, and *message-passing*.

Each such abstraction provides certain semantics about what thread interleavings can arise surrounding their use. When building a tool for testing concurrent programs, one

[2] Fun exercise for the reader: Show why 2 is a possible output, but 1 is not!

```
int x;
void count() {
    for (int i = 0; i < 1000; i++)
        x++;
}
void main() {
    tid1 = thr_create(count);
    tid2 = thr_create(count);
    thr_join(tid1);
    thr_join(tid2);
    printf("%d\n", x);
}
```

(a) Source listing for a multithreaded program which might count to 2000.

Thread 1	**Thread 2**
`load tmp <- x;`	
	`load tmp <- x;`
	`add tmp <- 1;`
	`store x <- tmp;`
`add tmp <- 1;`	
`store x <- tmp;`	

(b) Example interleaving of the compiled assembly for (a), in which 2 concurrent iterations of the loop yield 1 net increment of `x`.

Figure 2.1: Example concurrent program in which simultaneous accesses to shared state may interleave to produce unexpected results.

may include some computational understanding of the behaviour of any, or all, of these abstractions. Annotating a certain abstraction's semantics treats it as a trusted concurrency primitive in its own right, and allows the testing tool to reduce the possible space of interleavings (or the set of false positive data-race candidates reported, etc.), at the cost of increasing the implementation and theoretical complexity of the analysis. This thesis will consider locks, descheduling, and transaction begin/end as the only concurrency primitives, and assume the others listed above are implemented using those as building blocks (an exercise for the reader [36]).

Locks (or *mutexes*, short for "mutual exclusion locks") are objects, shared by multiple threads, which allow the programmer to mark certain *critical sections* of code that must not interleave with each other. When one thread completes a call to `mutex_lock(mp)`, all invocations by other threads on the same `mp` will wait (or "block") until the corresponding `mutex_unlock(mp)`. Figure 2.2(a) shows how a yielding mutex (not the best implementation, but the simplest) may be implemented using `xchg`, and (b) shows how a mutex may be used to fix the example from Figure 2.1.

2.1.4 Transactional Memory

Critical sections of code must be protected from concurrent access, even when it's not known in advance whether the shared memory accesses between threads will actually conflict on the same memory addresses. The concurrency primitives discussed above take a pessimistic approach, imposing a uniform performance penalty (associated with the primitives' implementation logic) on all critical sections, whether or not a conflict is likely. Some implementations may be optimized for "fast paths" in the absence of contention, but must still access shared memory in which the primitive's state resides.

Transactional memory [60] offers a more optimistic approach: critical sections of code

```
typedef struct mutex {
    volatile int held;
    int owner;
} mutex_t;
void mutex_lock(mutex_t *mp) {
    while (xchg(mp->held, 1))
        yield(mp->owner);
    mp->owner = gettid();
}
void mutex_unlock(mutex_t *mp) {
    mp->owner = -1;
    mp->held = 0;
}
```

(a) A simple mutual exclusion lock built using the `xchg` instruction.

```
int x;
mutex_t m;
void count() {
    for (int i = 0; i < 1000; i++) {
        mutex_lock(&m);
        x++;
        mutex_unlock(&m);
    }
}
```

(b) The `count` function from Figure 2.1, adjusted to use a mutex to ensure each increment of `x` is uninterruptible.

Figure 2.2: Using a locking primitive to protect accesses to shared state.

are marked as "transactions", analogously to locking a mutex, and allowed to speculatively execute with no protection. If a conflict between transactions is detected, the program state is rolled back to the beginning of the transaction, and a backup code path may optionally be taken. Consequently, no intermediate state of a transacting thread is ever visible to other threads; all changes to memory within a transaction become globally visible "all at once" (or not at all). This method optimizes for a common no-contention case of little-to-no overhead, pushing extra both code and implementation complexity to handling conflicts.

Transactional memory (TM) may be implemented either in hardware, using special instructions and existing cache coherence algorithms, or in software, via library calls and a log-based commit approach. Software transactions (STM) [4] can be used on any commodity processor, but must impose runtime overhead associated with logging. Hardware transactions (HTM) [68] achieve better performance by reusing existing cache coherence logic to detect conflicts, but require explicit support from the CPU, which is not yet widespread. Haswell [59] is the first x86 architecture to support HTM, offering three new instructions: `xbegin`, `xend`, and `xabort`, to begin, commit, and fail a transaction, respectively. The example program in Figure 2.3 demonstrates how these primitives can be used to synchronize a simple shared access without locking overhead in the common case[3], using GCC's compiler intrinsics [47].

Concerning possible execution patterns, the main difference between STM and HTM is the circumstances under which a transaction may abort. A software-backed transaction will abort if and only if a memory conflict occurs therein with another thread. HTM, however, is backed by the CPU's cache, and is therefore subject to other circumstances such as cache capacity or interrupt-triggered cache flushes which may force an abort even when no memory conflict occurs. Chapter 6 will explore the consequences of this difference

[3] The solution presented here is actually incomplete; stay tuned until Chapter 6 for the surprising twist!

```
void count() {
    for (int i = 0; i < 1000; i++) {
        if ((status = _xbegin()) == _XBEGIN_STARTED) {
            x++;
            _xend();
        } else {
            mutex_lock(&m);
            x++;
            mutex_unlock(&m);
        }
    }
}
```

Figure 2.3: The example `count` routine from Figure 2.2, rewritten to use HTM. If the transaction in the top branch aborts, whether from a memory conflict or random system interrupt, execution will revert to the return of `_xbegin`, `status` will be assigned an error code indicating the abort reason, and control will drop into the `else` branch. The programmer can then use explicit synchronization, such as a mutex, to resolve the conflict.

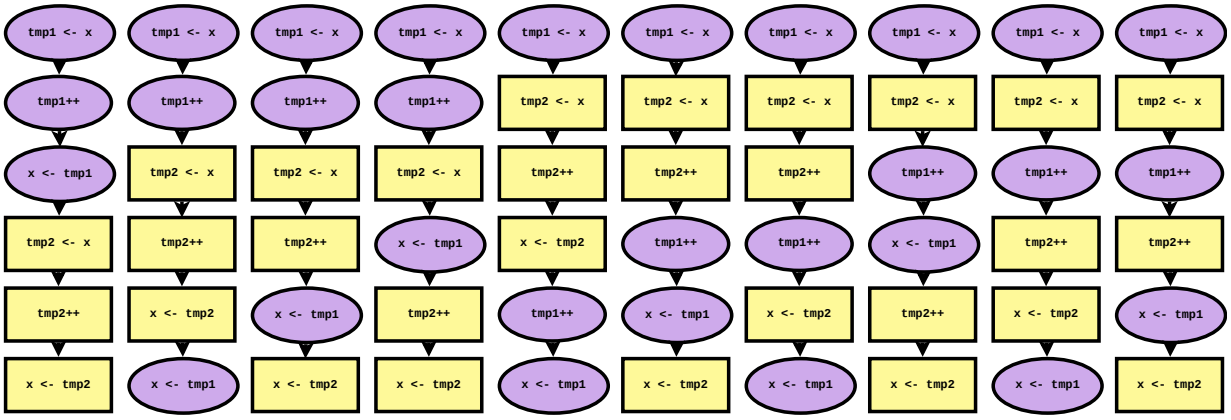
further. This thesis will focus on HTM as my platform for testing transactional programs, to highlight the importance of researching advanced testing techniques in anticipation of upcoming hardware features.

2.2 Stateless Model Checking

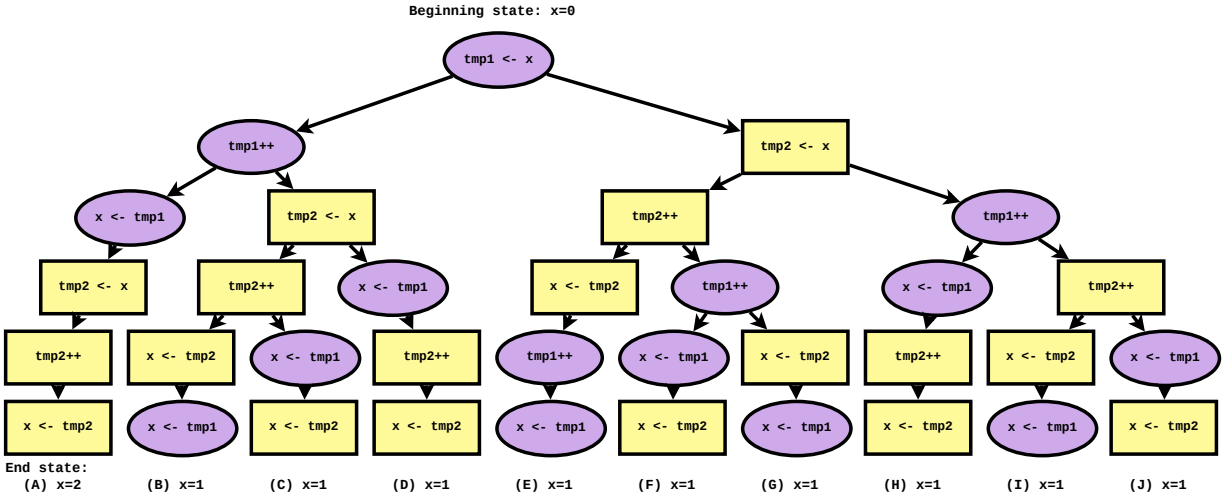
Model checking [50] is a testing technique for systematically exploring the possible thread interleavings of a concurrent program. A model checker executes the program repeatedly, each time according to a new thread interleaving, until all interleavings have been tested or the CPU budget is exhausted. During each execution, it forces threads to execute serially, thereby confining the program's nondeterminism to scheduler thread switches. It then controls the scheduling decisions to guarantee a unique interleaving is tested each iteration.

2.2.1 The state space

To understand what it means to exhaustively test all possible thread interleavings, one must define the possible execution sequences as a finite *state space*. To visualize this, using a single iteration of the `x++;` loop from Figure 2.1 as an example, with `x++;` expanded into its three corresponding pseudo-assembly instructions, Figure 2.4(a) shows all possible execution interleavings between 2 threads.



(a) Interleavings visualized individually, as a list.



(b) Interleavings (same order as in (a)), with common prefixes combined as “preemption points”, forming a tree.

Figure 2.4: Visualization of interleaving state space for the program in Figure 2.1. Thread 1 is represented by purple ovals, thread 2 by yellow squares, and time flows from top to bottom. As the two threads execute the same code, without loss of generality thread 1 is fixed to run first – the full state space is twice the size, and the other half is symmetric to the one shown.

Static versus dynamic analysis

Model checking is a *dynamic* program analysis, meaning that it observes the operations and accesses performed by the program as its code is executed. In contrast, *static* program analyses check certain properties at the source code level. Static analyses are ideal for ensuring certain standards of code quality, which often correlates with correctness, but cannot decide for certain whether a given program will fail during execution without actually running the code [51]. Static analyses face the challenge of *false alarms* (or *false positives*): code patterns which look suspicious but are actually correct. A debugging tool which reports too many false alarms will dissuade developers from using it [41]. Dynamic analysis, our approach, identifies program behaviours that are definitely wrong, so each bug report is accompanied by concrete evidence of the violation. Assertions, segfaults, use-after-free of heap memory, and deadlock are examples of such failures we check for, although a checker may also include arbitrary program-specific predicates.

Preemption points

During execution, a model checker identifies a subset of the program's operations as "interesting", i.e., where interrupting the current thread to run a different one is likely to produce different behaviour. These so-called *preemption points* may be identified by any combination of human intuition and machine analysis. Typical preemption points include the boundaries of synchronization APIs (e.g., `mutex_lock`) or accesses to shared variables. Considering that at each preemption point multiple threads exist as options to run next, the set of possible ways to execute the program can be viewed as a tree. Figure 2.4(b) shows a visualization of the corresponding tree from our example program, using each pseudo-assembly instruction as a preemption point.

The number of preemption points in each execution defines the depth of this tree, and the number of threads available to run defines the branching factor. Hence, in a program with n preemption points and k threads available to run at each, the state space size is $O(n^k)$. Nevertheless, to fully test all of a program's possible behaviours, we must check the executions corresponding to every branch of the tree. Addressing the scaling problem in this exponential relation is the central research problem for all model checkers.

Some model checkers explicitly store the set of visited program states as a means of identifying equivalent interleavings [61]. From the perspective of such tools, state spaces such as these wherein equivalent states may be reached by multiple paths are represented as a directed acyclic graph (DAG) instead of as a tree. This approach is called *stateful* model checking. This thesis focuses on *stateless* model checking (and execution trees, not DAGs), which instead analyzes the sequence of execution events to avoid a prohibitive memory footprint. Henceforth "stateless model checking" will be abbreviated simply as "model checking" for brevity. Also, the term "state space" was originally coined to refer to the stateful approach's emphasis on the DAG's nodes (i.e., program states); while stateless checkers emphasize the tree's branches (i.e., execution sequences) instead, I will continue to use "state space" for consistency with prior work.

2.2.2 On the size of state spaces

At its essence, stateless model checking research is a perpetual struggle to become more and more efficient in order to test and verify bigger and bigger programs. But whence this efficiency? Techniques for coping with the exponential explosion fall into two categories: (1) removing redundant interleavings from the state space when we can prove they are equivalent to some interleaving already tested, or **reduction techniques**, and (2) prioritizing interleavings judged as more likely to contain bugs should bugs exist in case we are unable to exhaustively test all interleavings after all, or **search heuristics**.

Reduction techniques

Dynamic Partial Order Reduction [46] (henceforth, DPOR) is the most popular algorithm for mitigating the exponential explosion that arises as program size increases.

Abstractly speaking: Let *independent transitions* denote a pair of executions of two threads, each from one preemption point to the next, in which there are no read/write or write/write access pairs to the same memory between threads. DPOR reduces a state space, originally exponentially-sized in the number of thread transitions, to an equivalent one (i.e., testing which suffices to check all program behaviours that could arise in the original state space) exponentially-sized in the number of *dependent* thread transitions. More technically, it identifies equivalent execution sequences according to Mazurkiewicz trace theory [96], and tests at least one execution from each equivalence class.

Concretely speaking: Figure 2.5 highlights part of an execution tree where the execution ordering of threads 1 and 2 are swapped, and each interleaving has a respective "subtree" (i.e., possible interleavings given the fixed execution prefix leading up to it). The specifics of execution before the thread 1/thread 2 sequence, other possible threads to run instead of threads 1 or 2, and what logic the program executes in those subtrees are all presumably arbitrary. In these two highlighted branches, if the transitions of threads 1 and 2 are *independent*, DPOR deduces that the subsequent program states (indicated by the red arrow) are equivalent. Thence, only one of the two interleavings and its respective subtree needs to be executed in order to check all possible program states. §3.4.2 explains how DPOR implements such a deduction in more detail.

Over the years, researchers have developed many enhancements to DPOR, such as Optimal DPOR [1], parallelizable DPOR [130], SAT-directed model checking [31], Maximal Causality Reduction [63], and DPOR for relaxed memory architectures [147].

Search heuristics

However, even though DPOR can prune an exponential number of redundant interleavings, the state space size is still exponential in the number of *dependent* (conflicting) interleavings. Developers will always want to test larger and larger programs, so no matter the quality of our reduction algorithm, we must accept that some tests will be too large to be fully tested in a reasonable time. Hence, recent model checking research has turned to heuristic techniques for achieving further reduction, optimizing the search to try to

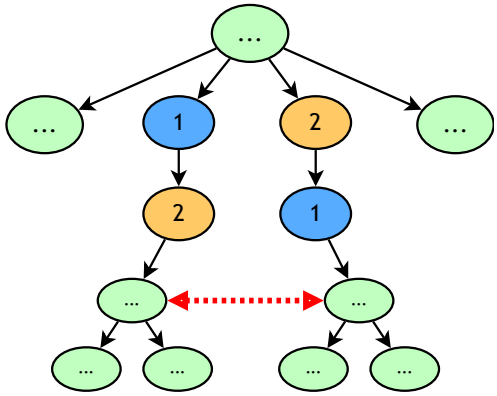


Figure 2.5: DPOR identifies independent transitions by different threads which can commute without affecting program behaviour. Here, if the transitions marked 1 and 2 have no shared memory conflicts, the states marked with the red arrow are guaranteed identical. Hence, only one of the subtrees need be explored.

uncover bugs faster (should they exist) at the expense of possibly missing other bugs, or missing the chance to complete a full verification.

Iterative Context Bounding [101] is a popular such technique which heuristically reorders the search to prioritize interleavings with fewer preemptions first. This heuristic is based on the insight that most bugs require few preemptions to uncover, so interleavings with a number of preemptions that exceeds a certain bound will be de-prioritized, only tested until after all the fewer-preemption interleavings are completed. Preemption sealing [9] is another heuristic strategy which restricts the scope of the search by limiting the model checker to use only preemption points arising from certain functions in the source code. This allows developers to vastly reduce state space size by identifying which program modules are already trusted, although it requires some human intuition to correctly mark those boundaries. Iterative Deepening, presented in Chapter 4, is another such search heuristic.

2.3 Data Race Analysis

2.3.1 Definition

Data race analysis [120] identifies pairs of unsynchronized memory accesses between threads. Two instructions are said to race if:

1. they both access the same memory address,
2. at least one is a write,
3. the threads do not hold the same lock,
4. and no synchronization enforces an order on the thread transitions (the *Happens-Before* relation, described below).

```
   int x = 0; bool y = false; mutex_t mx;

   Thread 1                Thread 2
1  x++; // A1
2  mutex_lock(&mx);
3  mutex_unlock(&mx);
4                          mutex_lock(&mx);
5                          mutex_unlock(&mx);
6                          x++; // A2
```

(a) True potential data race.

```
   Thread 1                Thread 2
1  x++; // B1
2  mutex_lock(&mx);
3  y = true;
4  mutex_unlock(&mx);
5                          mutex_lock(&mx);
6                          bool tmp = y;
7                          mutex_unlock(&mx);
8                          if (tmp) x++; // B2
```

(b) No data race in any interleaving.

Figure 2.6: Data-race analyses may be prone to either *false negatives* or *false positives*. Applying Happens-Before to program (a) will miss the potential race possible between A1/A2 in an alternate interleaving, while using Limited Happens-Before on (b) will produce a false alarm on B1/B2.

In Figure 2.6, the pairs of lines marked with comments (A1 and A2, B1 and B2) race.

A data race analysis may be either *static* (inspecting source code) [41] or *dynamic* (tracking individual accesses arising at run-time) [125]. This paper focuses exclusively on dynamic analysis, so although our example refers to numbered source lines for ease of explanation, in practice we are actually classifying the individual memory access events corresponding to those lines during execution. Actually, each `x++` statement likely compiles to two separate load or store instructions, so each of those two instructions from each of the two marked source lines pairwise will race (except for the two loads, which are both reads).

2.3.2 Happens-Before

Condition 4 of the above definition expresses the notion that the access pair can be executed concurrently, regardless of whether the hardware actually carries out the operations in the same physical instant. Several approaches exist to formally representing this condition.

- Most prior work focuses on *Happens-Before* [81] as the order relation between accesses. [131] and [110] identify a problem with this approach: it cannot identify access pairs separated by an unrelated lock operation which could race in an alternate interleaving, as shown in the example program in Figure 2.6(a). We call such unreported access pairs *false negatives*.
- [110] introduces the *Limited Happens-Before* relation, which will report such potential races by considering only blocking operations like `cond_wait` to enforce the order. However, consider the similar program in Figure 2.6(b), in which the access pair ceases to exist in the alternate interleaving. Limited Happens-Before will report all potential races, avoiding false negatives [125], but at the cost of necessarily reporting some such *false positives*.
- In recent work, the *Causally-Precedes* relation [131] extends Happens-Before to additionally report a subset of potential races while soundly avoiding false positives. It tracks conflicting accesses in intervening critical sections to determine whether lock events are unrelated to a potential race. Causally-Precedes will identify the potential race in Figure 2.6(a), as the two critical sections do not conflict, although it can still miss true potential races in other cases.

Landslide implements both Happens-Before (henceforth referred to as *Pure Happens-Before* for clarity) and Limited Happens-Before. Chapter 4 includes a comparison of the two approaches for the purpose of finding new preemption points for model checking.

2.4 Education

This thesis will tackle Pebbles and Pintos, two different system architectures used in educational operating systems courses. This section describes the projects which students implement and which Landslide tests.

2.4.1 Pebbles

The Pebbles kernel architecture is used at Carnegie Mellon University (CMU) in 15-410, Operating System Design and Education [35, 36]. In the course of a semester, students work on five programming assignments; the first two are individual, and the remaining three are the products of two-person teams. I will focus on the third and fourth of these, the thread library and kernel, called "P2" and "P3" respectively (the project numbers start at 0). The other three (a stack-crawling backtrace utility, a bare-metal game with device drivers, and a small extension to the P3 kernel) are not of concern in this thesis. The course's prerequisite is 15-213, Introduction to Computer Systems [17]. Both P2 and P3 are built using the *Pebbles* system call specification, outlined in Table 2.1

System call name	Summary
	Lifecycle management
`fork`	Duplicates the invoking task, including all memory regions.
`thread_fork`	Creates a new thread in the current task.
`exec`	Replaces the program currently running in the invoking task with a new one specified.
`set_status`	Records the exit status of the current task.
`vanish`	Terminates execution of the calling thread.
`wait`	Blocks execution until another task terminates, and collects its exit status.
`task_vanish`*	Causes all threads of a task to `vanish`.
	Thread management
`gettid`	Returns the ID of the invoking thread.
`yield`	Defers execution to a specified thread.
`deschedule`	Blocks execution of the invoking thread.
`make_runnable`	Wakes up another `descheduled` thread.
`get_ticks`	Gets the number of timer ticks since bootup.
`sleep`	Blocks a thread for a given number of ticks.
`swexn`	Registers a user-space function as a software exception handler.
	Memory management
`new_pages`	Allocates a specified region of memory.
`remove_pages`	Deallocates same.
	Console I/O
`getchar`*	Reads one character from keyboard input.
`readline`	Reads the next line from keyboard input.
`print`	Prints a given memory buffer to the console.
`set_term_color`	Sets the color for future console output.
`set_cursor_pos`	Sets the console cursor location.
`get_cursor_pos`	Retrieves the console cursor location.
	Miscellaneous
`ls`	Loads a given buffer with the names of files stored in the RAM disk "file system."
`halt`	Ceases execution of the operating system.
`misbehave`*	Selects among several thread-scheduling policies.

Table 2.1: The Pebbles specifcation defines 25 system calls. Students are not required to implement ones marked with an asterisk (*), though the reference kernel provides them.

P2

The thread library project [36] has two main components: implementing concurrency primitives, and implementing thread lifecycle and management routines. The required concurrency primitives are as follows:

- Mutexes, with the interface `mutex_lock(mp)` and `mutex_unlock(mp)`, whose functionality is described earlier this chapter. Students may use any x86 atomic instruction(s) they desire, such as `xchg`, `xadd`, or `cmpxchg`, and/or the `deschedule`/ `make_runnable` system calls offered by the reference kernel.
- Condition variables, with the interface `cond_wait(cvp, mp)`, `cond_signal (cvp)`, and `cond_broadcast(cvp)`. `cond_wait` blocks the invoking thread, "simultaneously" releasing a mutex which protects some associated state (atomically, with respect to other calls to signal or broadcast under that mutex). `cond_signal` and `cond_broadcast` wake one or all waiting threads. Students must use the `deschedule` and `make_runnable` system calls to implement blocking (busy-waiting is forbidden), and typically include an internal mutex to protect the condition variable's state as well. The primary challenge of this exercise is ensuring the aforementioned atomicity between `cond_wait`'s unlock and deschedule, with respect to the rest of the interface.
- Semaphores, with the interface `sem_wait(sp)` and `sem_signal(sp)` (sometimes called *proberen* and *verhogen* in other literature). The semaphore can be initialized to any integer value; if initialized to 1, it behaves like a mutex. Students typically implement semaphores using mutexes and condition variables, not using atomic instructions or system calls directly.
- Reader-writer locks (rwlocks), with the interface `rwlock_lock(rwp, mode)` and `rwlock_unlock(rwp)`. `mode` may be either `RWLOCK_READ` or `RWLOCK_WRITE`. Behaves as mutexes, but multiple readers may access the critical section simultaneously. Students typically implement rwlocks using mutexes and condition variables, not using atomic instructions or system calls directly.

The interface to each also includes an associated `_init()` and `_destory()` function.

The thread lifecycle/management routines are as follows:

- `thr_init(stack_size)` initializes the thread library, setting a default stack size to be allocated to new threads.
- `thr_create(child_func, child_arg)` spawns a new thread to run the specified function with the specified argument. There is a semantic gap between this function and the `thread_fork` system call (which takes no parameters, makes no changes to the user's address space, and cannot meaningfully be invoked from C code) which students must bridge. Returns an integer thread ID of the newly created thread.
- `thr_exit(status)` aborts execution of the calling thread, recording an exit status value. The main challenge of this function is to allow another thread to free the memory used for the exiting thread's stack, without risking any corruption as long as the exiting thread continues to run.
- `thr_join(tid, statusp)` blocks the calling thread until the thread with the speci-

fied thread ID exits, then returns, collecting its exit status.

Other than `thr_init` (which is necessarily single-threaded), several concurrency errors between any two (or all three) of these functions are very common in student submissions.

Finally, students also implement automatic stack growth using the `swexn` system call, which is not relevant to this thesis.

P3

In P3, students implement a kernel which provides the same system calls shown in Table 2.1, previously provided by the reference kernel. Pebbles adopts the Mach [3] distinction between *tasks*, which are resource containers, and *threads*, each of which executes within a single task. This requires less implementation complexity than the more featureful Plan 9's `rfork` [115] or Linux's `clone` models.

Although the internal interfaces are not mandated like they were in P2, all Pebbles kernels must necessarily contain the same abstract components. These include:

- A round-robin scheduler, including context switching, timer handling, and runqueue management;
- Some approach to locking, often analogous to P2's concurrency primitives (henceforth referred to as "kernel mutexes"), ll and some approach to blocking threads indefinitely;
- A virtual memory implementation, including a program loader;
- Lifecycle management code for creation and destruction of kernel threads and processes;
- Other miscellany such as a suite of fault handlers to ensure no user program can cause the kernel itself to crash.

Because any combination of system calls or fault handlers can be invoked by user programs simultaneously, concurrency bugs can arise from the interaction of any subset of kernel components with each other. The most common bugs studence face arise from the interaction of some component with itself (e.g., concurrent invocations of `new_pages`/`remove_pages` in the same process), or from the interaction between an exiting thread and some other thread trying to communicate with it (`vanish` versus, well, anything else, really). The most difficult concurrency problem in P3 is that of coordinating a parent and a child task that simultaneously exit: when a task completes, live children and exited zombies must be handed off to the task's parent or to the `init` process, when the task's parent may itself be exiting; meanwhile, threads in tasks that receive new children may need to be awakened from `wait`. Careless solutions to this problem are prone to data races or deadlocks.

Secrecy

The 15-410 course staff is notoriously secretive about the nature of many concurrency bugs students commonly encounter during P2 and P3. This is driven by a desire to cause students to find, diagnose, and fix these bugs on their own during the projects, rather than to be surprised by them afterwards during grading [37]. One such example is the

`paraguay` unit test distributed with P2 (§5.1.3), which targets a subtle condition-variable bug. The test uses the `misbehave` system call to target a particular thread interleaving likely to expose the bug which is otherwise very unlikely to arise in normal execution. The reference kernel specification [35] does not define the `misbehave` modes' behaviours, as doing so would deprive students of the learning experience of discovering the interleaving in question on their own. I will occasionally use intentionally vague phrasing to preserve the mystery of these bugs.

Use at other universities

In the Spring 2018 semester, the Operating Systems class at Penn State University (henceforth CMPSC 473 and PSU, respectively) offered the P2 thread library project as part of its curriculum. Students in this class implement P2 on a 6 week project timeline (compared to 2 weeks at CMU), work alone rather than in pairs, skip the `swexn` automatic stack growth portion, and rather than running their code with a reference Pebbles kernel binary in a simulator, use the Pebwine emulation layer [132] to run Pebbles-compatible program binaries in the Linux userspace. Otherwise, the project is identical to CMU 15-410's P2.

2.4.2 Pintos

The Pintos kernel architecture [114] is used at several universities, including Berkeley, Stanford, and the University of Chicago. The Pintos basecode implements a rudimentary kernel, consisting of a context switcher, round-robin scheduler, locking primitives, and program loader. upon which students add more features in several projects. Most relevant to this thesis, the basecode provides the following functions/libraries, among others:

- Semaphores (the basic concurrency primitive, implemented using direct scheduler calls): `sema_up`, `sema_down`, `sema_try_down`;
- Locks (which wrap a semaphore initialized to 1), `lock_acquire`, `lock_release`, `lock_try_acquire`;
- Condition variables (also implemented using scheduler calls): `cond_wait`, `cond_signal`, `cond_broadcast`, with the same semantics as Pebbles P2 condvars;
- Basic round-robin scheduling facilities: `thread_block` (a kernel-level analogue to Pebbles's `deschedule`), `thread_yield`
- Kernel thread lifecycle management, `thread_create` and `thread_exit`, including stack space memory management;
- Interrupt and fault handlers;
- A page allocator, `palloc_get_page`, `palloc_get_multiple`, `palloc_free_page`, and `palloc_free_multiple`

Both Pebbles and Pintos basecodes offer a standard C library including `malloc`, string-formatting, printing, etc.

Although there is some variety in supplemental assignments, all Pintos courses include three core projects building on the Pintos basecode:

- *Threads*: Students must implement an "alarm clock" (analogous to Pebbles's `sleep` system call), a priority scheduling algorithm, and a multi-level feedback queue scheduler.
- *Userprog*: Provided with rudimentary virtual memory and ELF loader implementations, students must implement argument passing and several system calls associated with userspace programs, including `exec`, `exit`, `wait`, and file descriptor management.
- *Filesys*: Provided with a simple "flat" filesystem implementation, students must extend it with a buffer cache, extensible files, and subdirectories.

Some schools further offer a virtual memory project, extending the provided VM with a frame table and supplemental page table and fault handler [56, 113], or supplemental HTTP server and `malloc` assignments [70]. Being largely architectural/algorithmic projects rather than concurrency-oriented ones, this thesis is not concerned with these assignments. The main concurrency challenges in Pintos projects arise from the *threads* and *userprog* assignments: implementing a correct `alarm` routine, ensuring the priority scheduler remains safe in the presence of concurrent threads of the same priority, and designing correct interactions between the `wait` and `exit` system calls.

2.5 Glossary

This section provides a convenient reference of terminology used throughout the thesis.

1.

Chapter 3

Landslide

Landslide is a model checker implemented as a plug-in module for x86 full-system simulators. The program to be tested runs in a simulated environment, and Landslide uses its access to the simulator's internal state to inspect and manipulate the memory and thread scheduling of the program as it executes. As of this thesis's writing, Landslide supports the use of two possible simulators:

- **Simics** [92], a proprietary simulator licensed commercially by Wind River, used at CMU in 15-410 to run Pebbles thread libraries and kernels, and
- **Bochs** [83], an open-source (LGPL) simulator used at the University of Chicago, Berkeley, Stanford, and other schools to run Pintos kernels.

The Bochs port of Landslide is likewise open-source and available at `https://github.com/bblum/landslide`. The HEAD commit at the time of writing is 5d45e2c. The Simics port uses Simics's proprietary API and is hence unlicensed and available upon request for educational use only. Development on the Simics port is largely frozen, as the Bochs port implements all the same features and more, and is also roughly 3x faster.

This chapter will discuss Landslide's outer and inner workings in all their gory detail. It is intended for the aspiring developer or the ambitious user and hence unlike other chapters is written in the style of documentation rather than as a report of research results. The reader interested only in a theoretical introduction to model checking's foundational algorithms, with detailed and friendly examples to help establish intuitions the later chapters may require, may skip to §3.4.

3.1 User interface

This section describes the features of Landslide the average student user should expect to interact with. Separate user guides also exist, described in Chapter 5.

3.1.1 Setup

Three setup scripts are provided, one for each supported kernel architecture: `p2-setup.sh`, `psu-setup.sh`, and `pintos-setup.sh`. The user should supply the directory containing her project implementation. The second of the three is largely the same as the first, with CMU-specific project details replaced by PSU-specific ones. The latter of the three also supports arguments specifying which of the Pintos projects to target. For example:

- `./p2-setup.sh /path/to/my/p2`
- `./psu-setup.sh /path/to/my/thrlib`
- `./pintos-setup.sh /path/to/my/threads` (2nd argument defaults to "`threads`")
- `./pintos-setup.sh /path/to/my/userprog userprog`

These scripts accomplish the following setup tasks (among other trivialities):

- Copy the user's code into `pebsim/p2-basecode/` or `pebsim/pintos/`, which contain a pre-annotated Pebbles reference kernel binary or pre-annotated Pintos basecode, respectively.
- Build the code in its new location.
- Run the instrumentation script on the resulting binary to let Landslide know where all the important functions are (see §3.3.9).

3.1.2 Running Landslide through Quicksand

The preferred method of invoking Landslide is through Quicksand, the Iterative Deepening wrapper program which has all of Chapter 4 to itself. This is done via the `./landslide` script in the top-level directory, which:

- Checks if the user needs to run `*-setup.sh` again, in case her source code was more recently updated than the existing annotated build (a common mistake),
- Passes its arguments through to `id/landslide-id`, the Quicksand binary, and
- (If during the student user study,) compresses the resulting log files, creates a snapshot tarball of them and the current version of the user's code, and sends it to me for nefarious research purposes.

Command-line argments

The following command line arguments are recommended for the common user.

- `-p PROGRAM`: the name of the test case to invoke
- `-t TIME`: wall-clock time limit, in seconds; or suffixed with one of `ydhms` for years, days, hours, minutes, or seconds respectively (default 1h)
- `-c CPUS`: maximum number of Landslide instances to run in parallel (defaults to half the number of system CPUs)
- `-i INTERVAL`: interval of time between printing progress reports (default 10s)
- `-d TRACEDIR`: directory for resulting bug traces (default current directory)

- `-v`: verbose mode (issues output for each executed interleaving by each instance of landslide, makes progress reports more detailed, etc)
- `-l`: leave Landslide log files from completed state spaces even when no bug was found (deleted automatically by default)
- `-h`: print help text and exit immediately

The following "secret" arguments also exist, primarily for my own use in running experiments or debugging.

- `-C`: enable "control experiment" mode, i.e., run only 1 instance of Landslide, with all (non-data-race) preemption points enabled in advance
- `-I`: enable Iterative Context Bounding (requires `-C`, although future work may relax this restriction); this generally causes bugs to be found faster should they exist, but degrades completion time (§3.4.5)
- `-O`: enable Preempt-Everywhere mode (§4.3, requires `-C`)
- `-M`: enable Maximal State Space mode, which prioritizes the maximal state space to optimize for fast verification, abandoning all subset jobs even if they might find bugs faster (§6.3, incompatible with `-C`). According to §4.2.1's soundness proofs, this is equivalent to `-O` (and according to my experience, way faster as well).
- `-H`: use Limited Happens-Before for data-race analysis (§2.3.2) (default for Pebbles kernelspace mode)
- `-V`: use vector-clock-based Pure Happens-Before for data-race analysis (§2.3.2) (default for P2/PSU userspace and Pintos modes)
- `-X`: support transactional memory (Chapter 6)
- `-A`: support multiple abort codes during transaction failure (§6.2); required for testing programs which behave differently under different abort circumstances, but impacts the state space size
- `-S`: suppress retry aborts during transaction failure (§6.2)
- `-R`: enable retry-set state space reduction for transactional tests (§6.2)
- `-P`: support Pintos architecture (enabled automatically when `pintos-setup.sh` is run)
- `-4`: support Pebbles architecture (enabled automatically when either `p2-setup.sh` or `psu-setup.sh` is run)
- `-e ETAFACTOR`: configure heuristic state space ETA deferring factor (described in detail in `id/option.c`)
- `-E ETATHRESH`: configure heuristic threshold of state space progress for judging ETA stability (described in detail in `id/option.c`)

Quicksand will automatically generate configuration files and invoke Landslide according to the process described in the next section.

3.1.3 Running Landslide directly

Rather than letting Quicksand juggle multiple instances of Landslide, the user may run a single instance directly, optionally configuring the preemption points by hand. This is recommended only for the enthusiastic user annotating her own kernel.

The script `pebsim/landslide` invokes Landslide thus. It should be run from within the `pebsim/` directory. When supplied no arguments, it reads configuration options from `pebsim/config.landslide` (a bash script expected to define certain variables as described in §3.3.9). The user may optionally specify a file containing additional config directives as an argument.[1] Such supported options are as follows.

Dynamic configuration options

First, the following options may be changed without triggering a recompile of Landslide. They are implemented as bash functions defined in `pebsim/build.sh`.

- `within_function FUNC` - adds `FUNC` to a whitelist of functions required to appear in the current stack trace before identifying a preemption point (see §3.4.1)
- `without_function FUNC` - as above, but a blacklist instead of a whitelist
- `within_user_function FUNC` - as two above but finds the function in the userspace test program rather than the kernel code.
- `without_user_function FUNC` - difference to two above same as stated one above.
- `data_race ADDR TID LAST_CALL CURRENT_SYSCALL` - specifies a data-race preemption point.
  - `ADDR` shall be the code address (in hex) of the racing address, *before* the execution of which a preemption will be issued.
  - `TID` indicates a thread ID required to be running for this data race. To specify data race PPs across all threads at once, set `FILTER_DRS_BY_TID=0` (see next section).
  - `LAST_CALL` indicates a code address required to be the site of the last `call` instruction executed (similar to specifying a stack trace, but using a full stack trace here degrades performance too much), or 0 to not use this feature. From personal experience I found this option rather useless and recommend always supplying 0. For further discussion see §4.1.
  - `CURRENT_SYSCALL` indicates the system call number if a user-space data race comes from within a kernel system call which accesses user memory (Pebbles only). Usually 0 (i.e., not in kernel code) but `deschedule`'s system call number is common as well.

[1] Quicksand actually supplies two such files as arguments: one "static" config file and one "dynamic" config file. The former contains options which require recompiling Landslide (e.g., whether or not to use ICB is controlled by an `#ifdef` in Landslide's code), while the latter contains options which Landslide interprets at runtime (e.g., which preemption points to use). The static options do not change between Landslide instances in a single Quicksand run, avoiding long Landslide start-up times.

- `input_pipe FILENAME` - FIFO file used for receiving messages from Quicksand (e.g. to suspend or resume execution). Requires `id_magic` option to be set (next section below). The odds that a human user will find spiritual enlightenment through using this option by hand are infinitesimal.
- `output_pipe FILENAME` - as above but for sending messages.

Static configuration options

Next, configuration options which affect an `#ifdef` in Landslide and will trigger a recompile upon changing. Unless otherwise specified these are boolean flags (1 or 0) and the example value shown indicates the default used if unspecified.

- **Search algorithm options**
  - `ICB=0` - enable Iterative Context Bounding (§3.4.5); corresponds to `-I` in §3.1.2.
  - `PREEMPT_EVERYWHERE=0` - enable Preempt-Everywhere mode (§4.3); corresponds to `-O` in §3.1.2.
  - `EXPLORE_BACKWARDS=0` - configure whether, at each newly encountered preemption point, to allow the current thread to run first then later upon backtracking to preempt (0), or to issue preemptions first and then try continuing the current thread later (1). 0 tends to produce shorter preemption traces while 1 tends to find bugs faster ([12] §8.7.1). Not compatible with ICB.
- **Memory analysis options**
  - `PURE_HAPPENS_BEFORE=1` - select Pure Happens-Before (1) or Limited Happens-Before (2) (§2.3.2); corresponds to `-V/-H` in §3.1.2.
  - `FILTER_DRS_BY_TID=1` - configures whether to use the `TID` parameter of `data_race` described above.
  - `FILTER_DRS_BY_LAST_CALL=0` - configures whether to use the `LAST_CALL` parameter of `data_race` described above.
  - `ALLOW_LOCK_HANDOFF=0` - configures lockset tracking to permit or disallow a lock taken by one thread to be released by another thread.[2]
  - `ALLOW_REENTRANT_MALLOC_FREE=0` - allow two threads to be in `malloc`, `free`, or so on simultaneously without declaring it a bug.[3]
  - `TESTING_MUTEXES=0` - configure "mutex testing" mode (1), in which the data race analysis will not consider a mutex's implementation to be protected by the mutex itself. In other words, the mutex's internal memory accesses will be flagged as data races, thereby enabling Landslide to verify the mutual exclusion property. Normally (0), Landslide assumes mutual exclusion is provided in

[2] If enabled, accesses performed by the second thread before unlocking will not be considered protected by that lock, as Landslide cannot infer what prior event abstractly represented the lock's ownership changing, leading to spurious data race reports. This could be solved in future work with a new annotation.

[3] Used in Pintos, where those functions lock/unlock the heap mutex themselves rather than relying on a wrapper function to do so before invoking them.

order to efficiently find data races in the rest of the code. Quicksand will automatically set this option for P2s when `-t mutex_test` is specified.

- **Interface options**
  - `TEST_CASE=NAME` - configure the name of the test program to run (mandatory; no default)
  - `VERBOSE=0` - enable more verbose output
  - `BREAK_ON_BUG=0` - configure whether to exit the simulator or drop into a debug prompt when a bug is found. Simics only and not compatible with Quicksand.
  - `DONT_EXPLORE=0` - if enabled, Landslide will not perform stateless model checking but rather will execute the default thread interleaving then exit (useful for manual inspection of preemption points).
  - `PRINT_DATA_RACES=0` - as it says on the tin (for stand-alone use; will message them to Quicksand regardless).
  - `TABULAR_TRACE=1` - configure whether to emit bug reports to the console (0) or to an HTML trace file (1)

3.1.4 Test cases

Landslide depends on human intuition to construct a test case that will produce both meaningful in quality and manageable in quantity thread interleavings.

The user may supply custom test cases for Pebbles (under `pebsim/p2-basecode`) by creating a file in `410user/progs` and adding it to config.mk as usual, or for Pintos (under `pebsim/pintos/src/tests/threads`) by creating a file and adding it to both `tests.c` and `Make.tests`. Tests for the most common interactions during the P2 and Pintos projects are of course already supplied, as described in §5.1.3 and §5.2.3.

Use of `tell_landslide()` annotations is not necessary, although `tell_landslide_preempt()` and `tell_landslide_dump_stack()` may optionally be used at the user's convenience. Additionally, the following "secret" annotations are occasionally used in the pre-supplied test cases to accomplish several mysterious goals described hereupon.

Magic post-test assertions

Test cases may define global variables of the following names to instruct Landslide to assert the following corresponding predicates at the end of each test execution, after all threads exit. Each predicate will be checked iff its first listed variable name is defined; if that variable is defined, all others associated must also be; any combination of the three first-listed variables may be specified at the user's option.

- `magic_global_expected_result == magic_global_value`
- `magic_expected_sum == magic_thread_local_value_parent + magic_thread_local_value_child`

- `magic_expected_sum_minus_result == magic_thread_local_value_parent + magic_thread_local_value_child - magic_global_value`

These could not be implemented as asserts in the test code itself without requiring the student to implement `thr_join()` and `thr_exit()`, avoiding which is important for tests to be student-accessible earlier in the project implementation timeline.

Misbehave

Many of the supplied P2 test cases invoke the `misbehave` system call with a mysterious argument (usually `BGND_BRWN >> FGND_CYAN`) before the creation of any child threads. The use of terminal color code constants is of course a red herring of obfuscation, as the true nature of the Pathos reference kernel's `misbehave` modes is a closely-guarded secret among 15-410 course staff (§2.4.1). The mode in question causes the reference kernel to prioritize scheduling the child thread over the parent whenever `thread_fork` is called, and the target thread over the invoking thread whenever `make_runnable` is called, which are necessary to allow Landslide to recognize a `yield()` preemption point and be able to run the newly-runnable thread as soon as possible.

To illustrate, consider the following program in Figure 3.1, and suppose Landslide is configured to preempt only on mutex API calls (such as in the first step of Iterative Deepening (§4.2)). Because Landslide ignores all kernel-level synchronization short of context switches when testing user-level code, if the kernel created the child thread and returned from `thread_fork` (the system call underlying `thr_create()`) without yielding first, the next preemption point will not occur until `thr_join()` waits for the child to exit. Hence, DPOR will erroneously think everything before that `thr_join()` happens-before (§3.4.2) anything the child does, and will fail to identify the racing accesses on `x`.

```
void child(int *xp) {
    *xp++;
}

void parent() {
    int x = 0;
    int tid = thr_create(child, &x);
    x++;
    thr_join(tid, NULL);
    assert(x == 2);
}
```

Figure 3.1: Example demonstrating the need for `misbehave` to force the kernel to yield during `thread_fork`.

Though Iterative Deepening's soundness (§4.2.1) guarantees all data races will eventually be detected starting from just synchronization preemption points, it assumes threads becoming runnable counts among those. In that sense, `misbehave` serves to restore the

last synchronization preemptions where they belong. If at this point the reader wonders why Landslide doesn't just identify the `thread_create` and `make_runnable` system calls in the arbiter itself (§3.3.5) and skip this mysterious user-visible complexity, they would be right to ask: I have left it this way for no better reason than to maintain consistency with the upcoming chapters' experimental environments, and intend on fixing it in a future update.

Other `misbehave` modes may be used, but are likely to have no effect, since Landslide's thread-scheduling algorithm will override any Pathos-internal scheduling priorities that may arise therefrom. Hypothetically speaking, a reader with access to the top-secret Pathos source code could find further `misbehave` documentation in its `inc/misbehavior.h`.

3.1.5 Bug reports

When Landslide finds a bug, it produces an execution trace of the particular interleaving of threads that led to the bug. This takes the form of a two-dimensional table, with a column for each thread, and each row representing the continuous execution of one thread between two (not necessarily consecutive) preemption points. In each row, the cell in the column corresponding to the executed thread will contain a stack trace, indicating the code location of the preemption point *at the end* of that thread transition (i.e., each stack trace indicates "this thread ran until it reached the indicated line of code"). The bug reports are formatted in html, recommended to be viewed in a web browser. An example is shown in Figure 3.2.

In addition to the preemption trace, the bug report provides some additional helpful information: a stack trace of the current thread at the ultimate point when the bug was executed, a message indicating the nature of the bug encountered, statistics about the size of the state space, and optionally additional information about the bug.[4]

3.2 Kernel annotations

The educational experiments in this thesis focus on projects which students implement on top of provided kernel basecode which Landslide already "understands". Such understanding is conferred via the annotations described in this section. For P2 and Pintos students I supply these annotations behind the scenes, but a CMU 15-410 student who wishes to use Landslide on her kernel project shall need to brave forth hereupon.

3.2.1 config.landslide annotations

The following annotations are specified in `pebsim/config.landslide` akin to the static configuration options described in §3.1.3.. These specify the names of kernel functions,

[4]For certain types of bugs, not pictured here; for example, use-after-frees will report separate stack traces indicating when the corresponding heap block was last allocated and freed. The intrepid source-diver may find all such cases of extra bug details by searching for the macro `FOUND_A_BUG_HTML_INFO` in Landslide's code.

A bug was found!

Current stack (TID 5):
0x01000290 in **panic** (p2-basecode/user/libthread/panic.c:35)
0x01000071 in **critical_section** (p2-basecode/410user/progs/paradise_lost.c:42)
0x01000100 in **consumer** (p2-basecode/410user/progs/paradise_lost.c:54)
0x01000340 in **thread_wrapper** (p2-basecode/user/libthread/thread.c:46)

USERSPACE PANIC: 410user/progs/paradise_lost.c:41: failed assertion `num_in_section == 1 && "long is the way, and hard, that out of hell leads up to light"'

Distinct interleavings tested: 51
Estimated state space size: 150.000000
Estimated state space coverage: 34.000000%

TID 4	TID 5	TID 6
0x01000915 in **mutex_unlock** (p2-basecode/user/libthread/mutex.c:96) 0x01000a19 in **sem_signal** (p2-basecode/user/libthread/sem.c:73) 0x010001e0 in **producer** (p2-basecode/410user/progs/paradise_lost.c:71) 0x01000259 in **main** (p2-basecode/410user/progs/paradise_lost.c:87) 0x01000288 in **_main** (p2-basecode/410user/crt0.c:18) 0xdeadd00d in <unknown in userspace>		
	0x00105c41 in **[context switch]** (kernel__pathos.o:0) 0x010030d7 in **yield** (p2-basecode/410user/progs/paradise_lost.c:87) 0x01000037 in **critical_section** (p2-basecode/410user/progs/paradise_lost.c:39) 0x01000100 in **consumer** (p2-basecode/410user/progs/paradise_lost.c:54) 0x01000340 in **thread_wrapper** (p2-basecode/user/libthread/thread.c:46)	
		0x00105c41 in **[context switch]** (kernel__pathos.o:0) 0x010030e5 in **deschedule** (p2-basecode/410user/progs/paradise_lost.c:87) 0x01000e67 in **cond_wait** (p2-basecode/user/libthread/cond.c:79) 0x010009c1 in **sem_wait** (p2-basecode/user/libthread/sem.c:54) 0x010000fb in **consumer** (p2-basecode/410user/progs/paradise_lost.c:54) 0x01000340 in **thread_wrapper** (p2-basecode/user/libthread/thread.c:46)
0x00105c41 in **[context switch]** (kernel__pathos.o:0) 0x0100314c in **vanish** (p2-basecode/410user/progs/paradise_lost.c:87) 0x01000259 in **main** (p2-basecode/410user/progs/paradise_lost.c:87) 0x01000288 in **_main** (p2-basecode/410user/crt0.c:18) 0xdeadd00d in <unknown in userspace>		
		0x00105c41 in **[context switch]** (kernel__pathos.o:0) 0x010030d7 in **yield** (p2-basecode/410user/progs/paradise_lost.c:87) 0x01000037 in **critical_section** (p2-basecode/410user/progs/paradise_lost.c:39) 0x01000100 in **consumer** (p2-basecode/410user/progs/paradise_lost.c:54) 0x01000340 in **thread_wrapper** (p2-basecode/user/libthread/thread.c:46)
	0x01000290 in **panic** (p2-basecode/user/libthread/panic.c:35) 0x01000071 in **critical_section** (p2-basecode/410user/progs/paradise_lost.c:42) 0x01000100 in **consumer** (p2-basecode/410user/progs/paradise_lost.c:54) 0x01000340 in **thread_wrapper** (p2-basecode/user/libthread/thread.c:46)	

Figure 3.2: Example preemption trace bug report.

global variables, default values, and so on which are required to accurately track the kernel's scheduler state: `CONTEXT_SWITCH`, `EXEC`, `FIRST_TID`, `IDLE_TID`, `INIT_TID`, `MEMSET`, `PAGE_FAULT_WRAPPER`, `READLINE`, `SFREE`, `SHELL_TID`, `SPURIOUS_INTERRUPT_WRAPPER`, `THREAD_KILLED_ARG_VAL`, `THREAD_KILLED_FUNC`, `TIMER_WRAPPER`, `VM_USER_COPY`, `VM_USER_COPY_TAIL`, `YIELD`.

Following are the less self-explanatory options.

- `PINTOS_KERNEL=0` - configure Landslide for Pebbles (0) or Pintos (1) kernel architecture. Normally set automatically by the setup scripts.
- `TESTING_USERSPACE=1` - configure Landslide whether to test (i.e., focus preemption points, memory analysis, etc. on) the userspace or kernelspace code.
- `CURRENT_THREAD_LIVES_ON_RQ=0` - Landslide infers the list of runnable threads from the `tell_landslide_on_rq()` and `off_rq()` annotations (described below). Some kernels[5] remove the current thread from their runqueue, such that the abstract set of all runnable threads is actually the runqueue plus the current thread rather than just the runqueue. Other kernels[6] leave the current thread on the runqueue, removing it only when it's descheduling and should actually be considered blocked. Set this option to 0 to support the former kernel type or 1 to support the latter.[7]
- `PREEMPT_ENABLE_FLAG=NAME` - name of a global variable which the kernel uses to toggle scheduler preemptability, for kernels which may disable preemption without disabling interrupts. For kernels wherein preemptability is corresponds directly by interrupts, leave this option unspecified.
- `PREEMPT_ENABLE_VALUE=VAL` - value of the above variable when preemption is enabled (usually 0; note that many kernels use a nesting depth counter where any positive value corresponds to disabled).[8]
- `PATHOS_SYSCALL_IRET_DISTANCE=VALUE` - indicate how much stack space is used by the reference kernel's system call wrappers. Used for cross-kernel-to-userspace stack traces; if unset, stack traces from kernel space will end at the system call boundary.
- `PDE_PTE_POISON=VALUE` - indicate a poison value used in the page tables to indicate absent VM mappings to check for as well as checking the present bit (if unspecified, will check present bit only)
- `BUG_ON_THREADS_WEDGED=1` - set to 0 to disable deadlock detection but instead let the kernel keep receiving system interrupts when all threads appear blocked.[9]
- `TIMER_WRAPPER_DISPATCH=NAME` - used to manually indicate a label before the end of the timer interrupt assembly wrapper, in case the `iret` instruction couldn't be found automatically (see `pebsim/definegen.sh`).

[5] most, actually

[6] the author's own student kernel from long ago

[7] This option replaces the deprecated `kern_current_extra_runnable()` annotation from `student.c` described in [12] §6.2.3.

[8] These two options replace the deprecated `kern_ready_for_timer_interrupt()` annotation from `student.c` described in [12] $6.2.3.

[9] once used in the bad old days; now recommended for debugging use only

- `starting_threads TID STARTS_ON_RQ` - specifies a system thread which already exists at the time `tell_landslide_sched_init_done()` (see below) is called; `TID` is the thread's ID and `STARTS_ON_RQ` is 0 or 1 to indicate whether or not it starts on the system runqueue. Typical threads to use this for are init and idle.
- `ignore_sym NAME SIZE` - specifies a global variable `NAME` of a given `SIZE` in bytes whose memory accesses should be ignored for the purposes of DPOR and data race analysis. Typical symbols to use this for are the console or heap mutex.
- `sched_func NAME` - specifies a function whose memory accesses should all be ignored for the purposes of DPOR and data race analysis. Typical functions to use this for are the timer handler and context switcher.
- `disk_io_func NAME` - specifies a function which may block a thread waiting for disk I/O (or other external interrupt) rather than blocking on another thread. If any threads are blocked in a disk I/O function during an apparent deadlock, Landslide will allow the kernel to idle until the simulator delivers the appropriate interrupt, rather than declaring a bug.
- `ignore_dr_function NAME USERSPACE` - specifies a function whose memory access should not be counted as data races (but still be considered memory conflicts for DPOR). `USERSPACE` should be 0 or 1 to denote a kernel-space or user-space function respectively.
- `thrlib_function NAME` - specifies a function whose memory accesses should be ignored both by the data race analysis and by DPOR. This is recommended for marking trusted-correct thread library code when testing multithreaded client code thereof, in order to avoid unnecessarily checking, for example, all the different ways `thr_exit()` and `thr_join()` could interleave. The user should be careful with this option to also enable the proper `thr_create()` misbehave mode in her test case (§3.1.4).
- `TRUSTED_THR_JOIN=0` - if set to 1, forces Landslide to add a happens-before edge (§3.4.2) between the exiting of some thread N and the end of any subsequent `thr_join`(N) call, even if that `join` would not ordinarily block. This is useful for state space reduction when testing threaded client code; for example, in the interleaving TID1: `x++; thr_exit();`, TID2: `thr_join(1); print(x);`, DPOR, not automatically trusting `join`'s behaviour, will attempt to test the TID2, TID1 interleaving to reorder the accesses on `x`, whereupon `join` will block, forcing these interleavings to be equivalent. This option allows DPOR to skip checking that `join` behaves properly and to prune the second interleaving by teaching it the expected blocking semantics. Obviously, not for use when actually testing `thr_join` itself!

3.2.2 In-kernel code annotations

The following annotations are provided as C functions which a kernel author shall include in her source code and call at appropriate times. The functions' actual implementations are empty; rather they serve as labels whose positions the annotation scripts extract

along with the other various annotations from the previous section. Some of these are mandatory for Landslide to function properly, while others serve to improve or otherwise manipulate the state space.

Mandatory annotations

- `tell_landslide_thread_switch(int new_tid)` - to be called during context switch, indicating the newly-running thread (must be called with interrupts and/or scheduler preemption disabled)
- `tell_landslide_sched_init_done()` - to be called after scheduler initialization, indicating the point after which Landslide should begin analysis. Any threads already initialized before this point (init, idle, etc) should be specified with `starting_threads` (previous subsection).
- `tell_landslide_forking()` - to be called whenever a new thread is created, "immediately" before the next `thread_switch()` or `on_rq()` call for that new thread (i.e., this call sets a flag which the next instance of either of the latter will check to see if the indicated thread is new). Most Pebbles kernels will call this twice; once in `fork` and once in `thread_fork`.
- `tell_landslide_vanishing()` - to be called whenever a thread ceases to exist, "immediately" before the next `thread_switch()` or `off_rq()` call for the exiting thread (works similarly to above).
- `tell_landslide_sleeping()` - to be called whenever a thread is about to `sleep()` waiting for timer interrupts, "immediately" before the next `thread_switch()` or `off_rq()` call for the sleeping thread (similar to the above). Landslide considers sleeping threads to be runnable as normal (they will just take more timer interrupts to arrive at), so this call is necessary to distinguish from the case when a thread is descheduled on a non-timer event.
- `tell_landslide_thread_on_rq(int tid)` - to be called when a thread is added to the runqueue (must be called with interrupts and/or scheduler preemption disabled).
- `tell_landslide_thread_off_rq(int tid)` - dual of the above. If `CURRENT_THREAD_LIVES_ON_RQ=0` (described above), this should be invoked (among other times) during context switch with the TID of the thread about to start running. Alternatively (thanks sully), even for a kernel which takes the current thread off its literal runqueue, the annotator may use these two calls to indicate the "abstract runqueue" which includes the current thread as well, and set `CURRENT_THREAD_LIVES_ON_RQ=1`.

Optional annotations

- `tell_landslide_preempt()` - specifies a preemption point. Subject to the constraints of `within_function/without_function`; hence may be ignored if used with Quicksand.

- `tell_landslide_dump_stack()` - instructs Landslide to print a stack trace whenever this point is reached (for debugging purposes).

Optional but strongly recommended annotations

The following annotations enable Landslide to track locksets for data race analysis. If not provided, it will be as if Landslide assumes no guarantees about mutual exclusion or happens-before, and hence will identify all memory conflicts as data races. (Note that the corresponding instrumentation for P2s is achieved automatically, as the names of the mutex interface are mandated by the project specification.)

- `tell_landslide_mutex_locking(void *mutex_addr)` - indicates the beginning of the lock routine for whatever synchronization API Landslide should treat as the primitive for data race detection. In Pintos this is the `sema_*()` function family; in Pebbles they may be called anything.
- `tell_landslide_mutex_blocking(int owner_tid)` - called "immediately" before a thread becomes blocked on the mutex. Definition of "immediately" similar to the `forking()` and friends annotations above. `owner_tid` allows Landslide to efficiently unblock/re-block threads when the mutex holder changes (rather than relying on heuristic yield-loop detection); see `kern_mutex_block_others()` and `deadlocked()` in `schedule.c` for implementation details.
- `tell_landslide_mutex_locking_done(void *mutex_addr)` - indicates the end of the lock routine.
- `tell_landslide_mutex_trylocking(void *mutex_addr)` - indicates the beginning of the trylock routine (if present).
- `tell_landslide_mutex_trylocking_done(void *mutex_addr, int succeeded)` - indicates when a thread is finished trylocking, even if it failed to get the lock (indicated by `succeeded`).
- `tell_landslide_mutex_unlocking(void *mutex_addr)` - indicates the beginning of the unlock routine.
- `tell_landslide_mutex_unlocking_done()` - indicates the end of the unlock routine.

3.3 Architecture

This section documents the organization of code within Landslide. Unless otherwise specified, Landslide's code lives in `work/modules/landslide/` (Simics implemenation) or `src/bochs-2.6.8/instrument/landslide/` (Bochs implementation) relative to the repository root.

Both simulators invoke Landslide once per instruction and once per memory read or write. The entry point is the aptly-named `landslide_entrypoint()` in `landslide.c`, which then dispatches to various other modules' respective analyses, described as follows.

3.3.1 Execution tree

The execution tree is stored as a chain of preemption point nodes named `struct hax` defined in `tree.h`. Although the state space of possible interleavings is exponentially-sized, Landslide does not actually need to store any nodes for execution sequences outside the current variation (see §3.4.3 and §3.4.2 for why), so the total memory consumption is only $O(n)$ in the number of preemption points in a single program run (for the test cases used in this thesis, typically 20-1000). Each `hax` stores the following information:

- Basic statistics such as the current instruction pointer, thread ID, stack trace of current thread at the moment of preemption, depth in the tree, parent node pointer, etc.;
- Snapshots of the current state of the scheduler (§3.3.2) and memory accesses and heaps (§3.3.3);
- Simulator-dependent data needed to time travel and resume execution from this checkpoint (§3.3.5);
- List of parent/ancestor nodes with memory conflicts and/or happens-before edges to this one for DPOR (§3.4.2);
- Current estimated state space proportion and execution time for the subtree rooted at this node (not necessarily fully explored yet) for estimation §3.4.3;
- Whether this point is an `xbegin` invocation and if so what `xabort` codes are possible and/or already explored for this transaction (§6).

3.3.2 Scheduler

The Landslide scheduler, which lives in `schedule.c`, has two main duties: to maintain an accurate representation of all the existing threads on the simulated system and track what concurrency-relevant actions each is performing at any given time, and to orchestrate the sequence of timer interrupts necessary to cause the simulated system to context switch to any given thread at any given time. System-wide state is stored in a single `struct sched_state`, including the thread queues (runqueue, deschedule queue, and sleep queue), while per-thread state is stored in `struct agents` (named after the terminology of [127]) which live on said queues.

It has one main entrypoint, `sched_update()`, in which both the state machine is updated and scheduling decisions are made. The interface also offers helper functions for finding and manipulating `agents`, and `sched_recover()`, which prepares the scheduler to force a new thread to run after a time travel (§3.3.5).

State machine

The first part of `sched_update()` is to update the state machine of thread actions and runnability. Much of this functionality is found in `sched_update_kern_state_machine()` and `sched_update_user_state_machine()`. The current intruction pointer is compared

against the known locations of the mutex API, system calls, runnable/descheduling `tell_landslide` annotations, and so on, and locksets, action flags, and runqueue membership are updated accordingly.

Interrupt injection

The second part of `sched_update()`, conditional on the arbiter identifying preemption points (§3.3.5), manages timer interrupts to switch to a desired thread. Whenever a preemption point is reached, the scheduler first creates a checkpoint in the execution tree (§3.3.1), asks the arbiter which thread to run next, and if that thread is different from the current one, forces the kernel into its timer interrupt handler (§3.3.4).

Because the kernel is part of the system being tested, Landslide can't necessarily always switch directly to a specific thread, but rather must keep triggering context switches until the desired thread is reached; any mechanism to tell the kernel which thread it wants would necessarily involve modifying the code being testsed and hence possibly obscuring bugs or introducing new ones.[10]

The scheduler marks up to one thread as the "schedule target", which when set makes Landslide wait until that thread is reached before looking for more preemption points, so the kernel may finish its context switches undisturbed. Whenever the schedule target is set and the end of the context switcher is reached, if the schedule target is not the current thread, the scheduler repeats this process until it is.[11]

3.3.3 Memory analysis

`memory.c` is responsible for all manner of memory access analysis. It tracks heap allocations, checks reads and writes in the heap region against same; tracks reads and writes (in any region) from each thread and checks them against each other for DPOR (§3.4.2) and data race analysis (§3.4.4). For userspace tests, it also tracks which virtual address space (`cr3`) belongs to the test program via a state machine of the test lifecycle, which lets it avoid false positive heap errors from other programs which have differently-addressed heaps (`check_user_address_space()` and `ignore_user_access()`.

Heap checking

`mem_update()` serves as the main entrypoint for tracking heap allocations. It's called every instruction to check for the boundaries of the `malloc` library, and behaves in a similar way to the scheduler state machine described above. Then, `mem_check_shared_access()` checks (after some elaborate manoeuvres to figure out whether to use the kernel- or

[10]For userspace testing, where I supply a pre-annotated reference kernel, such an approach would be more straightforward, but the kernel-testing repeated-context-switch approach infrastructure was already in place and it was easier to reuse that than to add more code.

[11]Note that this "loop" is not structured as an explicit loop in Landslide's code, but rather as part of the state machine which updates each time a new instruction is traced.

userspace heap) whether, if in the heap region, the memory is contained in a currently-allocated heap block, reporting a bug if not.

Memory conflicts

`mem_check_shared_access()` also records each such access in a per-thread-transition rb-tree, which is saved and then cleared at each preemption point. This allows `mem_shm_intersect()`, called at each preemption point once for each of its ancestors (n^2 total calls per interleaving), to perform a set intersection to find any memory conflicts. Any such conflicts which also fail a lockset and/or happens-before check (§3.4.4) are then reported as data races. Regardless, all such conflicts are later used by DPOR (§3.4.2) to find dependent transition pairs.

3.3.4 Machine state manipulation

The interface to inspect and manipulate the simulated machine state lives in `x86.c`.

Memory

`read_memory()` and `write_memory()` are both provided (with various wrapper macros in `x86.h`). The former is used basically everywhere throughout Landslide to query the machine state; the latter is used only by the interrupt manipulation below and by the scheduler to force Pintos to skip certain parts of its init sequence (§3.3.8). Both rely on the helper function `mem_translate()` for virtual address resolution, which at present supports only the normal x86 32-bit addressing mode (no PAE, long mode, etc.).

Interrupts

Several routines are provided for manipulating system interrupts. Note that the Landslide is called once per fetch-decode-execute loop of the CPU, after the CPU processes all already-pending interrupts and decides which instruction to execute, but before actually executing the instruction (true of both Bochs and Simics). I refer to this as the *upcoming instruction*. Whether or not Landslide wants that instruction to execute before triggering a thread switch is a matter of some concern in the following API.

- `cause_timer_interrupt()` triggers a pending timer interrupt, whose handler will be entered as soon as the upcoming instruction is executed.
- `cause_timer_interrupt_immediately()` does as above, but forces the system to enter the interrupt handler before the upcoming instruction is executed. That instruction will be executed upon return from the interrupt.
- `avoid_timer_interrupt_immediately()` suppresses a timer interrupt triggered by the simulator from outside of Landslide's control. It acknowledges the APIC and forces the system to jump to the end of the interrupt handler.

- `delay_instruction()` forces the system to execute a no-op before the upcoming instruction, effectively converting an invocation of `cause_timer_interrupt()` to `cause_timer_interrupt_immediately()`.
- `cause_keypress()` triggers a keyboard event corresponding to the specified character. The interrupt will be taken after the upcoming instruction is executed (provided no timer interrupt is simultaneously pending). Only `a-z`, `0-9`, `_`, space, and newline are supported (enough to name any Pebbles test case).
- `interrupts_enabled()` queries the CPU's interrupt flag (`eflags:IF`).
- `cause_transaction_failure()` forces `_xbegin()` to return a specified abort code.

Note that `kern_ready_for_timer_interrupt()` should generally be invoked separately from `interrupts_enabled()` if needed; while `interrupts_enabled()` must be true before invoking `cause_timer_interrupt()`, if the kernel is not ready the interrupt may not be received for a long time. Also, `cause_timer_interrupt_immediately()` must not be used while the kernel is not ready.

3.3.5 State space traversal

Traversal of the state space is implemented in three parts: first, identifying preemption points when first encountered and selecting which thread to run for its first execution, in `arbiter.c`, second, selecting which preemption point to backtrack to after completing an execution and which thread to "have switched to" instead, in `explore.c`, and third, rewinding the machine state to implement said backtracking, in `timetravel.c` (Bochs version) and `timetravel-simics.c` (Simics version).

Arbiter

The arbiter (named after the corresponding component of dBug [127]) is responsible for checking which code locations during execution should be identified as preemption points (`arbiter_interested()`), and thereupon for choosing whether to keep running the current thread or to preempt and switch to a new one (`arbiter_choose()`). Its behaviour in the former case is configured by the options listed in §3.1.3, and in the latter case by the options listed in §3.1.3. For example, `EXPLORE_BACKWARDS` is interpreted here; if set, it will cause Landslide to always preempt and switch threads the first time it encounters each new preemption point.[12]

Explorer

Landslide invokes the explorer at the end of each execution of the test case, which analyzes the current branch of the interleaving state space tree to figure out which alternate branch

[12]Another secret option, `CHOOSE_RANDOMLY`, also exists here to randomize whether to "explore backwards" (choosing independently at each preemption point, resulting in an overall unpredictable exploration order). It's not exposed to `config.landslide` but rather the user must edit it in `arbiter.c` directly, whereupon the probability may also be adjusted via `numerator` and `denominator`.

to try executing next. Its contents are largely algorithmic rather than architectural and hence further described in §3.4.2 and §3.4.5.

Time travel

After the explorer picks a past point of the program to preempt, Landslide collaborates with the simulator to revert the machine state to that point before switching to the desired thread. The Simics version is merely a bunch of wrapper glue code around the `set-bookmark` and `skip-to` backtracking commands. Bochs however does not support backtracking, so I instead use `fork()` to get Linux to copy the machine state for me.

The big issue to note here is that, while the simulation state should be completely reverted, parts of Landslide's state (e.g., scheduler runqueues, thread action flags) should likewise be reverted to mirror the change in program state, while others (tagged ancestor branches from DPOR, state space estimates) should be preserved from branch to branch. In Simics, I simply copy every data structure of the former case (`copy_sched()` and friends in `save.c`), leaving those of the latter undisturbed across backtracks.[13]

In Bochs, `fork()` automatically copies everything, so the reverse holds: all data of the latter case must whenever updated be propagaged to all `fork()`ed children processes explicitly. I worried while implementing this that I might miss a case, or that future updates to the code could easily forget this step, resulting undoubtedly in state corruption bugs which to diagnose would be a thesis in their own right, so I enlisted help from my compiler via the oft-ridiculed `const`. Every preemption point node in the execution tree (`tree.h`), each of whose state is kept generally read-only, and all modifications must go through `modify_hax()` (`timetravel.h`) using a modification callback, which internally casts away the `const`, performs the requested modification, and also messages all relevant child processes to perform the same (`timetravel_set()` in `timetravel.c`). The `const` is absolutely, inviolably, not to be casted away, at the sacred cost of what little type safety C offers.[14] Thence the typechecker enforces that all exploration-related state is properly propagated while scheduler state is automatically reverted.

3.3.6 Bug-finding output

The infrastructure for producing the diagnostic output to help users understand their bugs can be classified in three parts: the symbol table glue, the excessively clever stack tracer, and the preemption table generator.

[13] Simics actually wants to save/restore all its modules' internal state on its own, offering an attribute set/get API for modules to expose such state (used for other purposes in `simics_glue.c`), but doing deep copies of data structures this way would be more trouble than it's worth.

[14] Of course this would be followed by a footnote describing the one place where I cast it away anyway, `mem_check_shared_access()` in `memory.c`; why it's ok is documented in an `XXX` comment in the code.

Symbol table

The symbol table logic lives in `symtable.c` and is pretty much a lot of glue code. In the Simics version, Landslide relies on the `deflsym` Simics object created by the 15-410 python scripts, and queries its attributes using Simics API calls. In the Bochs version, function names and line numbers are handled separately: Bochs is patched with a new API function named `bx_dbg_symbolic_address_landslide`[15] which provides function names and hexadecimal offsets; while for line numbers, `pebsim/pintos/build.sh`[16] generates a header file `line_numbers.h` using `objdump` and `addr2line`, which the aforementioned hex offset then serves as an index into.

Stack traces

The stack tracer is implemented in `stack.c`. It does the standard approach of following the base pointer chain (not supporting code compiled with `-fomit-frame-pointer` by doing anything clever like understanding how much stack frame is allocated for each function), and printing symbol table information for the pushed return addresses at the top of each frame.

However, it also offers several special-case features which even some students have sometimes noticed as being more clever than Simics's stack tracer. I document those features here. As a common point of implementation among them, Landslide traces the stack pointer `esp` in addition to the base pointer `ebp`; not only updating it whenever dereferencing the base pointer, but also when decoding simple assembly routines, finding "hidden" stack frames without base pointers, identifying system call wrappers, and so on. The corresponding code lives in `stack_trace()` in `stack.c`.

- If a function is preempted at its beginning or end such that its corresponding base pointer is missing from the base pointer chain, Landslide will find its "hidden" frame and include it in the stack trace in the following cases.[17]
  - If the last pushed return address is at offset 0 into the body of its containing function, Landslide will find the next pushed return address at `esp+0`.
  - If as above but the function is the page fault handler, at `esp+4`.
  - If the return address is at offset 1 and the previous instruction was `push ebp`, Landslide will find the next pushed return address at `esp+4`.
  - If the return address is a multiple of 4 offset and all previous instructions are of the form `mov m32,r32`, Landslide will find the next pushed return address at `esp+0`. (This is common in student hand-written assembly functions.)
- If the instruction at a pushed return address is a `pop` or `popa`, Landslide will search for the next non-`pop(a)` instruction, and if it's `ret` or `iret`, treat the function as a

[15] does the same thing as the existing `bx_dbg_symbolic_address`, but with a better type signature

[16] Line numbers in Bochs for Pebbles/P2s are not supported yet; see `p2-setup.sh` for the work-in-progress.

[17] Note that in such cases, most other debuggers' stack tracers will be missing not the name of the interrupted function, but the name of the function which called that function, because it's the former's stack frame which should enable the debugger to find the pushed return address for the latter.

system call wrapper (which tend not to preserve the base pointer chain) and find the next return address above where all those registers were pushed.

- If a return address was pushed during a fault or interrupt (determined by checking for the `iret` opcode or the page fault wrapper special case mentioned above), Landslide will read the iret block to determine whether a stack switched happened and if so what `esp` used to be.
- If a return address's offset into its containing function is 0, and the last instruction in the preceding function (binary-wise) is a `call`, Landslide will recognize it as a `noreturn` tail-call, and print the correct function name.[18]
- Landslide runs the tortoise/rabbit algorithm to detect cyclic `ebp` chains and terminate after the first time around.
- Two other Pebbles-specifc special cases described in §3.3.7.

Also implemented in `stack.c` is the backend of the `within_function/without_function` configuration command, which searches a given stack trace for the presence of a function return address within a specified range.

Preemption traces

The preemption traces, described and exemplified in §3.1.5, are generated by `found_a_bug.c`, in cooperation with `save.c`. Whenever `save.c` creates a preemption point, it captures a stack trace of the current thread at the point it was interrupted, and saves it in the preemption point tree. `found_a_bug.c` then traverses the current branch of the tree, potentially producing both console output and HTML output (controlled by the `HTML_PRINTF` macro family). It should be invoked by the `FOUND_A_BUG` macro defined in `found_a_bug.h`, or by `FOUND_A_BUG_HTML_INFO`, which also allows the caller to specify a callback to print extra details (such as use-after-free stack traces) in the bug report.

3.3.7 Pebbles-specific features

This section lists special cases of instrumentation specific to the Pebbles kernel.

- `mem_check_shared_access()` (`memory.c`) will assert that kernel memory is direct-mapped.
- `use_after_free()` (`memory.c`) will ignore use-after-free reads originating from kernel code during the `swexn` system call (an extremely common and neither harmful nor technically interesting bug among student implementations).
- `cause_test()` (`test.c`) will issue keyboard input to type the test case name and press enter when the initialization sequence completes and the shell is blocked on `readline`.

[18]Normally Landslide reports function/line number information for the return address as-is, which indicates the next line of code after the relevant call rather than the call itself.

- `kern_mutex_block_others()` (`schedule.c`) will handle the special "blocked on via mutex" state changes whenever a mutex is acquired or released, for kernels which use the `tell_landslide_kern_mutex_blocking()` annotation.
- `sched_update_kern_state_machine()` (`schedule.c`) will handle the reference kernel's invocation of `sched_unblock()` within `cond_signal()` as a signal event for happens-before analysis.
- `cause_timer_interrupt_immediately()` (`x86.c`) will read the `esp0` value out of the TSS to support user-to-kernel mode switches in timer interrupts injected during userspace execution.
- `splice_pre_vanish_trace` (`stack.c`) will, when a `vanish`ing thread has already deleted its Simics symbol table object, splice in a saved "pre-vanish" stack trace (saved previously in `sched_update_kern_state_machine()`) so the user can see the userspace execution sequence preceding the `vanish` invocation.
- `stack_trace` (`stack.c`) will, when `ebp` crosses from kernel- to userspace across a system call boundary (a reference kernel feature to allow Simics stack traces to cross same), use `PATHOS_SYSCALL_IRET_DISTANCE` (§3.2.1) to track `esp`'s value across the stack switch.

3.3.8 Pintos-specific features

This section lists special cases of instrumentation specific to the Pintos kernel.

- `arbiter_interested()` (`arbiter.c`) will automatically insert preemption points on `intr_disable()` and `intr_enable()` calls (immediately before and after the interrupt state is changed, respectively) (as long as they aren't part of the mutex implementation, which has preemption points of its own).
- `lockset_remove()` (`lockset.c`) will warn instead of panic if a lock is unlocked twice, to allow for double `sema_up()` in cases where the lock is actually a multi-use semaphore rather than a mutex.
- `build.sh` will edit the `bootfd.img` binary to implant the name of the test case to be run in the kernel's boot command.
- `sched_check_pintos_init_sequence()` (`schedule.c`) will force the kernel to skip the `timer_calibrate()` and `timer_msleep()` routines used in I/O initialization.
- `keep_schedule_inflight()` (`schedule.c`) will detect when an attempted thread switch is impossible because the timer handler's try-lock will fail, and will abort the interleaving early as if it never existed (which it shouldn't).
- `sched_update_kern_state_machine()` (`schedule.c`) will:
  - track invocations of `timer_sleep()` and `list_insert_ordered()` to infer when a thread is sleeping rather than blocked automatically, rather than relying on the `tell_landslide_sleeping()` annotation.
  - allow `sema_up()` to reenter itself, which may happen when an IDE interrupt is

taken when interrupts are re-enabled at the end of said function.
    - handle interrupt disabling/enabling as an abstract global lock for the purposes of happens-before analysis.
- `sched_update()` (`schedule.c`) will handle "lock handoff" of the abstract disable-interrupts lock during a context switch for happens-before analysis.
- `memory.c` (various functions) will handle page allocations from the `palloc` family of functions in a separate memory heap, allowing the usual allocator `malloc` to allocate and free from `palloc`'ed memory, and checking both allocation heaps when checking for use-after-frees.
- `kern_address_in_heap()` (`kernel_specifics.c`) will ignore DMA accesses to the VGA console, which appear to be in Pintos's heap region.
- `test_update_state()` (`test.c`) will use the boundaries of `run_test()` to denote the test lifecycle.

3.3.9 Handy scripts

The options specified in §3.1.3 are handled by a family of gross shell scripts that live in `pebsim/`.

- `landslide` is the outermost script invoked by Quicksand (or by a §3.1.3 aficionado). It exports several key environment variables used by the other scripts, ensures the instrumentation is up-to-date, and launches the simulator.
- `getfunc.sh` defines several functions commonly used by `build.sh` and `definegen.sh` to extract function or global variable addresses from the program binary.
- `symbols.sh` defines the names of kernel functions that can be instrumented automatically without a corresponding manual annotation (e.g., `malloc` and friends, the names of the `tell_landslide` family, various library helpers such as `panic`).
- `build.sh` ensures the build of Landslide is up-to-date, and processes any dynamic configuration options which don't require updating the build (§3.1.3) It verifies all required `tell_landslide` annotations are present, verifies all required `config.landslide` options, processes the dynamic config options, checks whether or not `definegen.sh` needs to be run again (via hashes stored in `student_specifics.h` of the program binary and static config options), and does so if necessary.
- `definegen.sh` produces the content of `student_specifics.h`. It repeatedly invokes the helpers defined in `getfunc.sh` to find the addresses of both functions specified in the config options and functions whose names are known in advance.[19]
- `p2-basecode/import-p2.sh` and `pintos/import-pintos.sh` are invoked by their respective setup scripts to copy the student implementation into their respective directories. §5.1.2 and §5.2.2 describe their office in more detail.

[19]You might think it should invoke objdump but once and keep the output in a shell variable, but I tried that and it was mysteriously slower, so I gave up without ever figuring out why.

The final output of these scripts is an auto-generated header, `student_specifics.h`, containing a bunch of `#defines` of the addresses of important functions in the compiled binary, specific features enabled or disabled by the static config options (§3.1.3), and so on. The files `kernel_specifics.c`, `user_specifics.c`, and `student.c` provide several interface functions for interpreting the current program state with respect to these values.

3.4 Algorithms

This section describes Landslide's model-checking algorithms from a theoretical perspective. The more complex ones are accompanied by concrete examples to hopefully help the reader build a solid intuition, which upcoming chapters will require in their soundness proofs.

3.4.1 Preemption point identification

When should Landslide sunder the universe into alternate realities, in which each a different thread executes immediately following the current instruction? This singular question determines to what extent the state space of possible interleavings explodes exponentially. While other parts of the great work decide which lock API calls to consider, or which memory accesses constitute a data race, interpreting those combinations of preemption point *predicates* to decide if the current program state constitutes a single preemption *point* warrants discussion.

Preemption point identification is implemented largely in `pp.c`. At startup, `pps_init()` and `load_dynamic_pps()` process the statically-configured preemption points (§3.1.3) and dynamically-configured preemption points (§3.1.3), respectively. Each of these configurations may contain any number of `within_function`, `without_function`, and `data_race` commands.

Stack trace inclusion/exclusion

`check_withins()` implements the whitelist/blacklist behaviour for the former two of those commands (in a similar manner to the Preemption Sealing technique described in prior work [9]). It invokes the stack tracer (§3.3.6) for a list of which functions are on the call stack (hence the importance of the stack tracer's complex logic to not miss any frames even when interrupts or system calls are involved). Then, to determine if the current program state should be considered a valid preemption point, or whether it should be rejected, it compares each `within` or `without_function` directive in the following sequence-dependent manner:

- If no `within_function` commands are given, operate in "blacklist mode": the preemption point is by default valid as long as no `without_function` calls reject it. Otherwise, operate in "whitelist mode": the preemption point is by default rejected unless at least one `within_function` directive matches.

- Subject to the above, find the sequentially-last `*_function` directive (static preemption point commands considered before dynamic ones) which matches any function in the stack trace. If `within`, accept the preemption point; if `without`, reject it.

The same comparison is done for `within_user_function` and `without_user_function`.

Data race predicates

The `data_race` command specifies an instruction pointer value to identify as a data race preemption point. The point can optionally be qualified by a thread ID, most recent system call number, etc., as described in §3.1.3, and is queried through `suspected_data_race()`.

In Preempt-Everywhere mode, instead Landslide marks all shared memory accesses as long as they are not either part of the mutex implementation or part of the running thread's stack frame. The `data_race` command is ignored and `suspected_data_race()` instead checks whether the instruction pointer is associated with any such shared memory access.

Preemption point predicates

`arbiter_interested()` in `arbiter.c` then checks various annotations and hard-coded preemption point predicates to decide whether the current program state constitutes a preemption point. The following predicates are constrained by `check_withins()`:

- `suspected_data_race()`
- User or kernel `mutex_lock()` or `mutex_unlock()` call
- Custom kernel preemption point using `tell_landslide_preempt()` (relic of [12], largely obsoleted by data-race preemption pooints)

The following predicates ignore any `within_function` settings (mandatory preemption points needed, for example, to maintain the one-thread-per-transition invariant):

- Voluntary reschedule, e.g. explicit `yield()`
- `hlt` instruction (kernel waiting for interrupt)
- User thread becomes yield- or xchg-blocked (§3.4.6)
- `_xbegin()` or `_xend()`, if testing transactional memory (Chapter 6)

Whenever `arbiter_interested()` returns true, Landslide creates a new `struct hax` in the execution tree (§3.3.1), creates a checkpoint (§3.3.5), and queries `arbiter_choose()` to decide which thread to run next (§3.3.5).

Example

Consider the following examples to illustrate the behaviour of the stack trace directives.

1. `mutex_lock()`, in `malloc()`, in `thr_create()`, in `main()`
2. `mutex_lock()`, in `cond_wait()`, in `thr_join()`, in `main()`

and the following `within/without_user_function` combinations:

- `within_user_function mutex_lock`, `without_user_function malloc`

  Rejects stack trace 1 (last matching directive is to exclude `malloc`), accepts stack trace 2 (last matching directive is to include `mutex_lock`)
- `within_user_function thr_join`

  No `without`s present, so behaves as a whitelist, rejecting stack trace 1 (not in `thr_join()`), accepting stack trace 2 (in `thr_join()`).
- `without_user_function cond_wait`

  No `within`s present, so behaves as a blacklist, accepting stack trace 1 (not in `cond_wait()`), rejecting stack trace 2 (in `cond_wait()`).
- `without_user_function main`, `within_user_function mutex_lock`

  Accepts both (last matching directive is to accept `mutex_lock()`, regardless of `main()`)

3.4.2 Dynamic Partial Order Reduction

Landslide implements Dynamic Partial Order Reduction (DPOR) [46] to identify concurrent yet independent thread transitions whose permutations can safely be pruned from the state space while still testing all possible program behaviours.

The DPOR implementation consists of 3 parts: computing happens-before, computing memory conflicts, and tagging alternate branches to explore to drive the state space exploration. The former two are computed as each preemption point is reached, for the associated thread transition pairwise with all other preceding thread transitions. The latter is computed at the end of each full interleaving executed, using the results of the two former, and constitutes the bulk of the algorithm.

In this section t_i will denote a transition between two program states during execution, with each state being a preemption point as identified in §3.4.1, and $T(t_i)$ will denote the thread which was scheduled to produce that transition. A visual example will be given at the end to help reinforce the intuition behind the formalism.

Happens-before

The happens-before relation expresses when two thread transitions can potentially be reordered, or in other words, are concurrent (despite the serialized nature of the simulated execution). This relation is expressed in the following definitions paraphrased from [46].

- **Enabled**: A transition t_i is enabled in a state s when a state s' exists such that $s \xrightarrow{t_i} s'$ exists. In systems terms, the scheduler at state s considers $T(t_i)$ to be runnable.
- **Dependent**: Two transitions t_i and t_j are dependent if
  1. t_1 is enabled in s and $s \xrightarrow{t_1} s'$, and
  2. t_2 is enabled in s but not enabled in s', or vice versa.

  In systems terms, either $T(t_1) = T(t_2)$, or the execution of t_1 at s causes $T(t_2)$ to

change state from blocked to runnable or vice versa. [20] Landslide computes this relation in `enabled_by()` in `save.c`.

- **Happens-Before**: The happens-before relation for a transition sequence $S = t_1 \dots t_n$ is the smallest relation $\rightarrow_S$ on $1 \dots n$ such that
  1. if $i < j$ and S_i and S_j are dependent then $i \rightarrow_S j$, and
  2. $\rightarrow_S$ is transitively closed.

  Landslide computes this relation in `compute_happens_before()` in `save.c`.

The happens-before relation is a partial order expressing the scheduling constraints of a given interleaving. All pairs of interleavings not included are subject to reordering, and hence candidates for new interleavings to test.

Note that DPOR's notion of happens-before differs from the traditional distributed systems definition [81] as used in Pure Happens-Before §2.3.2; rather, it coincides with condition 3 of Limited Happens-Before (in fact, Landslide's Limited HB implementation simply reuses the same result computed for DPOR's purpose).

Memory conflicts

The memory conflict relation expresses when two transitions are dependent, or in other words, when their behaviour could potentially vary depending which executes first.

Upon execution of each $t_j \in S$, Landslide saves the current set of all memory accesses since the previous preemption point (call this $M(t_j)$), compares it to all $M(t_i)$ with $i < j$ and $i \not\rightarrow_S j$, and then begins recording subsequent memory accesses in a new empty set for t_{j+1}. (`shimsham_shm()` in `save.c`). These $M(t)$ sets are mappings from memory addresses to instruction pointer value, read-or-write boolean, lockset or vector clock, and various other metadata (`struct mem_lockset` in `memory.h`).

The set intersection is implemented in `mem_shm_intersect()` in `memory.c`. It checks for read/write and write/write pairs to the same address with an $O(\mathsf{max}(m, n))$ scan of both access sets (pre-sorted). If any address a exists with $a \in M(t_i)$ and $a \in M(t_j)$ and $M(t_i)(a) = \mathsf{write} \vee M(t_j)(a) = \mathsf{write}$ then t_i and t_j are said to conflict, which I will denote $t_i \leftrightsquigarrow_S t_j$.

Whenever a conflict is identified, it also invokes the data race analysis (§3.4.4). It checks for `free()`/access conflicts as well as access pairs, effectively treating deallocation of a heap block as a "poisoning" write to its entire contents, which is considered to conflict with accesses to any address therein on the grounds that reordering may expose a use-after-free.

State space exploration

The core of the DPOR algorithm is implemented in `explore()` in `explore.c`.

[20] The original paper's definition includes a second criterion that, from s, a state s' exists such that both $s \xrightarrow{t_1 t_2} s'$ and $s \xrightarrow{t_2 t_1} s'$. This captures the memory independence relation, but computationally requires direct comparison of program states. *Stateless* model checkers compute memory conflicts separately from happens-before, to find and prune such identical states implicitly, as described in the next two subsections.

Definition. Given a transition sequence (execution, interleaving, preemption trace) $S = t_1 \dots t_n$, the DPOR algorithm identifies any number of alternate interleavings that must be tested. Each such interleaving I will denote in this section as $I_{ij} = (t_1 \dots t_{i-1}, T_j)$, where $t_1 \dots t_{i-1}$ is the common execution prefix shared between S and the new interleaving, and T_j is the thread ID to be scheduled after t_{i-1}, $T_j \neq T(t_i)$. [21] Landslide's implementation represents S as a list of `struct haxes`, defined in `tree.h`, each one representing a preemption point, or intermediate state between two transitions.

Identifying new interleavings. To find which alternate interleavings need to be tested, DPOR compares pairwise each pair of transitions t_i and t_j, $i < 4$, in the current interleaving S. If $t_i \rightarrow_S t_j$ then they cannot be reordered, and if $t_i \not\rightsquigarrow_S t_j$ then reordering them will produce a state already encountered in this interleaving; hence, DPOR marks new interleavings only when $t_i \not\rightarrow_S t_j$ and $t_i \leftrightsquigarrow_S t_j$ (`is_evil_ancestor()`). [22]

For each such pair, let s denote the state (preemption point) before t_i. Then:

- If $T(t_j)$ is runnable at s, return $I_{ij} = (t_1 \dots t_{i-1}, T_j)$ (`tag_good_sibling()`).
- Otherwise, there must be some third thread runnable at s; [23] then, return all $I_{ik} = (t_1 \dots t_{i-1}, T_k)$ such that $T_k \neq T(t_i)$ and T_k is runnable at s (`tag_all_siblings()`).

To summarize, DPOR identifies I_{ij}s which will (eventually) reorder each conflicting, concurrent transition pair in S to reach a (possibly) new program state not exposed in the current interleaving. Prior work [1, 46, 49] refers to the set of these I_{ij}s, for a given i, as the *persistent set* at the preemption point after t_{i-1}.

Tracking already-visited interleavings. Let $U(I_{ij})$ denote the sub-state-space (or subtree) beginning at the next preemption point reached after executing T_j after $t_1 \dots t_{i-1}$; in other words, the set of all sequences $S' = t_1 \dots t_{i-1}, u_i, \dots u_{n'}$ with $T(u_i) = T_j$. Landslide orders its search depth-first, so for any such U outside the current interleaving, either all or none of its S's will have been tested already. Therefore, to avoid repeating interleavings, Landslide need only store at each `struct hax` a list of threads such that their corresponding subtree U is fully explored, and can omit any non-constant-size information about the contents of that U (`struct hax_child`). Hence the memory cost of Landslide's DPOR implementation is $O(nk)$, k being the maximum runnable threads at any preemption point (which in turn is always single digits for model checking tests).

Choosing which new interleaving to test next. Among all interleavings chosen by DPOR not already marked as explored, Landslide chooses the one with the longest execution prefix matching the current S, to maintain the depth-first search invariant. (In the case of a tie, differing only by which thread to run, it chooses arbitrarily.) All other new interleavings are marked to explore later in their corresponding `struct hax`, and automatically included in the result of any future iterations of DPOR until they are tested.

[21] I describe T_j as a thread ID here, rather than as a thread transition, because the nature of the transition (its memory accesses, the subsequent state, etc) is unknown until actually executed.

[22] To aid intuition, consider the two extremes: if all transitions are related by happens-before, the program is not concurrent and no alternate interleavings are possible; if all transitions are memory-independent, the program exhibits full data isolation between threads and all schedules are observably equivalent.

[23] AFSOC $T(t_i)$ is the only runnable thread at s, then either $T(t_i)$'s execution at s enabled $T(t_j)$, or it enabled an intermediate transition (whether by $T(t_i)$ or a third thread) which in turn enabled $T(t_j)$. In either case t_i is transitively dependent with t_j, contradicting $t_i \not\rightarrow_S t_j$.

Because the one with the longest execution prefix was chosen to test next, all others must share their execution prefixes with it, preserving the $O(nk)$ memory bound described above.

Termination. When DPOR returns no new I_{ij}s not already marked in the set of visited subtrees, the exploration is complete.

Example

Although a superhuman reader may quickly reach intuitive understanding of complex algorithms from dense prose and mathematical notation alone, I also present here a friendly example of using DPOR on Figure 2.1's example program, whose original state space is shown in Figure 2.4. In this program both threads are always enabled, imposing no scheduling constraints, so memory conflicts alone will drive exploration. First, let us consider a single iteration of DPOR, applied after executing the first branch. The result is shown in Figure 3.3.

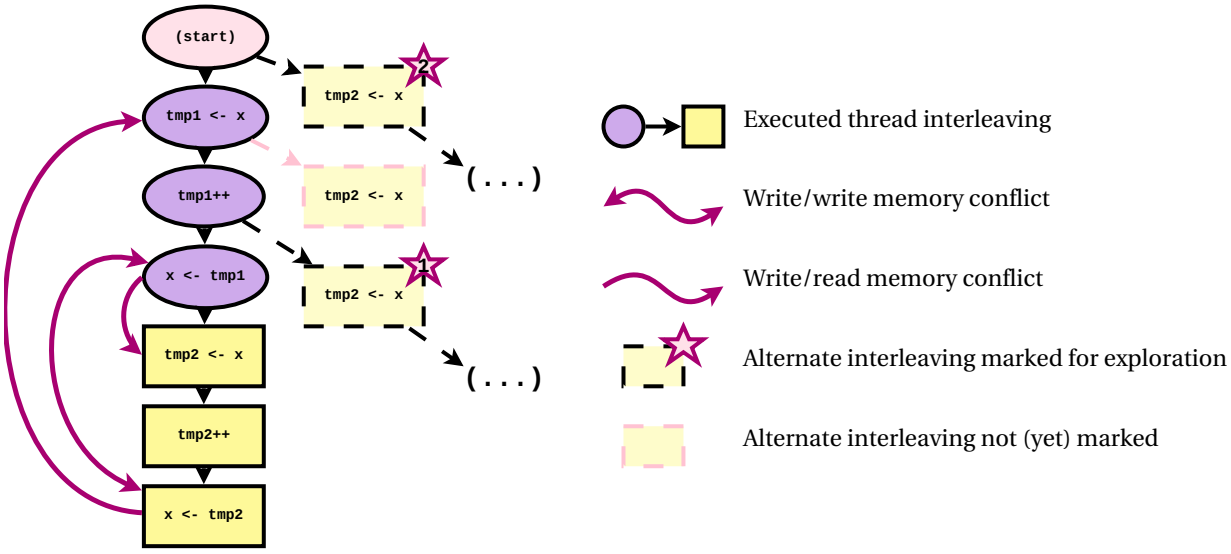


Figure 3.3: Result of a single iteration of DPOR.

In this interleaving, DPOR identifies 3 memory conflicts, two read/write and one write/write, among the threads' 4 accesses to `x`. For each such pair, it "marks" an alternate interleaving, which shall begin by preempting the thread of the first half of the conflict just before its execution thereof. The ultimate goal is to execute an interleaving which reorders the conflict, which may expose new behaviour. These marked interleavings form a work-queue which defines the exploration. DPOR consumes from this set in depth-first order (note the reversed order of ★1 and ★2) to avoid storing in memory any representation of exponentially-sized subtrees outside of the current branch. Note also that in ★2, the reordered `tmp2 <- x` is not directly part of the memory conflict which marked it, but it must be executed first to reach the conflicting `x <- tmp2`.

Now, let us run multiple iterations of DPOR to advance through the first half of the full state space shown in Figure 2.4(b). Figure 3.4 shows the outcome (with the previously-marked ★2, now re-labeled as ★3, having its subtree abbreviated).

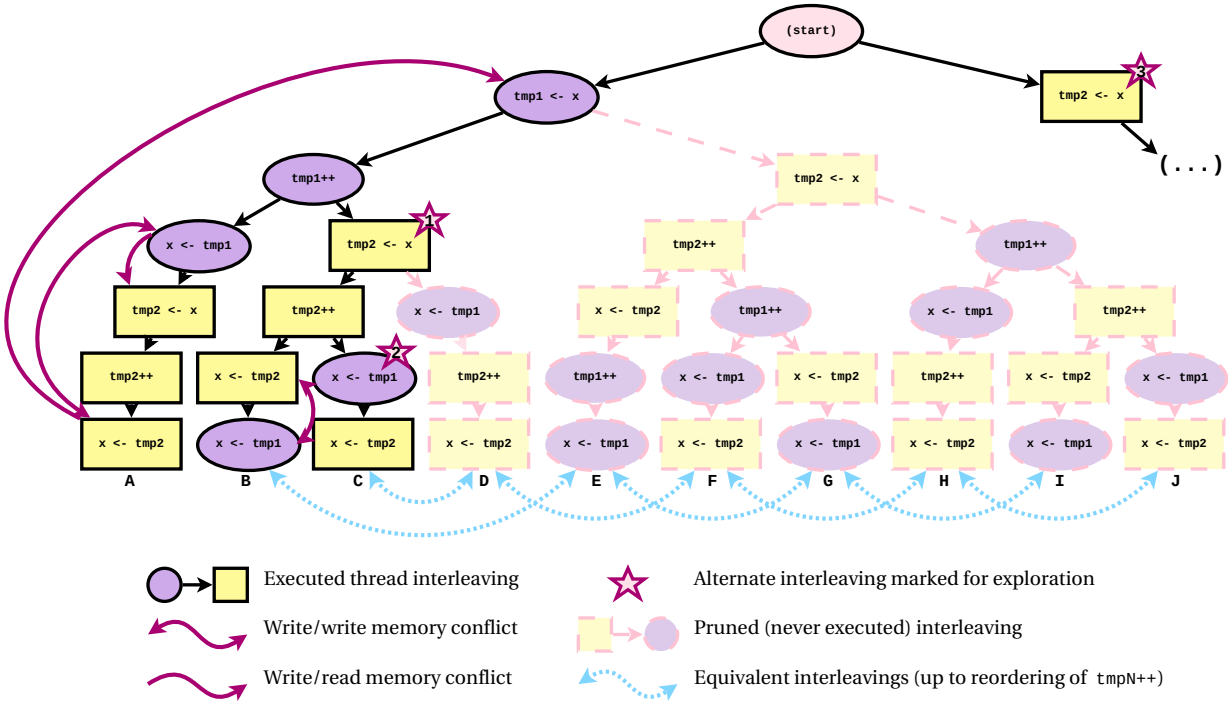


Figure 3.4: Result of 3 DPOR iterations, pruning 7 redundant branches.

After marking ★1 and ★3 from Figure 3.3's interleaving, now labeled A, DPOR advances to interleaving B, preferring to schedule the second thread before switching back to the first to ensure the memory access is properly reordered. From there, it identifies a new memory conflict, marks ★2, and advances to C, where it finds no memory conflicts that would mark anything not already marked and/or explored (memory conflicts that were already reordered in old branches are not highlighted with arrows). From C, ★3 alone remains in the work-queue, so DPOR advances to the second (symmetric) half of the state space, skipping (thereby pruning) branches D through J.

To see why branches D through J need not be tested, consider that each thread's `tmpN++` is a thread-local event, participating in no memory conflicts, and hence any two interleavings differing only by reordering `tmpN++`s must be equivalent. The dashed blue arrows denote such equivalences; note the two disjoint equivalence classes {B,E,G,I} and {C,D,F,H,J}, distinguished by the order of the two final `x <- tmpN`s. Note also that although B and C also have the same outcome (`x==1`), this depends on the *values* written to memory rather than *addresses* (and would change if one thread's `tmpN++` were a `tmpN+=2`, for example), which DPOR does not consider. Recent work [63] has extended DPOR to find such value-based equivalences, although is beyond this explanation's scope.

Finally, let us consider the final result after DPOR runs out of remaining unexplored marked branches, shown in Figure 3.5.

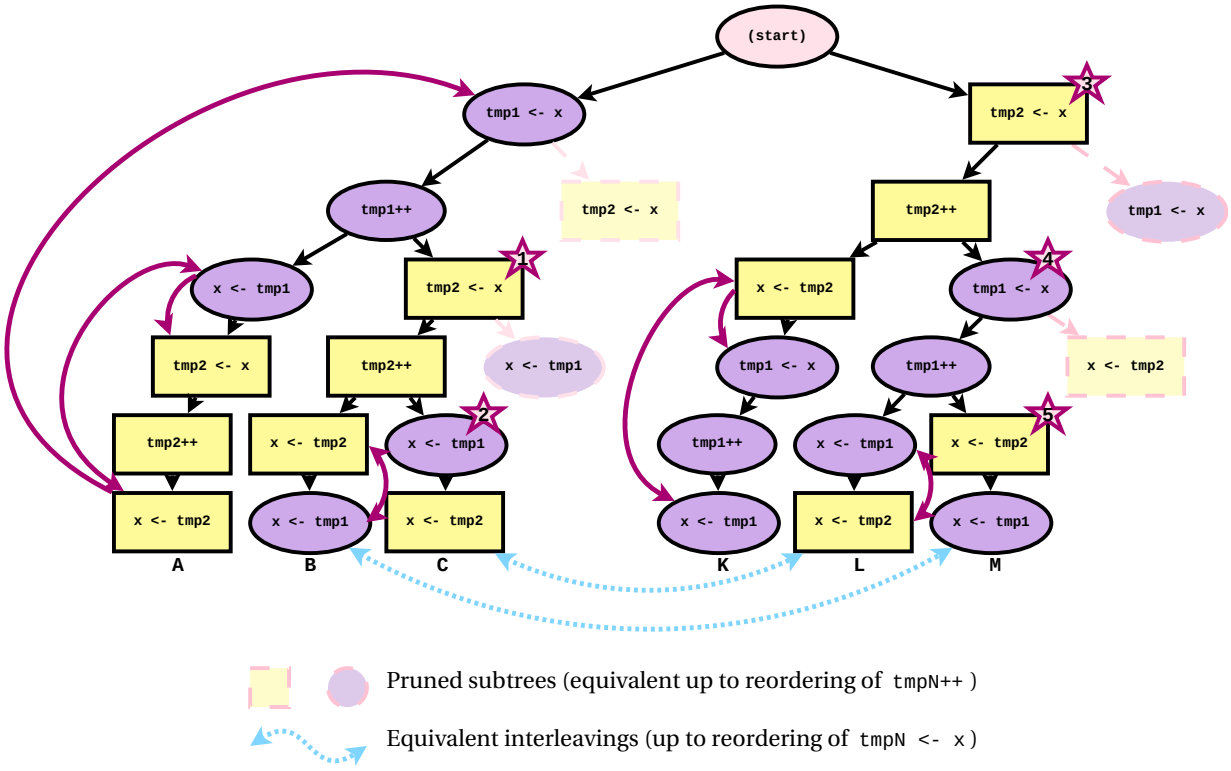


Figure 3.5: DPOR's termination state, having reduced 20 interleavings to 6.

Ultimately, the second half of the state space is pruned symmetrically. In general, the number of ways to interleave two threads executing N and M events each is given by $\binom{N+M}{N}$[24]; in this case, the original state space's size was $\binom{3+3}{3} = 20$. DPOR's reduction is characterized by replacing N and M with the number of *conflicting* events only; in this case, ignoring all `tmpN++` reorderings and testing only $\binom{2+2}{2} = 6$ branches.

Sleep set reduction

In the presence of non-conflicting transitions as well as conflicting ones, DPOR's approach as described so far can still end up testing equivalent interleavings. As the presence of more equivalence arrows in Figure 3.5 hints, its reduced subset state space still contains redundancy, arising from the fact that one pair of those 4 events is two reads, and hence not actually in conflict. Visual inspection shows that ★4, while locally justified in trying to reorder `tmp1 <- x` before `x <- tmp2`, effectively serves only to reorder it with `tmp2 <- x` relative to the first symmetric subtree (★1). In other words, even though DPOR marked each new branch with the intent only to reorder conflicting accesses, ★1 and ★4 contained interleavings equivalent up to independent reorderings anyway. Figure 3.6 summarizes the relevant interleavings to highlight one such equivalence.

[24] Generalizing to K threads, and simplifying to the same number N of events each, this formula becomes $\frac{n!k!}{n!^k}$.

```
T1: tmp1 <- x (read)
T2: tmp2 <- x (read)
T1: x <- tmp1 (write)
T2: x <- tmp2 (write)
```

(a) Original branch (C).

```
T2: tmp2 <- x (read)
T2: x <- tmp2 (write)
T1: tmp1 <- x (read)
T1: x <- tmp1 (write)
```

(b) Goal branch (K).

```
T2: tmp2 <- x (read)
T1: tmp1 <- x (read)
T1: x <- tmp1 (write)
T2: x <- tmp2 (write)
```

(c) Redundant branch (L).

Figure 3.6: Motivating example for the sleep sets optimization. Three of Figure 3.5's interleavings are highlighted, with the always-independent `tmpN++`s omitted for brevity.

Intuitively speaking, when DPOR entered the ★3 subtree, it did not "remember" which memory conflict it wanted to reorder T_1 around (i.e., that `x <- tmp2` should come before `tmp1 <- x`). Upon witnessing the conflict in the new (intended) order, it then tried to reorder it again, producing interleavings regrettably equivalent to ones already tested (all told, only `tmp2 <- x` and `tmp2++` having been reordered around `tmp1 <- x`). To "remember" the original purpose of testing subtree ★3, which was already fulfilled by testing K, DPOR can check just before tagging a new subtree (here, ★4) among all preceding transitions independent with the conflicting one (here, `tmp2 <- x` and `tmp2++` independent with `tmp1 <- x`) for an already-explored interleaving beginning with the target thread. If such exists, the new subtree is guaranteed to be equivalent to one already checked, and can safely be skipped.

Landslide implements this check in `equiv_already_explored()`, which checks (in this case after executing K), that if the first event to be reordered (here, `tmp1 <- x`) has already been tested in an equivalent reordering around any number of preceding events (here, `tmp2 <- x` and `tmp2++`), then the newly marked subtree is safe to prune. Note that this does not require storing any full subtrees outside of the current branch; only the subtree's root node need be saved to prove that an equivalent interleaving beginning with T_1 therein was already checked, preserving DPOR's $O(n)$ memory footprint.

This corresponds to the *sleep sets* optimization described by prior work [1, 46, 49], so named because it effectively puts T_1 "to sleep" until after the true conflicting access of `x <- tmp2`. Landslide's implementation differs from prior work, which explicitly tracks sets of reordered threads and expected conflicting accesses, by instead identifying where the reduction should occur during subsequent DPOR iterations. This approach also relies on the search ordering strategy (`arbiter_choose()`) to prefer scheduling the thread previously chosen for reordering by DPOR, to ensure the conflicting access happens before the preempted thread gets a chance to run again. Further optimizations such as *source sets* and *wakeup trees*, which prior work has shown achieve optimality (i.e., executing exactly one interleaving per equivalence class) [1] are not yet implemented. To the best of my knowledge, they provide further reduction only in cases of 3 or more threads; I suspect (without proof) that sleep-set DPOR is optimal for 2 threads.

Chapter 4's experiments and Chapter 5's user studies were conducted before this opti-

mization was implemented. Note that its absence has no bearing on DPOR's soundness, only its efficiency, and that Landslide showed good bug-finding performance even without it. Chapter 6's experiments include this optimization, because its presence was required to fairly compare the other reduction strategies presented therein.

3.4.3 State space estimation

For both the user's convenience and for Quicksand's prioritization algorithm (§4.2), Landslide attempts to guess how big partially-explored state spaces will ultimately end up being upon completion. Because the backtracking implementation uses checkpointing rather than replaying similar interleavings' shared execution prefixes from the beginning §3.3.5, the total number of interleavings (i.e., leaf nodes in the execution tree) must be estimated separately from the total runtime (i.e., sum of all edge weights in the tree).

As a concrete example, consider the state space of Figure 3.5, and suppose each transition to the next preemption point takes 1 second to execute. While the first branch executes in 6 seconds, the second branch, sharing the first transition as a common prefix, takes 4 seconds, and the one after that only 2; the state space being ultimately completed in 24 seconds. Even with perfect hindsight, naïvely multiplying the total interleavings (6) by the total execution time per branch (6) would double-count common prefixes and grossly overestimate (36) the total runtime.

Hence, Landslide uses two differently suitable algorithms for each of size and runtime estimation: the Weighted Backtrack Estimator (WBE) and the Recursive Estimator (RE), respectively, first introduced in [72] and later adapted to DPOR by [129]. In principle, both calculate the current progress as a proportion of the expected total by counting how many branches DPOR has marked for future exploration (§3.4.2) and assuming the sizes of their resulting subtrees are predicted by the known sizes of similar already-explored subtrees. In practice, the calculation strategy differs between the two approaches, which can occasionally result in drastically differing outputs (§6.3.2).

Implementation-wise, Landslide reports size estimates as both the percentage and as a total number of branches, and time estimates as an ETA. Quicksand's `-v` option (§3.1.2) will cause it to print them each time a new interleaving is tested; for example:

```
[JOB 1] progress: 66101/94825 brs (69.708252%), ETA 13m 37s (elapsed 46m 10s)
```

Both estimates are computed simultaneously in `_estimate()` in `estimate.c` (which, I might add, is well-commented in case the following prose is insufficient). Chapter 8 will discuss their limitations and some opportunities for future improvement.

Size (Weighted Backtrack) estimation

The WBE, used to estimate total number of interleavings, computes the *proportion* of the total size that the already-explored branches are expected to comprise, using DPOR's workqueue to anticipate how many unexplored marked branches remain. This serves as a progress bar [103] that represents the estimated percentage towards completion, approaching 100% (not necessarily monotonically) as exploration continues.

Summarizing prior work's formal definition [129], the proportion at a terminal node v_n[25], preceded by an execution sequence $(v_1 \dots v_{n-1})$, is computed as:[26]

$$\mathsf{proportion}(v_1 \dots v_n) = \prod_{i=1}^{n-1} \frac{1}{|\mathsf{marked\ children}(v_i)|}$$

where $\mathsf{marked\ children}(v_i)$ is the number of enabled thread transitions at v_i which have either already been explored or been marked by DPOR. Then, the total estimate is given as the sum over all branches $b = (v_1 \dots v_n)$ explored so far:[27]

$$\mathsf{estimate} = \frac{1}{\sum_{b \in B} \mathsf{proportion}(b)}$$

It is easy to see how these formulas might fit into DPOR's incremental search procedure: at the end of each branch compute its proportion and add it in to a global estimate value. However, DPOR may tag new branches to explore that would affect past branches' proportions, requiring them to be recomputed, which is not feasible without storing the entire exponentially-sized tree in memory. Instead, Landslide also stores per-subtree estimates at intermediate v_i nodes, $1 < i < n$, along the current branch. Whenever DPOR marks a new kth branch for exploration at some v_i its estimate is multiplied by $(k-1)/k$ to retroactively adjust all past branches' proportions contributing to that estimate. The change is also propagated to its sub-subtrees, whose estimates must also incorporate the new marked children value. This allows Landslide to update the global estimate after each branch in $O(n)$ time and memory, without recomputing past branch proportions individually.

Runtime (Recursive) estimation

The RE, used to estimate total execution time, computes at each node the expected time to execute all subtrees rooted at children of that node, assuming unvisited subtrees' times will be an average of their visited siblings. This estimate at the root node, minus the current time elapsed so far, serves as a guess at how long until completion. Let $\mathsf{usecs}(v_i)$ denote the time elapsed during execution of the transition $v_{i-1} \to v_i$. Then a node's estimate is given by:

$$\mathsf{estimate}(v_i) = \mathsf{usecs}(v_i) + \frac{|\mathsf{marked\ children}(v_i)|}{|\mathsf{explored\ children}(v_i)|} \sum_{v_j \in \mathsf{explored\ children}(v_i)} \mathsf{estimate}(v_j)$$

Like the WBE, whenever DPOR tags a new kth child at some v_i, its estimate is multiplied by $k/(k-1)$ (note the reciprocal of before) to retroactively re-weight previously

[25] Prior work [72, 129] refers to this instead as *probability*, i.e., the probability that the node will appear in a branch chosen uniformly at random from the completed tree. I find "proportion" to be more illuminating on how the algorithm works.

[26] Simplified from [129]: the missing $F(v_i)$ is 0, using the empty fit strategy.

[27] Simplified from [129]: $t(b)$, the time for each branch, is 1, because we are counting them.

explored subtrees' estimates. Unlike the WBE, this change does not need to be propagated to descendant subtrees' estimates. This estimate also takes $O(n)$ time and memory.

Example

To illustrate how the two estimators can under-estimate the total tree size and/or diverge from each other, consider the state space from Figure 3.5, of size 6. Suppose for RE that each transition takes 1 second to execute.

1. After branch A, two tags exist, ★1 and ★3. Under WBE, the subtree estimate at `tmp1++` will first be 1/2 (half its children being fully explored), and the root estimate will be 1/4, half that, which is propagated back down to `tmp1++`, becoming also 1/4. Dividing the current progress (1) by that yields 4 total branches, an underestimate.

   Under RE, the estimate at `tmp1++` will be 9 seconds (incorrectly assuming ★1's subtree will be 1 branch), and the root estimate will be 20 seconds, an underestimate.

2. After branch B, ★2 is now marked. Under WBE, the subtree estimate at `tmp2++` is 1/2, which at `tmp1++` is then divided by its marked children and added to its estimate, yielding 3/4. Note that it has "forgotten" that only branch A, alone, contributed to its original 1/2, rather than two branches as in this subtree. The root and `tmp1++`'s subtree estimates are updated (and propagated down) to 3/8. Dividing the current progress (2) by that yields 5.33 branches, an underestimate.

   Under RE, the estimate at `tmp2++` is 5 seconds, the estimate at `tmp1++` is updated to 11 seconds, and the root estimate to 24 seconds, accurate.

3. After branch C, nothing new was marked. The subtree estimate at `tmp2++` is 1 (having been completely explored) and the root estimate is 1/2. Dividing the current progress (3) by that yields 6, accurate.

   Under RE, no estimates change from after B.

4. After branch K, ★4 now exists. `tmp2++`'s subtree estimate is at first 1/2, then the root estimate and it get updated to 3/4. Dividing into the current progress (4), 5.33, an underestimate.

   Under RE, the estimate at `tmp2++` is 9 seconds, and the root estimate is 22 seconds, an underestimate.

5. After branch L, ★5 joins the party. `tmp2++`'s estimate is updated to 3/4, and the root estimate ultimately becomes 7/8. Dividing into the current progress (5), 5.7, an underestimate.

   Under RE, the estimate at `tmp2++` is 11 seconds, and the root estimate is 24 seconds, accurate.

6. After branch M, both estimators have perfect information and converge to accuracy.

To illustrate how the estimators can over-estimate the total tree size, consider the same state space, except with the ★4 subtree also pruned by DPOR's sleep sets extension (§3.4.2); i.e., only branches A, B, C, and K remain, with an 18 second execution time. Both estimators' behaviour is identical through branch C, only now WBE's prediction hap-

pens to be accurate at A (although for the wrong reasons), but overestimates at B and C, while RE's predictions are all overestimates. As before, both reach perfect accuracy upon completion, now occurring at K. Intuitively speaking, the estimators underpredict when DPOR keeps finding new branches to tag as it makes progress, and overpredict when sleep set reduction achieves extra pruning on right subtrees. Not shown in this example, interleaving-dependent control flow can, of course, beget unexpected state space structure in essentially arbitrary other ways.

3.4.4 Data race analysis

Whenever a memory conflict is identified for DPOR as described above, the access pair's corresponding locksets and/or happens-before edges are checked to determine if it's also a data race. Note the distinction: DPOR memory conflicts indicate that two thread transitions, if reordered, could produce different behaviour, even if all accesses therein are adequately synchronized; while a data race indicates furthermore that the two threads can be interleaved precisely at the moment of one or both accesses, supposing that a new preemption point were introduced to split one or both transitions in half.

The core of the comparison is in `check_locksets()` in `memory.c`. It checks each DPOR memory conflict's locksets, for limited happens-before, and happens-before edges, for pure happens-before (§2.3.2).

Limited Happens-Before

Conditions #1, #2, and #4 defined in §2.3.2, provided the Limited Happens-Before definition for #4, coincide with DPOR's version of happens-before described in the previous section. Hence all that remains to be checked is #3, the set of locks held by each thread at the time of access.

Routines for recording lockset changes and computing set intersection are found in `lockset.c`. Apart from standard data structure manipulation, one algorithmic point of note is that locks are distinguished by types in addition to address. This allows (e.g.) mutexes stored as part of the implementation of semaphores to protect a different set of accesses than are protected by the semaphore they implement.

Pure Happens-Before

In Pure Happens-Before, condition 4 is replaced with the traditional distributed systems notion of Happens-Before [81]. Landslide implements this via the vector clocks approach described by FastTrack [45]. I refer the reader interested in the vector clock algorithm itself to the FastTrack paper, limiting discussion here to Landslide's corresponding implementation of each inference rule.

I use the Djit+ rules for reads and writes rather than the FastTrack ones, even though they more often incur $O(n)$ runtime in the size of the vector clocks: because Landslide tests should be limited to few threads in order to manage the state space size, n is always in the single digits, so I optimize for code simplicity.

- Reads and writes (`memory.c`)
  - DJIT+ READ/WRITE SAME EPOCH - `vc_eq()` case of `add_lockset_to_shm()`
  - DJIT+ READ/WRITE - `vc_happens_before()` case of `check_locksets()`
- Synchronization (`schedule.c`)
  - FT ACQUIRE
    - `kern_mutex_{,try}locking_done()` cases of `kern_update_state_machine()`
    - `user_mutex_{,try}lock_exiting()` cases of `user_update_state_machine()`
    - `cli` case of `kern_update_state_machine()` (Pintos only)
    - `cli/sti` lock handoff case in `sched_update()` (Pebbles only)
  - FT RELEASE
    - `kern_mutex_unlocking()` case of `kern_update_state_machine()`
    - `user_mutex_unlock_entering()` case of `user_update_state_machine()`
    - `sti` case of `kern_update_state_machine()` (Pintos only)
    - `cli/sti` lock handoff case in `sched_update()` (Pebbles only)
  - FT FORK - `agent_fork()`
  - FT JOIN - `sched_unblock()` case of `kern_update_state_machine()` (Pebbles only; Pintos case is handled by above `cli/sti` cases in context switch)

3.4.5 Iterative Context Bounding

Iterative Context Bounding [101] is a state space exploration strategy that prioritizes interleavings with fewer total preemptions first. Let $P(S)$ denote the number of preemptions in an execution sequence S. Then, to summarize in pseudocode a naïve exploration of some state space U as:

Algorithm 1: Straightforward exploration ordering

```
1 foreach S ∈ U do
2 |   Execute(S)
3 end
```

ICB's approach could likewise be summarized as follows:

Implementation

First of all, note that Algorithm 2 is structured in a way that repeats interleavings with fewer than n preemptions that have already been checked in previous iterations of the outer loop. This is because the number of preemptions in each branch is not known in advance; rather, the state space must always be explored in an overall depth-first approach, at best skipping too-preemptful interleavings as they are encountered. As simple as it

Algorithm 2: ICB exploration ordering

```
for B ∈ [0..max_P(U)] do
    foreach S ∈ U, P(S) ≤ B do
        Execute(S)
    end
end
```

would be to state "**foreach** $S \in \mathsf{sort}_P(U)$" in pseudocode, implementing such an ordering would be much less straightforward.

Therefore, Landslide's ICB implementation combines with DPOR when tagging new branches to explore at the end of each branch: just as DPOR skips alternate interleavings that are memory-independent, ICB further filters interleavings requiring more preemptions than the current bound out of the to-explore set. The macro `ICB_BLOCKED`, defined in `schedule.h`, decides if a given thread would require a preemption beyond the current bound to switch to[28]. The DPOR implementation then checks, for some I_{ij} it wants to mark for exploration, whether `ICB_BLOCKED`(T_j) at the state after t_i, and skips it if so (`tag_good_sibling()`/`tag_all_siblings()`).

Then, the entire state space is repeated with increasing bound until no such are filtered. This core ICB loop appears in `time_travel()` in `landslide.c`. Although not explicitly structured as a C-style loop in the code, it resets Landslide's progress through the state space, allowing exploration to continue until it finally observes all interleavings to have fewer preemptions than the bound.

Complexity

If the search is terminated early after reaching a predetermined fixed bound for B, ICB in principle reduces the state space from exponentially-sized[29] in both K, the number of threads, and N, the number of events, to still exponential in K (typically small) but only polynomial in N (typically large). Under a preemption bound of B, there are only $B + K$ opportunities for context switching[30], so the corresponding state space size is at most $\binom{KN}{B}(B+K)!$. All N-related factors therein are bounded above by N^B.

Prior work often recommends 2 for such a cutoff [101, 135, 141], although §4.3's larger dataset suggests 3 would be considerably more thorough. On the other hand, any finite such bound can provide only a heuristic verification guarantee anymore. Preserving the full formal verification, i.e., continuing iteration until $B = \mathsf{max}_P(U)$ not only remains exponential in N, but also introduces a factor of $\mathsf{max}_P(U)$ repeated work. Future work could memoize already-tested interleavings so that each iteration of B could test only

[28] Since "voluntary" context switches (e.g. arising from `yield()`) are often necessary for correct execution, `ICB_BLOCKED` does not count such switches towards the preemption count. Therefore, within a certain preemption bound B, interleavings with more than B context switches may still be tested.

[29] Combinatorial, to be precise; see §3.4.2.

[30] This K appears from the "mandatory" context switches at thread exit; more of which could also be introduced from blocking synchronization.

those schedules with exactly B preemptions, restoring the original B-independent (but still exponential) complexity.

Bounded Partial-Order Reduction

Prior work [24] has shown that when combined with DPOR to prune equivalent interleavings, DPOR's reduction might not be sound with respect to the subset of U under $P \leq n$. That work introduced Bounded Partial Order Reduction (BPOR), a compatibility extension to DPOR for ICB to address this problem.

To summarize, when DPOR identifies some interleaving I_{ij} to test, it may not be possible to execute $T(t_j)$ after t_i without exceeding the current preemption bound. However, there may exist another interleaving $J_{i'j}$ within the bound which runs $T(t_j)$ before t_i. If a DPOR implementation naïvely configured with ICB simply skipped I on account of the preemption bound, J may not get marked for exploration from any other iteration and/or pair of conflicting transitions. Even though restricting the state space to a certain maximum preemption bound is already unsound in terms of losing full interleaving coverage, failing to test even J would be a failure of DPOR itself to soundly prune the already-reduced state space defined by that bound. Hence the need for BPOR, to ensure that if such an alternative J to I exists within the bound, it gets marked for exploration immediately.

To implement BPOR, whenever `ICB_BLOCKED` causes DPOR to skip an interleaving, Landslide searches all transitions $t_k \in S$ such that $t_k \prec_S t_i$ and $T(t_k) = T(t_i)$ and $\neg\{\exists t_l \in S$ such that $t_k \preceq_S t_l \prec_S t_i$ and $T(t_l) = T_j\}$ (`stop_bpor_backtracking()`). All I_{kj}s which can be tested within the preemption bound are marked instead of I_{ij} (`tag_reachable_aunts()`). The reader interested in further algorithm details and the corresponding soundness proof is referred to [24].

3.4.6 Heuristic loop, deadlock, and synchronization detection

Chapter 4

Quicksand

4.1 Preemption points

4.2 Iterative Deepening

4.2.1 Soundness

4.3 Evaluation

Chapter 5

Education

Concurrency is taught in as many different ways as there are systems programming classes at universities which teach the subject. Yet one thing they all have in common is presenting the concurrency bug as some elusive menace, against which humanity's best weapon is mere random stress testing. This chapter will prove stateless model checking's mettle as a better alternative in the educational theatre.

While the previous chapter demonstrated Landslide's bug-finding power compared to prior MC techniques in a controlled environment, whether it offers pedagogical merit in the hands of students and/or TAs is a separate question. And while my MS thesis [12] showed that students could annotate P3 Pebbles kernels and thence use Landslide to debug them, the annotations alone required 2 hours of effort on average per user, meaning the only students who could benefit were the ones already succeeding enough to have such free time. Since then, I have extended Landslide with a fully-automatic instrumentation process for Pebbles thread libraries (P2s) (§5.1.2) and Pintos kernels (§5.2.2) to improve its accessibility.

I have run several user studies in the Operating Systems classes at Carnegie Mellon University (CMU), University of Chicago (U. Chicago), and Penn State University (PSU), wherein students get to use Landslide to find and diagnose their own bugs during the semester. At CMU, I analyzed logs and code snapshots taken as students used Landslide during P2 (§5.3.1), as well as the grades ultimately assigned after students who either did or didn't use Landslide submitted their projects (§5.3.2). At CMU and PSU, I surveyed students on their experience after submitting their Landslide-debugged P2s (§5.3.3). At U. Chicago, I collaborated with a TA to check submitted Pintos kernels, then returned any resulting bug reports to students (§5.3.1) and likewise surveyed them on the quality of the diagnostic output (§5.3.3).

5.1 Pebbles

This section presents the user studies done in CMU's 15-410 in semesters Fall 2015 to Spring 2018, taught by David Eckhardt, and in PSU's CMPSC 473 in Spring 2018, taught by Timothy Zhu. In both cases the instructors assisted to introduce me during the guest

lecture and to distribute the recruiting emails; TAs were not involved. The in-house user study has CMU IRB approval under study number STUDY2016_00000425, and the external user study under STUDY2017_00000429.

5.1.1 Recruiting

Since the Spring 2015 semester I have given a guest lecture in 15-410 to recruit students to participate in the user study. The 50-minute lecture is given 1 week into the 2.5-week-long P2 project, approximately when the students should be getting child threads running in `thr_create()` and experiencing concurrency bugs for the first time. It introduces the research subject abstractly using an example "Paradise Lost" bug from a previous lecture [38], explains how Landslide works concretely, shows a short demo of effortlessly using Landslide to find the example bug, and provides the necessary IRB legalese about the risks and benefits of participation. The most recent lecture slides are available on the course website at `http://www.cs.cmu.edu/~410-s18/lectures/L14_Landslide.pdf`, and all semesters' editions at `https://github.com/bblum/talks/tree/master/landslide-lecture`.

The PSU version of the lecture is available at `http://www.contrib.andrew.cmu.edu/~bblum/psu-lecture.pdf` as well as under the github link above. Being a 70-minute lecture slot rather than 50, I extended the demo to both find and (attempt to) verify a fix for two bugs: one a simple data race and the other the more complicated Paradise Lost bug as above. After finding each bug, I demonstrated using Landslide on a fixed version of the code to show how it proves the test case correct by completing all state spaces, or (in the case of Paradise Lost) suffers an exponentially-exploding state space. Not that I scientifically measured it or anything, but this extended demo seemed to help students more clearly understand Landslide's intended workflow, at the cost of about 10-15 extra minutes of lecture time.

At both schools students then signed up using a Google form I emailed them, which upon completion linked them to the Landslide user guide, which is available at `http://www.contrib.andrew.cmu.edu/~bblum/landslide-guide-p2.pdf` (CMU version) and `http://www.contrib.andrew.cmu.edu/~bblum/landslide-guide-psu.pdf` (PSU version) and `https://github.com/bblum/talks/tree/master/irb` (both versions).

During the last week of P2 at CMU, I held several "Landslide clinic" sessions (basically office hours, but given a different name to remind students to limit themselves to questions a normal TA couldn't answer), where students could receive in-person technical and/or moral support. Collecting study information during these sessions was not included in the IRB protocol. In the PSU study, I had returned to Pittsburgh shortly after giving the lecture, so technical support for PSU students was limited to email correspondence.

5.1.2 Automatic instrumentation

As described in §3.1.1, all setup from the user's point of view is handled through the `p2-setup.sh` script[1]. It, its helper scripts (§3.3.9), and the `landslide` script itself contain several checks to prevent studence from accidentally misusing Landslide in ways that could produce mysterious crashes, false bug reports, and so on (the need for each one, as the reader might imagine, discovered through bitter experience). These include:

- `p2-setup.sh` checks if the directory argument correctly points at the top-level P2 basecode directory rather than any subdirectories such as `user/libthread/`.
- `check-need-p2-setup-again.sh` checks if any source files in the original P2 source directory (the argument supplied to `p2-setup.sh`), in case the student hoped to fix some bug and verify their fix but forgot to re-run the setup script.
- `landslide` checks the supplied test name matches one of the endorsed Landslide-friendly tests (students love trying to run Landslide with `racer`, `largetest`, or even the string `OPTIONS`).
- `landslide` checks if any other instance of itself is simultaneously running in the same directory, and if so, refuses to do so and advises the student to `git clone` the repository afresh for simultaneous use[2].

Landslide also includes several P2-specific instrumentations and features to cope with various student irregularities:

- Quicksand emits different combinations of `within_function/without_function` directives for Landslide depending on the name of the test. For example, for `paradise_lost` Landslide will not preempt in a function named `critical_section()`, which the test case uses to protect an internal counter used to detect the bug; and it will not preempt in any of the `thr_*()` thread library API functions for tests intended to target just the concurrency primitives. In future work this could be improved as annotations to be placed inside the test case code itself.
- Landslide finds ad-hoc synchronization patterns, such as `while (!flag) yield()` or `while (xchg(...)) continue`, which students often open-code rather than using the prescribed synchronization API, and treats them as synchronization points as described in §3.4.6.
- Landslide finds "too suspicious" spinwait-loops in mutex implementations which are neither yield- nor `xchg`-loops (as described above), which would ordinarily be classified as infinite loop bugs, and reports them with a suggestive message (`undesirable_loop_html()` in `landslide.c`) referring the student to the appropriate lecture material [39].
- The `landslide` wrapper script logs the time and command-line options of invocation and captures a snapshot of the student code and results of the test and saves them to AFS (CMU's network file system) after each run.

[1] PSU's version is called `psu-setup.sh`; in this section `p2-setup.sh` refers to both unless otherwise noted.
[2] This is ironically implemented with a non-atomic lock file and should really be using `flock` instead.

5.1.3 Test cases

Landslide ships with several "approved" test cases, i.e., programs copied from, derived from, or at least vaguely resembling the tests distributed with P2, which I curated to produce concurrent behaviour suitable for stateless model checking. Some tests are crafted to target specific bugs which, from personal experience as a TA, are common in many student submissions; others are crafted to exercise generally concurrency-heavy code paths and uncover any number of unforeseen problems. Many use some of the features/annotations described in §3.1.4.

The following tests were released to CMU students:

- `broadcast_test`: Tests the `cond_broadcast()` signalling path with a single waiter.
- `mutex_test`: Tests student mutexes under 2 threads with 2 iterations (the 2nd iteration serves to expose problems with `mutex_unlock()` as well as `mutex_lock()`). This test uses the `TESTING_MUTEXES` described in §3.1.3 to enable data-race preemption points within the mutex implementation.
- `paradise_lost`: Written for the sake of the Landslide lecture demo (§5.1.1). Tests for the Paradise Lost bug by attempting to break mutual exclusion.
- `paraguay`: Copied directly from the P2 test suite; tests for proper handling of seemingly "spurious" wakeups in `cond_wait()`. Written by Michael Sullivan.
- `rwlock_downgrade_read_test`: Copied directly from the P2 test suite; tests for mutually-exclusive and deadlock-free `rwlock_downgrade()`. Written by me (as a TA).
- `thr_exit_join`: Copied directly from the P2 test suite; tests for a variety of problems between `thr_exit()` and `thr_join()`, but especially for memory issues pertaining to stack deallocation.

The following tests were released to PSU students, in addition to the ones above:

- `atomic_compare_swap`: Tests the `cmpxchg` assembly function for being properly atomic. Uses the `magic_*` global variables described below, and invokes `vanish()` directly, to avoid requiring the student to implement `thr_join()`/`thr_exit()` before being able to run this test.
- `atomic_exchange`: As above for `xchg`.
- `atomic_fetch_add`: As above for `xadd`.
- `atomic_fetch_sub`: As above for `xadd`.
- `broadcast_two_waiters`: As `broadcast_test`, but uses two waiting threads to ensure both get signalled.

The tests can all be viewed at `https://github.com/bblum/landslide/tree/master/pebsim/p2-basecode/410user/progs`.

5.1.4 Survey

Starting in Fall 2017, I sought to gauge the students' personal opinions on their experience with Landslide, in addition to simply counting from the automatic snapshots how many bugs were found. Shortly after the P2 submission deadline, I asked participants to answer several survey questions, reproduced below.

1. How many bugs did Landslide help you find in your code? (Please indicate a number.)
2. How many of the bugs you found with Landslide do you believe you fixed before submitting your project? (You may answer "all", "none", or a number.)
3. How many of the bugs you found with Landslide did you verify you had fixed by running Landslide again to make sure the bug was gone? (You may answer "all", "none", or a number.)
4. In addition to the bugs Landslide found, did it report anything that you believe was NOT a bug? For example, Landslide printed an execution trace that was actually impossible, or Landlside reported a bug about some behaviour that was actually allowed by the P2 specification. (If so, please describe.)
5. I found Landslide's debugging output easy to understand. (Multiple choice from strongly disagree to strongly agree.)
6. It's easier to diagnose the root cause of a bug with Landslide than with a stress test (e.g. juggle). (Multiple choice from strongly disagree to strongly agree; plus "Not sure" and "Easier for some bugs but harder for others")
7. I felt the time I saved by having Landslide to help debug was worth the time it took me to learn how to use Landslide. (Multiple choice from strongly disagree to strongly agree.)
8. I feel that by using Landslide I learned to understand concurrency better. (Multiple choice from strongly disagree to strongly agree.)
9. Suppose after you submitted your project, we gave you 100 CPU-hours on the cloud provider of your choice to test it. Then we extended the project deadline by a day for you to use the results to fix bugs and get partial credit. How would you divide that CPU time between the staff-provided stress tests and Landslide? (Multiple choice: 0/10/.../100 CPU-hours on Landslide, 100/90/.../0 CPU-hours on stress tests.)
10. If I found out next semester that a friend of mine (or a student in my degree program) were taking OS, I would recommend that they should probably invest some time during the project to learn Landslide and try to find bugs with it. (Multiple choice from strongly disagree to strongly agree.)
11. Regarding the previous question, why or why not?

The following questions were served only on the CMU version of the survey.

12. Did you answer this survey together with your partner, or on your own while they were busy? (If you both have time for it, please try to submit one survey together.)

(Multiple choice: together or alone)

13. Your andrew ID
14. Your partner's andrew ID (if any)

The following questions were served only on the PSU version of the survey.

15. Any feedback on how Landslide's user interface could be improved / made easier to use or understand? (setup process, messages printed while running, or the execution trace / stack traces emitted after a bug is found?)
16. Your PSU username

5.2 Pintos

This section presents the user study done in U. Chicago's CMSC 23000 class in the Fall 2017 semester, taught by Haryadi Gunawi. Kevin Zhao, the TA, assisted to run Landslide on student submissions and to distribute recruiting materials and testing results. The study has CMU IRB approval under study number STUDY2017_00000429.

5.2.1 Recruiting

For this study students were recruited remotely via email. After each of the *threads* and *userprog* project deadlines (§2.4.2), CMSC 23000 staff sent students an email inviting them to volunteer to receive Landslide's bug reports, disclaiming that it did not represent part of the official grading process but could help improve their future submissions.

5.2.2 Automatic instrumentation

As described in §3.1.1, all setup from the user's point of view is handled through the `pintos-setup.sh` script. It and its helper `pebsim/pintos/import-pintos.sh` perform most of the same sanity checks as listed in §5.1.2, then applies the patch `annotate-pintos.patch` (plus several more hacks in the script itself) to insert the `tell_landslide()` annotations (§3.2.2) into the student's kernel code. The following tricks were developed after trial-and-error on student submissions from the same semester, and serve to make sure the annotations apply consistently to (almost) all variations of commonly-submitted code.

- Finds the declaration of `ready_list`, the scheduler runqueue declared by the basecode, and detects if the student has modified to be an array of lists rather than a single one. If so, defines the length of that array in a macro to be used by `is_runqueue()` (part of the patch described below). Either way defines a function `get_rq_addr()` to return the address of the (first) list.
- Changes the basecode's definition of `TIME_SLICE` from 4 to 1 (units of timer ticks) so Landslide's timer injection will properly drive the context switcher.

- Inserts `tell_landslide_forking()` into `thread.c` (using `sed` rather than the patch, described below, because it must go in a function which students have to implement, which is likely to disturb the context and make a patch fail).
- Adds the new `priority-donate-multiple` test.
- Applies the `annotate-pintos.patch` patch to the imported student implementation, which:
  - Adds `tell_landslide_thread_on_rq()` and `tell_landslide_thread_off_rq()` annotations to `list_insert()` and `list_remove()` respectively (in `lib/kernel/list.c`, which the students don't modify), which check whether the argument list is the scheduler runqueue using a helper function `is_runqueue`, which in turn uses `get_rq_addr()` and `READY_LIST_LENGTH` described above.
  - Modifies the existing `priority-sema` and `alarm-simultaneous` tests to be more Landslide-friendly.
  - Inserts the `tell_landslide_sched_init_done()`, `tell_landslide_vanishing()`, and `tell_landslide_thread_switch()` annotations in the appropriate places (which the students generally do not modify).
- Detects if the student has renamed the `elem` field of the TCB struct, and if so renames its use in `is_runqueue()` (described above) correspondingly.
- Detects if the student has renamed the `cur` (currently running thread) variable in the context switcher, and if so renames it back.

5.2.3 Test cases

Like the P2 tests, the Pintos test cases are either hand-picked from the provided unit tests, with an eye for which will produce interesting concurrent behaviour, or created using a TA's intuition for the most likely student bugs. The following tests are approved to be Landslide-friendly:

- `priority-sema`: Modified to be Landslide-friendly from the basecode, for *threads*. Creates two child threads to wait on a semaphore and signals them. Replaces threads with different priorities (originally chosen to produce deterministic output which the test checked for) with threads of the same priority.
- `alarm-simultaneous`: Modified to be Landslide-friendly from the basecode, for *threads*. Creates two child threads which each invoke `timer_sleep()` for a different amount of time. Number of (threads,iterations) reduced from (3,5) to (2,1).
- `wait-simple`: Unmodified from the basecode's version, for *userprog*. Userspace process `exec()`s a child process, which immediately exits, and `wait()`s on it.
- `wait-twice`: Unmodified from the basecode's version, for *userprog*. Slightly more complicated version of `wait-simple`, intended to expose failure-path bugs if the former finds no easier ones.

- `priority-donate-multiple`: Written by Kevin Zhao, TA at U. Chicago, for *threads*. Tests for a priority donation race during `lock_release()` in which a thread holding a lock can accidentally keep a contending thread's donated priority after finishing releasing it.

The (unpatched versions of) the first four tests are available at `https://github.com/bblum/pintos` (a fork of the main Pintos basecode repository). The fifth test is available at `https://github.com/bblum/landslide/blob/master/pebsim/pintos/priority-donate-multiple.c`.

5.2.4 Survey

Similar to the survey for Pebbles projects §5.1.4, I surveyed the Pintos user study participants for their opinions. Because of the different nature of the user study, of course, the questions here focus more on the debugging experience than on using Landslide directly.

1. How many Landslide bug reports did you receive from course staff? (Please indicate a number.)
2. Among those bug reports, how many were you able to diagnose and recognize the root cause in your code? (You may answer "all", "none", or a number.)
3. Among those bug reports, how many described a behaviour that you believe was NOT a bug? For example, Landslide printed an execution trace that was actually impossible, or Landslide reported a bug about some behaviour that was actually allowed by the Pintos specification. (You may answer "all", "none", or a number.)
4. About how much time did you spend interpreting Landslide's debugging output? (Please indicate a number of minutes, or a range if uncertain, e.g. "30-60 minutes".)
5. I found Landslide's debugging output easy to understand. (Multiple choice from strongly disagree to strongly agree.)
6. It's easier to diagnose the root cause of a bug with Landslide than with a stress test (for example `exec-multiple`). (Multiple choice from strongly disagree to strongly agree; plus "Not sure" and "Easier for some bugs but harder for others")
7. I feel that by interpreting Landslide's debugging output I learned to understand concurrency better. (Multiple choice from strongly disagree to strongly agree.)
8. These kinds of concurrency bugs are important to fix, even though they don't count against my grade. (Multiple choice from strongly disagree to strongly agree.)
9. Suppose after you submitted your pintos, we gave you 100 CPU-hours on the cloud provider of your choice to test it. Then we extended the project deadline by a day for you to use the results to fix bugs and get partial credit. How would you divide that CPU time between the staff-provided stress tests and Landslide? (Multiple choice: 0/10/.../100 CPU-hours on Landslide, 100/90/.../0 CPU-hours on stress tests.)
10. If course staff were to allow students to resubmit updated code after reviewing Landslide bug reports to receive partial credit for each bug that had been fixed, it

would be worth my time to try that (even if I could be spending that time working on the next project instead). (Multiple choice from strongly disagree to strongly agree.)

11. If a friend of mine took OS next semester, I would recommend that they should sign up to receive Landslide bug reports during projects in the future. (Multiple choice from strongly disagree to strongly agree.)
12. Regarding the previous question, why or why not?
13. Your name.
14. Your project partner's name (if applicable)

5.3 Evaluation

I pose the following evaluation questions.

- How many bugs, and of what severities, does Landslide help students find and fix before submitting their code?
- Does Landslide use result in higher quality submissions, whether directly for P2, or for subsequent projects as well?
- Do students feel the experience is worthwhile, as compared to stress testing?
- How well does Landslide apply to operating systems projects outside of CMU?

The data set is comprised of Landslide's automatically-generated usage snapshots from CMU students from semesters Spring 2015 to Spring 2018 inclusive, CMU's official project grades from same, CMU students' survey responses and submitted project code from Fall 2017 and Spring 2018 PSU students' survey responses and submitted project code from Spring 2018, and U. Chicago students' survey responses from Fall 2017. Table 5.1 shows the student participation rate across semesters for the thread library study[3].

5.3.1 Bug-finding

Firstly, I sought to prove Landslide does as advertised: finds concurrency bugs of severity, subtlety, and difficulty consistent with the lessons an advanced operating systems class should hope to teach its students, and provides diagnostic output that helps students understand and solve them.

At CMU, I configured Landslide to save a usage snapshot every time a student ran Landslide, including command-line options issued, the current version of their project code, a log of Landslide's output, and the preemption traces for any bugs found. These I analyzed by hand to determine how many distinct bugs were reported (as multiple preemption traces may refer to the same bug; inspecting their code if necessary), how many

[3] Participation at CMU was determined by receiving any automatic usage snapshots; participation at PSU was determined, for lack of snapshots, by sign-up form submissions, which might over-count slightly.

Semester	Used Landslide? yes	no	Total
CMU S'15	8	18	26
CMU F'15	22	8	30
CMU S'16	18	16	34
CMU F'16	12	6	18
CMU S'17	19	13	32
CMU F'17	14	5	19
CMU S'18	10	8	18
U. Chicago F'17	4	17	21
PSU S'18	38	98	136
CMU total	103	74	177
Non-CMU total	42	115	157
Total	145	189	327

Table 5.1: Participation rates across semesters, among students who submitted thread libraries (CMU, PSU) or kernels (U. Chicago).

were deterministic (i.e., Landslide reported them on the very first interleaving tested, with no need for artificial preemptions) how many were concurrency bugs, whether any were false positives, and whether the student was able to fix them thereafter (determined if a subsequent snapshot showed them re-running the test and verifying the bug's absence; an approximate measure at best, but far easier to implement than checking all the bugs by hand).

At U. Chicago, course staff ran Landslide on student submissions behind-the-scenes after each project deadline had passed, and returned its preemption traces to the study volunteers. I collaborated with the course staff to confirm whether each trace represented a real bug and not a false positive before distributing them to the students. On account of the small sample size (4 volunteering groups), we decided on this approach, rather than to study how students would cope with false positives themselves, to optimize for student happiness over scientific rigor.

As PSU's version of the study was planned on relatively short notice, and without immediate access to a shared, yet confidential, academic file system (such as CMU's AFS), I was unable to examine its students' objective usage data or bug reports for this section. Results from PSU are presented instead in §5.3.3.

CMU

In the recruiting lectures for every semester after the first, I included tallies of Landslide's bug-finding achievements to date as an extra advertisement to entice students to participate. These included each of the measures I counted by hand as described above, as well as aggregate totals of how many groups participated, how many groups found bugs, how many found at least one concurrency bug, and how many were able to fix and sub-

	S'15	F'15	S'16	F'16	S'17	F'17	S'18	Total
Participating groups	8	22	18	12	19	14	10	103
Groups w/any bugs found	5	21	12	7	14	10	6	75
Groups w/any bugs fixed	4	18	9	6	9	7	5	58
Groups w/all ($\neq$0) bugs fixed	2	11	3	3	7	5	4	35
Known false positives	1	2	6	5	3	1	0	18
Deteterministic bugs found ($\leq$)	5	20	18	18	30	15	9	115
Concurrency bugs found	5	56	19	15	13	9	5	122
Total bugs found	10	76	37	33	43	24	14	237
Deteterministic bugs fixed ($\geq$/$\leq$)	3	19	9	15	22	12	7	87
Concurrency bugs fixed ($\geq$)	1	38	11	10	12	4	4	80
Total bugs fixed	4	57	20	25	34	16	11	167

Table 5.2: Landslide's bug-finding performance across all semesters of the CMU 15-410 study. $\leq$ marks possible overcounts on account of unidentified false positives; $\geq$ marks possible undercounts on account of students not necessarily re-running to verify bugfixes.

sequently verify any or all thereof. Among the concurrency bugs, a wide variety of types were found: deadlocks, use-after-frees, segfaults, infinite loops, assertion failure, and unit test failure. Table 5.2 shows these statistics for each semester. Note that the tallies of bugs fixed may be undercounting on account of the possibility some students may have truly fixed them but skipped the verification step thereafter.

This table also shows the number of false positives, as self-reported by Landslide as described in §5.1.2. Please note that this approach is limited by Landslide's ability to classify them as "suspicious", even if not definitely bugs. These represent technical obstacles that either prevented Landslide from being able to analyze the synchronization in play (e.g., recursive `mutex_lock()` invocations) or were deemed too pedagogically important to allow the student to proceed without fixing (e.g., busy spin-wait synchronization loops)[4]. Other false positives arising from bugs in Landslide itself required more individual effort to confirm; to classify these I relied on students to help report them during the Landslide clinics and survey, and I report on them in §5.3.3. Nevertheless, apart from the spin-wait synchronization loops (which can arise nondeterministically due to lock contention, but which Landslide already self-identifies), I am aware of no cases of false positive *nondeterministic* bugs; all unexpected false positives encountered to date were output on the first interleaving tested. Though unscientific, this provides some assurance that Table 5.2's count of concurrency bugs is accurate, even if the deterministic tally may overcount.

Overall, Landslide helped students find and fix a lot of bugs. Among the 103 participating groups across all semesters, roughly three-quarters received bug reports, slightly more than half were able to verify their fix for at least one, and one-third overall were able to fix and verify *all* such bugs. Even avoiding the deterministic bug series for their possible overcounting, it's fair to conclude that Landslide caused at least two-thirds of all

[4] Landslide was also configured to issue a "bug report" if `MAGIC_BREAK`, a Simics debugging trap, was ever invoked, with specific instructions to remove it; I consider these closer in spirit to compilation errors than bug reports and so did not count them even among false positives.

concurrency bugs it found to get fixed before project handin. Although lacking a control experiment to make a more scientific comparison of these tallies, I tentatively conclude the students were well rewarded for their time. The questions of whether it was a *better* use of time than conventional stress testing, or whether the bug-finding resulted in submissions that course staff also, independently, judged better than before, are beyond the scope of just tallying bug reports and will instead be addressed in the upcoming sections.

University of Chicago

Among the 4 groups volunteering to receive Landslide bug reports, Landslide found 7 distinct bugs across the two projects tested, which I confirmed by manual inspection of the students' code. 3 among these were deterministic; 4 were concurrency bugs.

1. `priority-donate-multiple` found 2 deterministic bugs and 1 concurrency bug in `threads` projects. It found the targeted priority-leak bug described in §5.2.3 in one group, and exposed a `NULL` pointer dereference in another when attempting to donate priority to an already-exited thread. The latter bug was observed deterministically. A third group neglected to implement priority donation during `lock_release()` at all, which was found deterministically.
2. `wait-simple` found 1 deterministic bug and 3 concurrency bugs in `userprog` projects. It found a deadlock in two groups' implementations, wherein `process_wait()`'s synchronization assumed it would always be called before `process_exit()`. In both cases the former's `cond_wait()` call would block forever if reordered after the latter. This test also found several use-after-frees for one of those two groups, whose `process_exit()` didn't guard against a parent process exiting and deallocating its memory first; this root cause manifested in heap errors in 3 separate locations. Lastly, the heap checking alone found a deterministic use-after-free in a third group's `exec()`.

In cases where Landslide emitted multiple preemption traces for the same bug, whether manifested the same way through different combinations of preemption points or with different stack traces entirely arising from the same underlying cause, course staff sent them all to the student, including a disclaimer along the lines of "some of these may indicate the same bug". The other 3 tests, `priority-sema`, `alarm-simultaneous`, and `wait-twice`, found no bugs among the 4 groups (`wait-twice` being run only on the group that passed `wait-simple`). One of the 4 groups had no bugs found among any of the 5 tests.

Finally, one false positive was found while testing `wait-simple`. Landslide mistakenly reported that `free()` had reentered `malloc()` due to a technical discrepancy between when Pebbles and Pintos kernels take and release their heap allocation lock with regard to the rest of the `malloc()` API. This typically indicates a heap corruption bug, but in this case, `malloc()` was preempted after it released the heap lock, which was perfectly safe. After discarding the bug report and suppressing this false positive, Landslide moved on to find the true deterministic use-after-free described above.

Overall, I consider all of the 7 bugs to be "severe": they are all either correctness violations (priority mis-donation) or stability issues (crash, deadlock, or data corruption).

In the upcoming survey section, although only one of these groups did the survey, they enthusiastically reported finding the bug reports very helpful (§5.3.3). Despite the small sample size, I consider this a positive result that Landslide can be a useful concurrency programming aid even at universities with different class projects than CMU's 15-410. As for 1 false positive among 8 total reports, it is difficult to decide what rate of mistakes is acceptable when the user's patience is at stake [41], but the upcoming survey results (§5.3.3) suggest that students are generally at least willing to ask for help and make progress when technical support is available.

5.3.2 Student submission quality

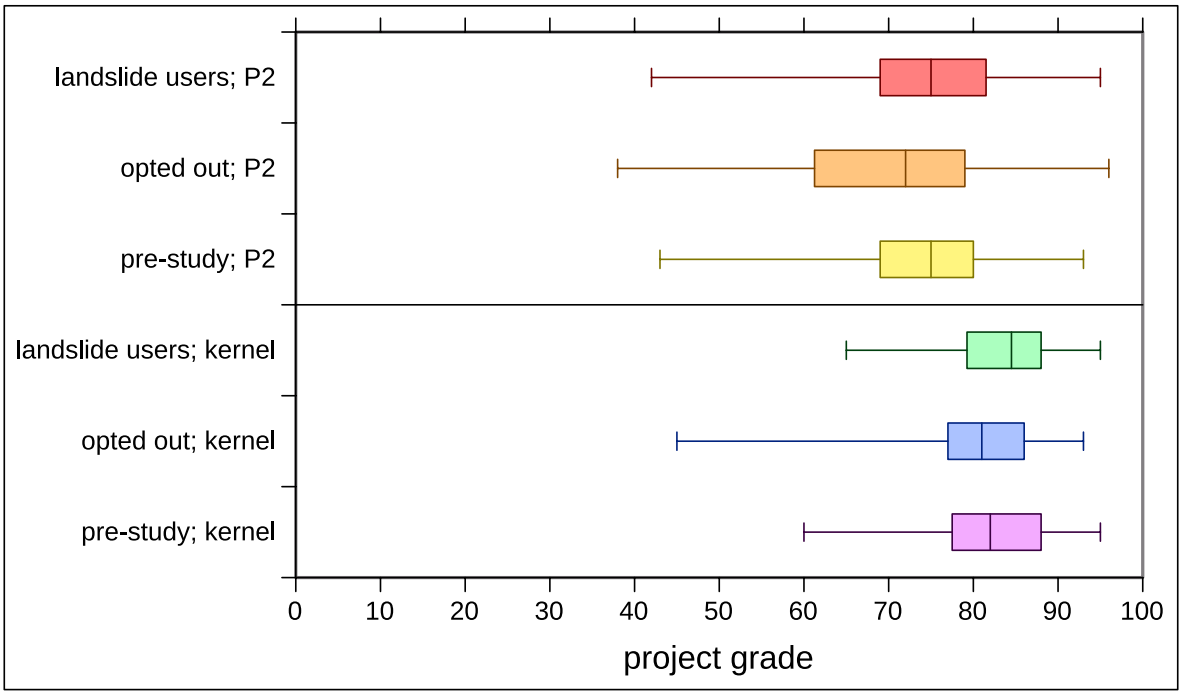
I evaluated Landslide's impact on student submission quality in two ways. First, I analyzed CMU 15-410 students' overall project grades between Landslide users and non-users to see if Landslide helped them submit overall better implementations. Second, I studied several individual concurrency bugs that I thought Landslide was likely to detect, all already well-known by 15-410 course staff, to see if using Landslide correlated with submitting projects absent those bugs.

Impact on grades

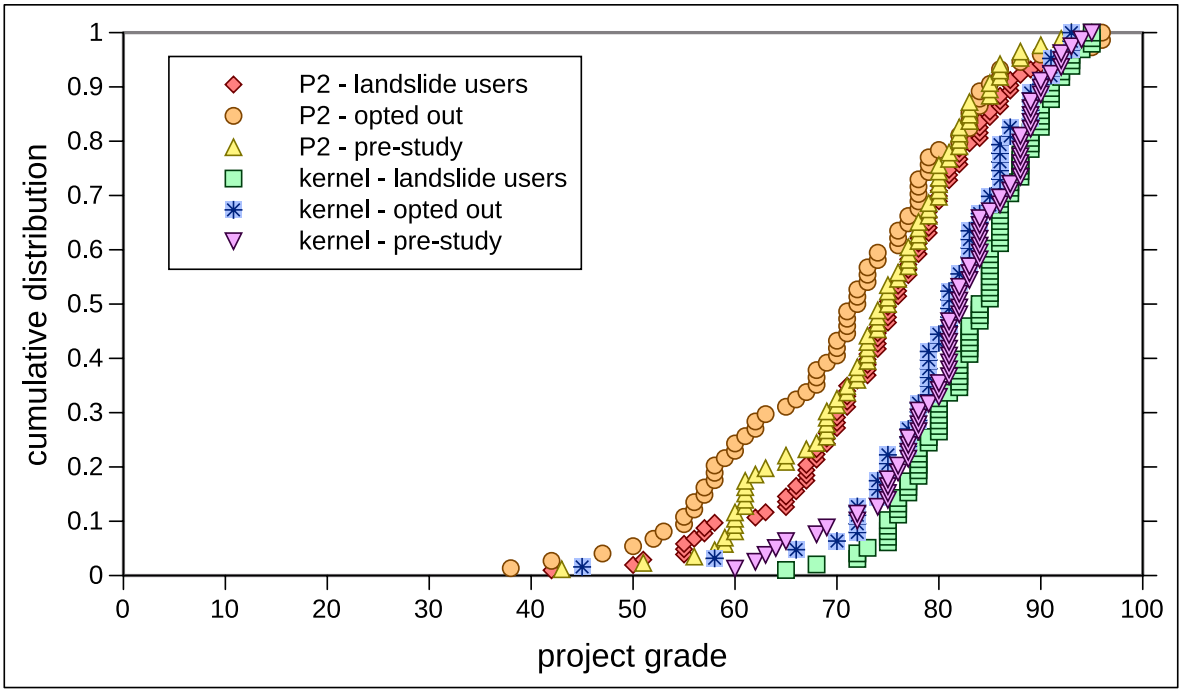
Figure 5.1 shows the distribution of project grades in CMU 15-410 between Fall 2013 and Spring 2018, grouped by whether or not students used Landslide during P2. The control group is further distinguished by whether Landslide was offered and the student declined, or whether the study had not yet begun that semester. Intuition suggests students in the former group would be more likely to be struggling too much to have free time to volunteer in the first place, so any comparison between them and Landslide users is vulnerable to selection bias. The latter control group, in which students did not have a choice, mitigates such bias, although itself may be vulnerable to other confounding factors such as grading criteria varying across semesters. In addition to P2, I also show the subsequent P3 (kernel project) grades, likewise broken down by who used Landslide previously during P2[5]. Should Landslide use be correlated with higher grades on the later, more difficult concurrency project, one explanation could be that Landslide helped students internalize new concurrency programming and/or debugging skills that would help them write better kernels even without Landslide's aid.

Results. Overall, Landslide users did indeed receive slightly better grades overall on their P2 submissions than non-users from the same semesters, although comparing to the second control group lessens that difference somewhat. The difference is also less pronounced in both comparisons among P3 grades (unsurprisingly, as learning lasting lessons about concurrency should be harder than fixing bugs case-by-case), although the experimental group still maintains a tiny lead.

[5]The P3 distributions are slightly smaller than those from P2, as some students dropped the class in between. Some new project groups also tend to appear during P3 as students either solo or find new partners, but as these cannot be meaningfully classified as Landslide users or non-users, I omit them here.



(a) Visualized as a box plot.



(b) Visualized as an EDF.

Figure 5.1: Distribution of project grades among Landslide users and non-users. Study participants (S'15 to S'18) are the experimental group; non-participants (S'15 to S'18) and students from semesters predating the study (F'13-F14) are the control groups. Compare Landslide users to non-users for within-semester differences, or users to pre-study students to mitigate selection bias.

Population			Distribution				Significance	
project	**group**	N	**min**	**mean**	**max**	**stddev**	p **vs users**	**cutoff**
	users	103	42	74.8	95	10.3	–	–
P2	non-users	74	38	70.9	96	12.5	0.036	0.05
	pre-study	86	43	73.7	93	9.7	0.671	0.05
	users	98	65	83.6	95	6.2	–	–
P3	non-users	63	45	80.7	93	8.2	0.034	0.05
	pre-study	79	60	81.6	95	7.6	0.145	0.05

Figure 5.2: Detailed statistics from student grade distributions.

Statistical significance. Table 5.2 presents more detailed statistics from these six distributions. To assess whether the differences are significant, I use the k-sample discrete Anderson-Darling test [121] provided by the R statistical programming environment [117, 122], comparing each control group to its respective Landslide users group. Anderson-Darling uses a weighted sum-of-squares distance metric to compare two empirical distribution functions (EDFs), testing if their samples are likely to arise from the same underlying, though unspecified, distribution[6]. I deem the difference in grade distributions significant when $p < 0.05$[7], although as in all p-value calculations, this rejects only the null hypothesis, not alternative hypotheses, such as "more skilled students were more likely to sign up to begin with". Ultimately, only the same-semester comparisons were significant. However, the distribution difference being smaller in P3 than in P2 further suggests, albeit informally, that the larger impact in P2 is not attributable *only* to selection bias, or else a similar difference should have been observed in the P3 distributions. Another possible explanation could simply be that the P3 grading critera result in less grade variance overall, but this pattern is at least consistent with the "Landslide helps students submit better P2s" hypothesis.

Common bugs

Based on the results from §5.3.1, I selected 4 bugs to study in depth to ascertain whether Landslide played an instrumental difference in helping the students ultimately submit respectively correct implementations. To avoid bias of picking too obscure and/or trivial bugs that Landslide alone might find but even course staff would not expect students to solve, I chose only bugs which had substantial penalties in the grading rubric (guided, as well, by my own intuition as a former TA). While checking for a given bug's presence by manual inspection, I blinded myself to whether each group had been a Landslide user or not. After unblinding, I then re-classified students who used Landslide in general, but whose usage snapshots showed they did not run the test case in question, as non-users.

[6] I choose Anderson-Darling over the simpler Kolmogorov-Smirnov, which computes only the maximum instantaneous distance between EDFs, for its better sensitivity both to repeated deviations and to tail differences [43].

[7] I choose not to correct for multiplicity among these 4 comparsons like I do in the next section because each tests a different hypothesis.

Bug name	Landslide users #correct	Landslide users #buggy	Non-users #correct	Non-users #buggy	Δ%correct	Significance p	Significance cutoff
Exit UAF	11	10	0	16	+52.4%	0.00061	0.0125
Mutex	17	1	19	0	−05.6%	0.49	0.0125
Paraguay	20	1	14	2	+07.7%	0.57	0.0125
Downgrade	10	3	19	4	−05.7%	0.69	0.0125
Total	58	15	52	22	+09.2%	0.26	0.05

Table 5.3: Correlation of student Landslide use with solving certain bugs in their final submission during Fall 2017 and Spring 2018 semesters. Note that the totals in the bottom row double-count students, which makes sense only if you believe the incidences of each bug in a given submission are independent.

Table 5.3 presents the results.

Each of the four bugs studied is described in detail as follows.

1. **Thread exit/join.** After a thread finishes exiting, the memory allocated for its stack must be reclaimed for other uses. A common student pitfall is to allow a `thr_join()`ing thread to free said stack space while `thr_exit()` is still executing userspace C code (which typically accesses the stack), or even for `thr_exit()` to free it itself. In the former case, threads must interleave specifically to exhibit a use-after-free; in the latter case, the use-after-free will be deterministic (i.e., present in all interleavings). Other than Landslide, however, the students have no Valgrind-like heap debugging tool which would report a bug immediately upon any illegal heap access. This means that a subsequent `thr_create()` invocation would need to race to recycle the memory for a new thread stack and conflict with the old exiting thread before any problem could be detected. The Landslide-friendly test `thr_exit_join` is most likely to expose this bug. Whether each student's submitted P2 solved or suffered from this bug was determined by me personally inspecting their code.
2. **Mutex.** The Landslide-friendly `mutex_test` checks for the possibility of two contending threads accessing a mutex-protected critical section simultaneously. It includes one thread repeating once the lock, unlock cycle so as to check for unlock/lock races as well as lock/lock interactions, and, as mentioned previously (§5.1.3), checks for data race access pairs inside the mutex implementation itself. As students have free rein to design their mutex internals, this refers to the general class of mutex bugs in which any number of things can go wrong, depending on the implementation, leading to mutual exclusion (or otherwise assertion) failure. Whether each student's submitted P2 solved or suffered from such bugs was determined by checking their grade files for any mutex-related penalties (assessed by the TAs), then me double-checking their implementations by hand to confirm.

   Further investigating the one group who submitted a buggy mutex, I found that while they had run `mutex_test` in Landslide (and even found and fixed a separate deterministic bug already), Landslide found no bug in what was presumably a correct implementation, then they updated their code, introducing the bug, without

testing it again thereafter.

3. **Paraguay.** Named after Ivan Jager, the Paraguayan 15-410 TA from 2004-2006 who originally discovered it, this refers to a condition-variable bug in which a thread which sequentially sleeps on two different condition variables can spuriously wake up early from the second `cond_wait()`. The precise reasoning why many naïve student implementations are susceptible to this bug, as well as the 3 major ways of fixing it, are one of 15-410 staff's closely-guarded secrets (§2.4.1). Suffice it to say that the `paraguay` test invokes a custom Pathos `misbehave` mode which biases thread scheduling towards the interleaving required to exhibit the bug (Landslide, of course, replaces this `misbehaviour` with model checking). Hence, despite being a subtle concurrency bug, the official course test suite is likely to expose it, so comparing how many Landslide users and non-users submitted this bug in particular would speak more to the impact of Landslide's preemption traces in helping to diagnose a bug that a "stress" test could already find. `paraguay` is itself the Landslide-friendly test for its eponymous bug. Whether each student's submitted P2 solved or suffered from this bug was determined by me personally inspecting their code.
4. **R/W-lock downgrade.** In addition to the standard R/W-lock interface, P2 requires students to implement `rwlock_downgrade()`: called with the lock held in write mode, the caller adopts the reader role instead, allowing other waiting reader threads to proceed simultaneously, all while allowing no waiting writers to access in between. The Landslide-friendly test `rwlock_downgrade_read_test` checks that readers are allowed simultaneous access after a downgrade with no possibility for deadlock. Rather than one specific bug, this refers to any of several failures that can arise during a downgrade. Whether each student's submitted P2 solved or suffered from these bugs was determined by checking their grade files for any downgrade-related penalties (which were assessed by the TAs as a result of their manual inspection, rather than mine).

Statistical significance. The p values in Table 5.3 are calculated in R [117] using Fisher's exact test [44], treating each row as an independent 2x2 contingency table. I divide 0.05, the standard significance cutoff, by the number of bugs to conservatively account for multiplicity [100] (not that it matters with these p values).

The latter three bugs' occurences are thoroughly uncorrelated with Landslide use, the `thr_exit()` bug standing alone with extremely high significance: not a single student who did not use Landslide these semesters submitted a correct implementation. This difference is easily explainable: the class-provided unit tests already do a good enough job catching the other three bugs that students are able to find and fix them before submission regardless of what testing tool they used. (The high correct-to-buggy ratio corroborates this.) Nevertheless, that does not mean Landslide is pointless for these bugs; it may well have helped the students reproduce them more reliably and/or diagnose them faster. This is merely a null result for submission quality, not necessarily a negative one, and the next section will attempt to evaluate such quality-of-life improvements instead.

On the other hand, because the `thr_exit()` bug typically manifests as a use-after-free, the official tests must stress the thread library until the memory in question gets

re-allocated in just the right interleaving pattern to cause corruption that leads to a visible crash. Landslide, meanwhile, detects this error immediately upon access (§3.3.3), with no need for complex corruption conditions. Among the 11 Landslide users who submitted correct `thr_exit()`s in this regard, it is difficult to say whether the heap checking alone, together with unit or stress tests, would have been sufficient, or whether heap checking and model checking were both necessary in concert to help them. To isolate the impact of model checking, I simulated the students having access to a stand-alone heap checker by checking only the first thread interleaving with Landslide (similarly to §4.3) and assuming the students would find and fix any such "deterministic" use-after-free bugs before the deadline. Re-classifying these into the "correct" group, the 11-10-0-16 distribution becomes 11-10-3-13, with a new p value of 0.048: still positively correlated, but no longer significant under the multiplicity-corrected cutoff.

5.3.3 Survey responses

Analyzing only the raw technical data of how users interacted with Landslide can paint only part of the picture. For one, offering students better testing and debugging tools may not necessarily find strictly more bugs or help students submit more correct implementations than with stress testing; it may instead find the same bugs faster, affording the students more free time apart from the project, a quality-of-life improvement normally invisible to graders. For two, Landslide's automatic snapshots, being captured at the time of issuing each preemption trace, necessarily miss the student's subsequent experience interpreting them. The surveys listed in sections §5.1.4 and §5.2.4 serve to probe these more qualitative aspects of the Landslide experience.

Response data

The response distributions for each of the surveys' multiple choice questions are shown in Figure 5.3. In total, 28 students (or pairs thereof) answered the survey: 12 pairs from CMU, 15 individuals from PSU, and 1 pair from U. Chicago. The first four questions/graphs focus on concrete debugging results, and the latter six on the students' subjective opinions. Note that two of the questions (3 and 7 from §5.1.4) were not asked on U. Chicago's version of the survey, so their corresponding graphs show only CMU and PSU response data. Likewise, U. Chicago's questions 4, 8, and 10 (see §5.2.4), which were not asked on CMU's and PSU's surveys, having only one respondent, are not pictured; the answers thereto were "15-20 minutes", "Agree", and "Agree", respectively. Also, this respondent's answer of 6 on the "How many bugs" question appears to indicate the total number of preemption trace files course staff sent them; upon further investigation, the 6 traces seem to represent 2 distinct bugs among them (§5.3.1).

The survey responses were very positive: students reported being able both to diagnose and to verify as fixed the vast majority of Landslide's reported bugs, compared Landslide favorably to stress testing, and found the experience worthwhile and worth recommending. Regarding the 100 CPU-hours question in particular, students could approach answering it in several different ways: do the three students who answered 20/80

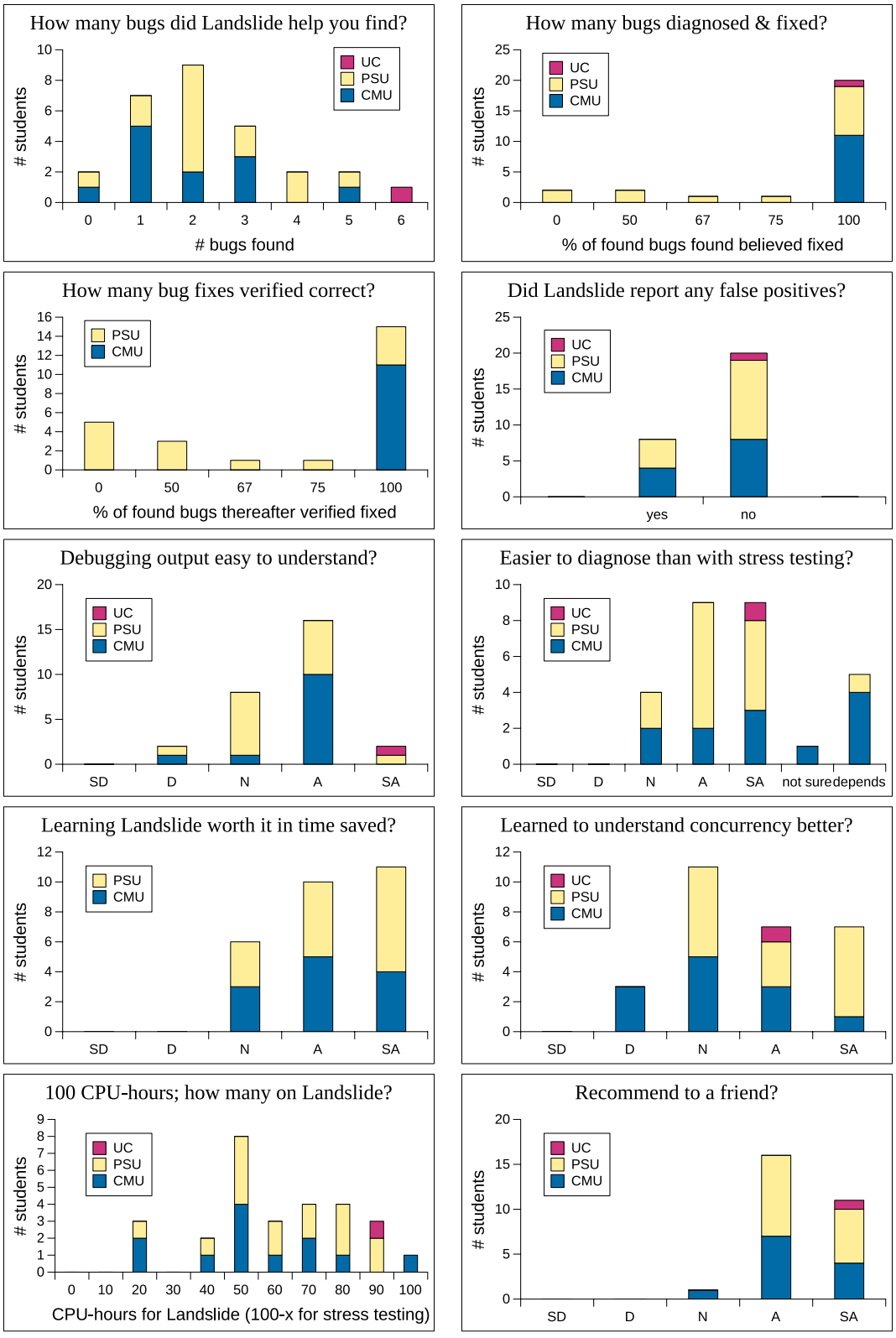


Figure 5.3: Student survey responses. SD/D/N/A/SA stands for strongly disagree/ disagree/neutral/agree/strongly agree.

think stress testing is four times as good as Landslide at finding bugs, or that stress testing requires four times as long to reach the same point of diminishing returns? On the other extreme, one student said they would spend all 100 CPU-hours on Landslide, which probably speaks more to their enthusiasm than to a careful attempt to maximize expected number of bugs found (even the author themself recognizes stress testing's advantages for certain types of bugs, such as resource exhaustion). Nevertheless, the responses' overall bias to spending at least half the CPU-time on Landslide shows clearly that the students found the experience worthwhile.

Three questions with open-form answers bear further discussion: what kind of false positives Landslide reported, reasons they found it worth recommending to a friend, and suggestions for improving the interface (PSU only[8]).

False positives

Even though 71% of students reported receiving no false positive bug reports, the nature of Landslide's bug-detection algorithms is such that it should ideally never report any correct behaviour as wrong, so I consider those 29% that did report such the most negative result among the survey responses. They described their false positives, and I either make excuses or own up, as appropriate, as follows. Reports from CMU students (Simics version):

1. Landslide complained of a nonexistent `MAGIC_BREAK` (Simics debugging function), despite the student's code never invoking it. This arose because of technical confusion between the test program's and the shell's address spaces, and was subsequently fixed in commit 3f24d67 (Simics repository only).
2. Landslide reported "some weird errors" and/or mysteriously crashed when multiple instances were run from the same directory. (Multiple students suffering this failure mode contacted me for support, though only one reported it on the survey; in some cases, I recall said weird errors manifesting as false positive invalid heap access reports.) Simultaneous Landslides can clobber certain auto-generated header and/or temporary files, scrambling its instrumentation and leading to chaotic behaviour. I introduced a guard against this in commit 977e8fb (Simics repository) and e49b5df (Bochs repository).
3. One student reported "Data races which I believe they are not". Looking at their usage snapshots to corroborate this, these appear to be true, yet benign, data races (in several cases corresponding to `mutex_test`'s verification of the mutex's internal memory accesses (see §5.1.3)); i.e., expected behaviour rather than false positive bug reports. Student confusion about these could be alleviated by improving Landslide's user interface messages when printing data race information.
4. One student complained of a bug report that, while truly a bug, showed wrong filenames and line numbers in stack traces, and (quite naturally) suggested that accurate stack traces would make debugging easier. Checking against their usage

[8] I thought to ask this question too late for CMU's surveys.

snapshots, this seems to be an issue with Simics's symtable's handling of assembly functions. The Bochs version handles these correctly.

Reports from PSU students (Bochs version):

5. Landslide kept crashing for one student while running `paraguay`. This is likely due to exceeding Linux's process limit and/or exhausting the class's official VM's memory, which can arise when many data race candidates cause Quicksand to spawn many Landslide jobs, and (as on `paraguay`) each with very large state spaces that must defer. I had been working on mitigating this problem just before beginning the PSU study; properly addressing it would involve improving Quicksand's memory-exhaustion detection code and/or making Quicksand at all aware of the process limit to begin with. (Note that this is not strictly a false positive bug report, just a Landslide crash.)
6. Students who initialized child threads with a base pointer value of `0xffffffff` observed Landslide crashes, as it attempted to stack trace through that address and access memory that wrapped around the address space. (Several students contacted me about this via email, and I issued a prompt fix; one student later reported it on the survey.) Commits 0573e34 (Bochs repository) and 654f459 (Simics repository) fixed this bug. (Like the above, not actually a false positive bug report.)
7. Landslide issued false invalid heap access reports to one student who had been using `new_pages` to allocate thread stacks "very close" to the `malloc` heap. They reported this during the study and I fixed it promptly for them in commit 4a26da7 (Bochs repository only), then reminded me of it again in the survey.
8. Finally, one student reported simply, "Race condition". Without the same usage snapshots to consult as I'd have for a CMU student, or more self-reported detail, I regrettably can offer no comment.

Overall, most of the issues reported as answers to the "false positive" survey question were merely Landslide crashes or user interface confusion. Those few truly erroneous bug reports (items 1, 2, and 7), while certainly guilty of burdening students with worry over whether the problem is their own code or in Landslide itself, are at least not too discouraging for two reasons. Firstly, in all cases the students were able to recognize the false positive quickly enough to ask for help, and I was able to deploy a fix and let them proceed before the project came due. Secondly, each such problem, now having been fixed, will befall no future student again – not to assert Landslide is completely bug-free now, but at least that it grows more and more stable with each passing semester.

Reasons worthwhile

After the question asking "Would you recommend a friend taking OS next semester to use Landslide?", I asked the students for open-form reasons why or why not. As the former question's answers were exclusively positive (only 1 student even answering "no opinion"), this question's answers turned out to be mostly praise. 3 students declined

to answer this question; I reproduce the rest here, paraphrasing for clarity and brevity[9]. Answers from CMU:

1. Easy to use and found nontrivial bugs.
2. Useful to find some bugs, but other types of bugs (such as memory corruption) were impossible to find with Landslide.
3. Helped verify our atomic primitives were correct.
4. Would recommend, but takes too much time (unclear, but seems to be referring to execution time rather than setup/usage).
5. Tests are automated and can be left running for a long time. However, faced uptime issues with CMU's linux servers (which reboot every night); wished for a more reliable execution environment.
6. It's pretty helpful.
7. Seems more reliable than stress tests.
8. Found a bug not found by stress tests.
9. Finds concurrency bugs with little effort that may be undiscovered otherwise. (Also provided some interface feedback here, since I didn't ask CMU students a separate question for such; see next section.)
10. Easy to learn and a simple way to test our P2.

Answers from PSU:

11. Although didn't find any concurrency bugs for me, gives me more confidence about my code.
12. Helpful, just didn't have a lot of time to use it.
13. Does not make concurrency debugging easy, but definitely makes it easier.
14. Helpful to find bugs you weren't previously aware of. Makes more sense to use an expressly designed tool rather than (unit/stress) tests.
15. Helpful and easy to use.
16. Saves time overall, can run long tests overnight.
17. Helps to find concurrency bugs and their root causes better than stress tests.
18. It finds the concurrency bugs you need to fix for full credit.
19. Helpful for finding some uncommon bugs I hadn't found or wasn't looking for.
20. Found bugs I kept overlooking, which may have taken many hours to find otherwise.
21. Helpful for the difficult step from code being "finished" (scare-quotes theirs; presumably meaning "feature-complete") to getting rid of all concurrency bugs.
22. Easy to use, setup taking no more than 5-10 minutes, and allowing being run overnight.

[9]Note that most students used the term "race condition" rather than "concurrency bug", as taught in CMU and PSU lecture material; I replaced these while paraphrasing in accordance with §2.5.

23. Very efficient at concurrency testing. Stress test crash reports do not necessarily point to the root cause, due to memory corruption for example; plus bugs may not show up every time due to nondeterminism.
24. Helped find a bug I wouldn't have found otherwise. Did not show an interleaving directly (i.e., did not issue a bug report), but reported a data race that turned out to be a concurrency bug upon inspection.

Answer from U. Chicago:

25. Found several subtle, legitimate bugs we wouldn't have easily caught otherwise, but made sense once revealed. Fixing them took little time but allowed us to proceed confidently on the next project. Often wished for Landslide to have been available to use during the next project as well.

Overall, students most commonly praised Landslide's ease of use, its ability to find bugs that elude stress testing, and the confidence instilled from verifying bugs had been fixed.

Interface suggestions

Lastly, I asked the PSU students for any feedback they might have on making Landslide's interface easier to use or understand.

1. Requested for preemption traces to be more clear about the meaning of each stack trace in each table cell, and complained of inaccurate line numbers (likely referring to how the current behaviour indicates the line of code *after* a function call, corresponding to the `call` instruction's pushed return address, rather than the function call itself). (This answer from CMU; see above.)
2. Including a manual or tutorial would be helpful (presumably beyond the user guide's instructions, such as recapping the procedure shown in the lecture demo which wasn't written down anywhere).
3. Don't print warnings about line length exceeding 80 characters (inconsistency between 15-410 and CMPSC 473 compilation options).
4. "It takes too long. But I guess that's impossible to fix." (Well, it's an open research problem to fix!)
5. Preemption traces should explicitly indicate where in the interleaving the bug occurred. (Root cause identification is its own research area, but more detail is certainly possible.)
6. Improve explanation of data races (in the user guide, perhaps).
7. Happy with it as-is (3 students)
8. No response (7 students)

Though I did not ask this question on the CMU survey, my experience handling student questions in person suggests CMU students also mostly wish for better explanations of data races and more detail and clarity in the preemption traces. Though I present the formal definition of data races in the lecture (§5.1.1) and refer back to it in Landslide's documentation and output, showing a concrete example in future iterations of the user

guide would go a long way to illustrate the abstract concepts. The preemption traces could be improved by making it clearer that the stack trace in each cell of the table represents executing the thread in question from wherever it previously left off (or its inception) all the way until it reaches that stack trace, then preempting it to run another thread. They could also easily report more diagnostic information; for example, showing the sets of memory conflicts between each thread, annotating the type of each preemption point (yield, mutex, or data race), and/or indicating the adversarial memory access for each data race preemption point.

Other universities

Comparing the survey response distributions between CMU versus PSU and U. Chicago (Figure 5.3), CMU students tended to verify their bugfixes more often by re-running Landslide and found the preemption traces easier to understand, while PSU and U. Chicago students generally compared Landslide more favourably to stress testing (while CMU students comparatively preferred the more nuanced answer that it depends on the type of bug), and reported more often that helped them understand concurrency better. Considering the higher demand CMU's 15-410 makes for prerequisite concurrency experience compared to PSU's relative tempering of P2's difficulty to make it more accessible (§2.4.1), these trends seem to correlate with the different levels of preparation each course's students had, showing that students of various skill levels can each benefit from the experience in different ways.

5.4 Discussion

Human subjects research is inherently messy. Each of the previous section's approaches to evaluating Landslide's educational value was accompanied by some drawback which prevented it from being perfectly objective science, but many of them presented tentatively positive results nonetheless. Landslide helped many students find and fix many bugs (§5.3.1), but making a direct comparison to stress testing, the prior state of the art, is not straightforward. Immediate improvement in students' project grades was observed (§5.3.2), although statistical significance was lost when attempting to account for selection bias; and impact on grades alone is a very narrow measure of pedagogical value anyway. Landslide's debugging power was also found to be statistically significant for the `thr_exit_join` bug in particular. Students responded overwhelmingly positively in the survey (§5.3.3), although it is easy to imagine students being equally happy with a "debugging tool" that just tells them all the answers.

Nevertheless, I believe each of these partial results taken together paint an overall picture of success: students fixed their own bugs *and* were happy about it; students were able to ask for help rather than be deterred by inevitable technical difficulties as far as I know; students provided intelligent feedback suggesting they truly understood the debugging process; et cetera. The rest of this section will discuss the study's limitations and provide some perspective for the future.

5.4.1 Bias

As long as an educational user study is run on a volunteer basis, one cannot completely avoid selection bias: those with enough free time to participate are more likely to be the most capable students already, who are least in need of the extra debugging help. This was especially evident in the annotation-required P3 study from [12], in which only 5 groups volunteered (15% of the 34 total who submitted P3 that semester), among which 2 had enough after just the annotation phase and did not continue to do any in-depth testing. In contrast, since switching to the automatically-annotatable P2, the participation rate rose to 58% (Table 5.1) among all P2-submitting groups. Reaching over half the class could already be seen as a major step in mitigating said selection bias.

The survey may also be susceptible to several sources of bias beyond participation itself. I suspected students might feel inclined to be overly polite in their answers (whether consciously or subconsciously). I attempted to counteract this by concluding my survey link emails with, *Please try to answer honestly rather than flatteringly–if any part of the experience was bad for you, I want to hear about it to make Landslide better in the future!* It's also possible that survey respondents were more likely to be those who enjoyed Landslide the most, meaning I might not hear as much negative feedback as I should.

I took no special measures to compensate for bias in gender, race, or being non-natively English-speaking during recruitment or the survey. Surveying students to measure any differences in these demographics between study participants and the overall class population would have required mandatory survey participation, and in turn, a more rigorous IRB approval process. According only to my memory of students who attended the Landslide clinics (§5.1.1), the racial diversity was roughly representative of the class at large, and the proportion of women I perceived was in fact somewhat higher than the overall gender ratio. More scientific analysis of such statistics was deemed beyond the study's scope for now.

5.4.2 Retrospect

More than just trying to draw firm conclusions from the opinions of students who are just learning concurrency to begin with, student feedback in turn guided the constant development of Landslide, and the experimental design itself, as the semesters went by. In this section I will fantasize about how I might have run more perfect experiments granted the impossible wish of knowing then what I know now.

Pebbles

The survey was introduced into the study only in time for two semesters' worth of student responses, after several iterations of collecting only usage and bug report snapshots. Apart from the obvious improvement of having been surveying students from the beginning, the following questions could have improved the survey.

1. *Did you have any technical difficulties with Landslide that I had to intervene on, whether in person or over email?* (Some students reported this in the "false positives" ques-

tion, although fewer than I helped overall, so others must have not mentioned it.) Comparing answers to this question across subsequent semesters would give a sense of how much Landslide's stability was improving over time and whether it was mature enough for unsupervised use in the future.

2. *How well do you feel you understood the research challenges explained in the lecture?* and, *How well do you feel a user should need to understand same in order to benefit from Landslide's bug reports?* (Answers on a scale from "Not at all; Landslide is a magic black box to me" to "I'm ready to work on research in this field myself".) These questions would help fine-tune the lecture material and user guide to maximize student comfort, and potentially also corroborate the claim that Landslide is accessible even to novice users.
3. *What additional debugging information would you want displayed on the preemption traces?* Knowing now that interpreting preemption traces was a sticking point for many users, I would hope to identify the most wished-for features to know what to prioritize improving. This could also assess students' understanding of what kinds of information would or would not be reasonable for Landslide to record and report.
4. *For each bug Landslide found in your code, how trivial or severe do you feel it was?* This would help get a sense of how the students regarded Landslide on a spectrum between annoying style checker or a life-saver, and potentially suggest options to make Landslide suppress certain types of bug reports. For example, it currently reports spin-wait loops in `mutex_lock()` as bugs with a special message referring students to relevant lecture slides, but it's possible refusing to test any code beyond until that bug was fixed might have made Landslide less useful overall.

Some students (around 0 to 2 per semester if memory serves) emailed me during P3 to ask if they could test their kernels with Landslide just like their thread libraries. I answered each by explaining that it would take more effort on their part, and then if they were still interested, guided them through the annotations on a case-by-case basis. This process was not included in the IRB-approved study protocol, so I collected no results from them. If I had planned in advance, I could have supported this "bonus stage" officially, and further surveyed the brave volunteers about how P3 Landslide could be made more generally accessible.

Finally, to evaluate whether the experience of using Landslide left the students with any lasting lessons learned, a follow-up survey could have been given one or two years later. Such a survey would ask, for example, *Have you encountered any debugging problems since finishing OS that made you wish for a tool like Landslide?* and *Do you feel the way you think about testing, debugging, and program correctness has been influenced in any way by using Landslide?* to evaluate its lasting impact on their understanding of concurrency.

Pintos

While part of the point of this experimental design was to evaluate Landslide as a grading tool in the hands of TAs, I would be remiss to mention that I also feared the automatic annotation process would not be as robust as the P2 version. Indeed, while helping Kevin

get oriented with using Landslide, I implemented several fixes/improvements to the setup scripts as I found student kernels that failed to automatically annotate (for example, those with `ready_list` changed to an array, as described in §5.2.2). Had I given Landslide directly to students that semester, the students themselves would have had to email me for tech support.

I attribute the comparatively low participation rate of Pintos students to two major factors: one, not incentivizing the students to directly improve their grades (instead offering only the vague promise of a "learning experience" debugging their code only after handin), and two, not traveling to the university to introduce the research topic in an in-person lecture (leaving the students potentially confused about what advantage, if any, was offered over stress testing). Hypothetically, I could have achieved greater user study participation either by offering extra credit to students or by offering an autograder-like interface for students to receive bug reports before their deadlines instead of after (either way requiring a more rigorous IRB review process).

5.4.3 Future educational use

Now being done collecting student usage data to publish as research results, and no longer bound by the IRB's requirement that Landslide be isolated from the grading process lest it be seen as coercion to participate, Professor Eckhardt and I have discussed options to deploy it as an official part of 15-410's curriculum. This section has already clearly shown students are capable of debugging with it on their own time, and I believe it well-automated enough to supplement Fritz (the existing stress testing infrastructure) in the class's grading process as well. TAs could also, at their option, use Landslide by hand to confirm any bugs encountered during manual inspection and/or write new Landslide-friendly tests to expose bugs not yet targeted by the 6 tests offered here.

Over the study's seven semesters, I believe the stability of Landslide's instrumentation process has improved enough to require little to no ongoing technical support anymore, although Landslide-specific office hours may still prove helpful. I am willing to continue giving the guest lecture as long as proximity and curriculum allow, although I also hope the documentation herein be enough to pass the mantle like any other piece of the course infrastructure. Future problems to address include grading bias (i.e., students submitting blindly-hacked code that just barely passes Landslide, even if not necessarily correct, thereby gaming the autograder), and improving usability to reach even the most struggling students (i.e., that last 42% who submitted P2s without participating in the study).

Regarding non-research use in Pintos classes, Landslide can now handle a considerably wider variety of student implementation quirks on account of the fixes from this time (§5.2.2). In its original shape (before the F'17 semester, having only enough instrumentation necessary for the Pintoses used in §4.3's experiments), Landslide was already able to automatically instrument 18 out of the 21 `threads` project submissions at U. Chicago. I was able to quickly deploy a fix to make the annotation scripts handle the other 3 cases, although such technical support is not something any TA would be able to do. In its current shape I would recommend it for TA use grading, but not necessarily

directly to students without someone familiar with the codebase on immediate hand for tech support. However, I also believe Landslide's success in these user studies, provided me present to handle technical issues, serves as testament for stateless model checking in general in the educational theatre. While Pintos's kernel-level environment presents a unique challenge for concurrency testing, other, more readily automatic model checkers for user-space programs, such as dBug [127] or CHESS [102], could easily be used on other thread-library-like programming projects at any university.

Chapter 6

Transactions

Transactional memory (TM) is a concurrency primitive by which programmers may attempt an arbitrary sequence of shared memory accesses, which will either be committed (made visible to other processors/threads) atomically, such that no intermediate state modification is ever visible, or in the case of a conflict which would prevent such, discarded with an error code returned to allow the programmer to write a synchronized backup path. Transactional memory specifications typically have three API functions, abstractly speaking:

- xbegin begins a transaction, staging any subsequent shared memory accesses in some temporary thread-/processor-local storage, and checking for conflicts with the accesses of any other threads or CPUs. If the transaction is unsuccessful, as described below, begin instead returns an error code indicating the programmer should fall back to some other, possibly slower, synchronization method.
- xend ends a transaction, attempting to commit all staged accesses to the shared memory atomically with respect to reads or writes from other concurrently-executing code. If any of those accesses conflict (i.e., read/write or write/write) with any other access to the same memory since the transaction started, they are instead discarded and execution state reverts to the begin with an error code as described above.
- xabort explicitly aborts a transaction, regardless of any memory conflicts, discarding changes and reverting execution as described above. Some implementations allow an arbitrary abort code to be specified which will appear in begin's error code.

Implementations. Software TM implementations (STM) typically function as a library, tracking staged memory accesses in local memory, and aborting whenever a conflict is detected between two transactions' tracked accesses [4]. Hardware TM implementations (HTM) use processor-level hardware support, which stages changes in per-CPU cache lines, and may abort for STM's reason above [68], or additionally whenever a conflict is detected between one transaction's traced access and *any* other memory access, or whenever a conflict occurs on the same cache line, not necessarily the same address, or in case of any system interrupt or cache overflow. Intel's commercial implementation of HTM has a rocky history of hardware bugs [58, 67], attesting to the feature's complexity

and the need for formal verification on both sides of its API.

Terminology. The world of hardware transactional memory is home to several more confusing acronyms in addition to "HTM". Transactional Synchronization Extensions (TSX) refers to Intel's implementation of HTM on Haswell and more recent microarchitectures [59]. Restricted Transactional Memory (RTM) refers to the `xbegin`, `xend`, and `xabort` subset of TSX instructions, which of course correspond to `begin`, `end`, and `abort` listed above, as well as `xtest`, an instruction which returns whether or not the CPU is executing transactionally. GCC and Clang expose these as C/C++ intrinsics named `_xbegin()`, `_xend()`, `_xabort()`, and `_xtest()` [47]. Hardware Lock Elision (HLE) refers to the `xacquire` and `xrelease` subset of TSX instructions, which extend the traditional interface to offer a slightly higher-level way to access the CPU feature, optimized for simplicity for locking-like synchronization patterns [66, 118] In this thesis I focus on RTM, the more general (i.e., expressive (i.e., bug-prone)) interface, and among all these acronyms restrict myself to "HTM" (when referring to transactional memory as a concurrency primitive in the abstract) and "TSX" (when referring to Intel's implementation and/or GCC's intrinsics interface). The non-pedantic reader may treat these as interchangeable.

The example TSX program from Figure 2.3 (§2.1.4) is reproduced here for the reader's convenience.

```
1  if ((status = _xbegin()) == _XBEGIN_STARTED) {
2      x++;
3      _xend();
4  } else {
5      mutex_lock(&m);
6      x++;
7      mutex_unlock(&m);
8  }
```

Figure 6.1: Example TSX program.

6.1 Concurrency Model

While up to now Landslide's tested programs' concurrency has been limited to timer-driven thread scheduling, HTM presents a fundamentally new dimension of nondeterminism, namely the hardware's ability to revert execution sequences and the delayed visibility of changes to other threads. In order to efficiently test HTM programs in Landslide, in this section I develop a simpler concurrency model and offer a proof of equivalence to HTM execution semantics. I make two major simplifications: simulating transaction aborts as immediate failure injections, and treating transaction atomicity as a global mutex during data-race analysis; and provide corresponding equivalence proofs.

Notation. Let $I = TN_1@L_1, TN_2@L_2, ...TN_n@L_n$, with N_i a thread ID and L_i a code line number, denote the execution sequence of a program as it runs according to the

specified thread interleaving. This serialization of concurrent execution is told from the perspective of all CPUs at once and hence assumes sequential consistency. For discussion of relaxed memory models refer to Section 6.4.

6.1.1 Example

Consider again the program in Figure 6.1. Note that the C-style `x++` operations, when compiled into assembly, become multiple memory accesses which can be interleaved with other threads.

```
2a  temp <- x;
2b  temp <- temp + 1;
2c  x <- temp;
```

If these instructions from the `x++` in the transaction are preempted, with another thread's access to `x` interleaved in between, the transaction will abort. So, the interleaving

$$\textbf{T1}@1, \textbf{T1}@2a, \textbf{T1}@2b, \textbf{T2}@1, \textbf{T2}@2, \textbf{T2}@3, \textbf{T1}@2c, \textbf{T1}@3$$

or, henceforth abbreviated for clarity:

$$\textbf{T1}@1-2b, \textbf{T2}@1-3, \textbf{T1}@2c-3$$

is not possible; rather, **T1** will fall into the backup path:

$$\textbf{T1}@1-2b, \textbf{T2}@1-3, \textbf{T1}@4-7$$

However, the `x++` operation from the failure path (correspondingly $6a$, $6b$, $6c$) *can* be thusly separated with conflicting accesses interleaved in between, since the mutex only protects the failure path against other failure paths, but not against the transaction itself. So (assuming `x` is intended to be a precise counter rather than a sloppy one), the following interleaving exposes a bug:[1]

$$\textbf{T1}@1-2b, \textbf{T2}@1-3, \textbf{T1}@4-6b, \textbf{T3}@1-3, \textbf{T1}@6c-7$$

Prior work [29] proposed the idiom shown in Figure 6.2 to exclude this family of interleavings, which shows that correctly synchronizing even the simplest transactions may be surprisingly difficult or complex.

6.1.2 Modeling Transaction Failure

In the previous section's examples, the way I stated interleavings such as $\textbf{T1}@1-2c, \textbf{T2}@1-3, \textbf{T1}@4-7$[2] glossed over how such a sequence of operations would be carried out under

[1] Note also that this bug requires either at least 3 threads or at least 2 iterations between 2 threads to expose; this highlights MC's dependence on its test cases to produce meaningful state spaces in the first place.

[2] For a clearer example to follow, I have reordered **T1**'s write to `x` before **T2**'s part, compared to before.

```
   bool prevent_transactions = false;

0  while (prevent_transactions) continue;
1  if ((status = _xbegin()) == _XBEGIN_STARTED) {
2      if (prevent_transactions)
3          _xabort();
4      x++;
5      _xend();
6  } else {
7      mutex_lock(&m);
8      prevent_transactions = true;
9      x++;
A      prevent_transactions = false;
B      mutex_unlock(&m);
C  }
```

Figure 6.2: Variant of the program in Figure 6.1, with additional synchronization to protect the failure path from the transactional path. The optional line 0 serves to prevent a cascade of failure paths for the sake of performance by allowing threads to wait until transacting is safe again.

HTM. For example, **T1**'s write during $2c$ is not actually visible to **T2**, although it would be under a thread-scheduling-only concurrency model.

Intel's official TSX documentation summarizes its interface and behaviour in prose [68]. Recent work has used proof assistants to formalize some of the execution semantics of x86 in general [119] and of transactions in particular (at both hardware- and language-level) [21]. However, state-of-the-art advances in model checking algorithms still state their theorems and proofs in prose [1, 16, 24, 31, 63, 76, 147], so this section's proofs will regrettably do likewise, leaving the rigor of mechanization to future work. The reader may at least rest assured that the proofs herein rely on transactional semantics that have themselves been formally verified.

To summarize HTM's execution semantics:

1. Any modifications to shared state (such as $2c$) by a transacting thread are not visible to any other during the transaction (such as $2c$ in this example, despite **T2** executing afterwards).
2. All local and global state changes during a transaction (such as **T1**'s lines $1 - 2c$ in this example) are discarded when returning an abort code from `xbegin` (jumping to line 4, in this example).

While use of HTM in production requires the performance advantage of temporarily staging such accesses in local CPU cache, model checking such programs need be concerned only with the program's *observable* behaviours. I claim that MCing the simpler interleaving **T1**@1, **T2**@$1 - 3$, **T1**@$4 - 7$ is an equivalent verification as MCing the one above; in fact, this interleaving suffices to check all observable behaviours of all interleav-

ings of all subsets of T2@$1-3$ with all subsets of T1@$2a-2c$, whether they share a memory conflict or not. Stated formally, let:

- Ti@α be an HTM begin operation,
- Ti@$\beta_1 \ldots$ Ti@β_n be the transaction body (with β_n the HTM end call),
- Ti@$\phi_1 \ldots$ Ti@ϕ_m be the failure path, and
- Ti@$\omega_1 \ldots$ Ti@ω_l be the subsequent code executed unconditionally.

Note that arbitrary code may not be structured to distinguish these as nicely as in the examples; e.g., more code may exist in the success branch after `_xend()`; such would be considered part of ω here.

Then, without loss of generality (for any number of other threads Tj/Tk, and for any number of thread switches away from Ti during the transaction):

Lemma 1 (Equivalence of Aborts)**.** *For any interleaving prefix*

$$\begin{gathered}\text{Ti@}\alpha, \text{Ti@}\beta_1 \ldots \text{Ti@}\beta_b, \\ \text{Tj@}\gamma_1 \ldots \text{Tj@}\gamma_j, \\ \text{Tk@}\kappa_1 \ldots \text{Tk@}\kappa_k, \\ \text{Ti@}\beta_{b+1}\end{gathered}$$

with $b < n$, $j \neq i$, $k \neq i$, etc., either:

1. Ti@α, Tj@$\gamma_1 \ldots$ Tj@γ_j, Tk@$\kappa_1 \ldots$ Tk@κ_k, Ti@$\phi_1 \ldots$ *(conflicting case), or*
2. Ti@α, Ti@$\beta_1 \ldots$ Ti@$\beta_b \ldots$ Ti@β_n, Tj@$\gamma_1 \ldots$ Tj@γ_j, Tk@$\kappa_1 \ldots$ Tk@κ_k *(independent case)*

exists and is observationally equivalent.

Proof. Case on whether the operations by Tj and/or Tk have any memory conflicts (read/write or write/write) with Ti@$\beta_1 \ldots$ Ti@β_n. If so, then the hardware will abort Ti's transaction, discarding the effects of Ti@$\beta_1 \ldots$ Ti@β_n and jumping to Ti@ϕ_1, satisfying case 1. Otherwise, by DPOR's definition of transition dependence ([46], §3.4.2), Ti@$\beta_{b+1} \ldots$ Ti@β_n is independent with the transitions of Tj and Tk, may be successfully executed until transaction commit, and reordering them produces an equivalent interleaving, satisfying case 2. □

The claim's second part follows naturally.

Theorem 1 (Atomicity of Transactions)**.** *For any state space S of a transactionally-concurrent program, an equivalent state space exists in which all transactions are either executed atomically or aborted immediately.*

Proof. For every $I \in S$ with Ti@α, Ti@$\beta_1 \ldots$ Ti@β_b, Tj@$\ldots$, Tk@$\ldots$, Ti@$\beta_{b+1} \in I$, apply Lemma 1 to obtain an equivalent interleaving I' satisfying the theorem condition. The resulting S' can then be MCed without ever simulating HTM rollbacks. □

6.1.3 Memory Access Analysis

Next comes the issue of memory accesses within transactions with regard to data-race analysis (§2.3). Theorem 1 provides that the body of all transactions may be executed atomically within the MC environment. While they may interleave between other non-transactional sequences, no other operations (whether transactional or not) will interrupt them. I claim this level of atomicity is equivalent to that provided by a global lock, and hence abstracting it as such in Landslide's data-race analysis is sound.

Let $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ be a pair of memory accesses to the same address, at least one a write, in some transactional execution I normalized under Lemma 1. Then let $\mathsf{lockify}_m(\mathbf{Tk}@L)$ denote a function over instructions in I, which replaces $\mathbf{Tk}@L$ with $\mathbf{Tk}@\mathsf{lock}(m)$ if L is a successful HTM begin, with a no-op if L is a transaction abort, or with $\mathbf{Tk}@\mathsf{unlock}(m)$ if L is an HTM end, or no replacement otherwise. Finally, let $I' = \exists m.\mathsf{lockify}_m(I)$, the execution with the boundaries of all successful transactions replaced by an abstract global lock. Lemma 1 guarantees mutual exclusion of m.

Theorem 2 (Transactions are a Global Lock)**.** $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ *is a data race in* I *iff it is a data race in* I'*.*

Proof. I prove one case for each variant definiton for data races supported in Landslide [16]. For each, I state below what it means to race in an execution with synchronizing HTM instructions.

- **Limited Happens-Before.** To race in I they must be reorderable at instruction granularity, at least one with a thread switch immediately before or after. [110, 125].
  - $I \Rightarrow I'$: If $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ race in I, then they cannot both be in successful transactions, or else placing $\mathbf{Ti}@\mu$ within the boundaries of $\mathbf{Tj}@\nu$'s transaction would cause the latter to abort, invalidating $\mathbf{Tj}@\nu$, or vice versa. Hence they will not both hold m in I'. Otherwise their lock-sets and DPOR dependence relation remain unchanged.
  - $I' \Rightarrow I$: If $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ race in I', both corresponding threads cannot hold m; WLOG let $\mathbf{Ti}$ not hold m during $\mathbf{Ti}@\mu$. Then in I, $\mathbf{Ti}@\mu$ is not in a transaction. With the remainder of their lock-sets still disjoint, and still not DPOR-dependent, $\mathbf{Tj}@\nu$ (or its containing transaction) can then be reordered directly before or after $\mathbf{Ti}@\mu$.
- **Pure Happens-Before.** WLOG fix $\mathbf{Ti}@\mu \prec \mathbf{Tj}@\nu \in I$. Then to race in I there must be no pair of synchronizing instructions $\mathbf{Ti}@\epsilon$ (a release edge) and $\mathbf{Tj}@\chi$ (an acquire edge) such that
$$\mathbf{Ti}@\mu \prec \mathbf{Ti}@\epsilon \prec \mathbf{Tj}@\chi \prec \mathbf{Tj}@\nu \in I$$
to update the vector clock epoch between $\mathbf{Ti}@\mu$ and $\mathbf{Tj}@\nu$ [45, 116].
  - $I \Rightarrow I'$: If $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ race in I, then they cannot both be in successful transactions, or else Lemma 1 normalization would provide the corresponding HTM end and begin for $\mathbf{Ti}@\epsilon$ and $\mathbf{Tj}@\chi$ respectively. Hence there will be no unlock/lock pair on m in I' to satisfy the above sequence.

- $I' \Rightarrow I$: If **Ti**@μ, **Tj**@ν race in I', then they cannot both hold m, or else lockify$_m$ would provide the corresponding unlock and lock for **Ti**@ϵ and **Tj**@χ respectively. Hence there will be no HTM end/begin pair in I to satisfy the above sequence.

Hence, data-race analysis is sound when transaction boundaries are replaced by an abstract global lock. □

6.2 Implementation

Support for TSX programs in Landslide is implemented in four parts, broadly speaking: the user interface, plus one internal part corresponding to each proof above, plus a bonus optimization for pruning equivalent interleavings under the new concurrency model.

6.2.1 User Interface

Landslide provides its own "implementation" of the TSX interface, which matches GCC's interface exactly, located in `410user/inc/htm.h` under `pebsim/p2-basecode/`. The interface is implemented in `410user/libhtm/htm.c`, perhaps surprisingly, as totally empty functions; Landslide hooks these functions' addresses during instrumentation and inserts preemption points, failure injections, et cetera as necessary whenever the execution encounters them.

Transactional test programs should be ported to the Pebbles userspace if not already, then any use of compiler HTM intrinsics replaced with Landslide's interface[3]. They should then be put in either `410user/progs/` or `user/progs/` and the `410TESTS` or `STUDENTTESTS` line (respectively) of `config-incomplete.mk` be edited to add the test name, before running `p2-setup.sh` (§3.1.1) on a (hopefully) correct P2 as usual. Several example HTM tests are provided as `410user/progs/htm*.c`.

Finally, Quicksand supports the following command-line options to enable various sets of HTM features within Landslide.

- `-X` (for "tsX" or "Xbegin") enables the basic features: preemption points on each `_xbegin()` and `_xend()` call, failure injection on each of the former (always returning `_XABORT_RETRY` as the failure code) (§6.2.2), and treatment of transactional regions during data-race analysis (§6.2.3).
- `-A` (for "xAbort codes") enables multiple xabort failure codes (§6.2.2). Requires `-X`.
- `-S` (for "Stm" or "Suppress retries") disables the `_XABORT_RETRY` failure reason, causing Landslide to emulate the semantics of STM rather than HTM. Requires `-X -A`.
- `-R` (for "Retry sets") enables state space reduction based on independences between individual `xbegin` results (§6.2.4). Requires `-X` and *not* `-A`.

[3]Attempts to execute a real TSX instruction under Landslide, instead of using the custom interface, will be reported as "invalid opcode" bugs, as neither of its supported simulation platforms support the feature.

6.2.2 Failure Injection

Each preemption point (`struct hax` as defined in `tree.h`) includes three new fields. The boolean `h->xbegin` is set if the preemption point occurred at an `_xbegin()` call. Then if set (as the tag of an option type), the twin lists of integers `h->xabort_codes_ever` and `h->xabort_codes_todo` store possible error codes this `_xbegin()` call should be tested to possibly return. The former list stores all such error codes, whether already tested or yet to be tested, while the latter serves as a workqueue that indicates only those not already tested yet (serving an analogous purpose as the `all_explored` flag for thread scheduling). The state space estimators (§3.4.3) check the length of these lists, in addition to DPOR's tagged threads, when computing the number of marked children.

Adding possible abort codes

Up to four abort codes are considered depending on the testing options listed in the previous section. All abort codes are simultaneously added to both lists, unless already present on `xabort_codes_ever`, in which case not added to `xabort_codes_todo` to avoid duplicate work.

- `_XABORT_RETRY`: When a `xbegin` preemption point is created (`save_setjmp()`), if `-S` is not set, both `xabort_codes` lists are initialized to contain this code. This represents the possibility for a hardware transaction to fail for reasons outside of the programmer's control such as system interrupts or cache eviction. If `-S` is set they are initialized to empty, representing either STM's policy of failing only when a true memory conflict arises, or a TSX user wrapping all her `_xbegin()`s in a retry loop such as in [13].
- `_XABORT_CONFLICT`: Whenever DPOR detects a memory conflict between two transitions (`shimsham_shm()`), if `-A` is set, if the later-executed transition is a transactional success path, it adds this code to both lists of the immediately preceding preemption point. Note that a transaction should suffer a conflict abort only when executed *after* a conflicting memory access to avoid circular causality (see `410user/progs/htm_causality.c` for rationale). Should the transaction happen to be executed first, DPOR will first try to reorder it as normal, and then abort it as described if the conflict persists.
- `_XABORT_EXPLICIT`: Whenever the program invokes `_xabort()` (`sched_update_user_state_machine()`), if `-A` is set, this code is added to both lists, bitwise-ored with the user-supplied argument code as specified in [47], and execution of the transactional path immediately stops.
- `_XABORT_CAPACITY`: Whenever the program invokes a system call during a transaction, if `-A` is set, this code is added to both lists, and execution of the transactional path immediately stops.

Limitations. Landslide does not yet check for false sharing, i.e. read/write or write/write access pairs to different memory addresses that share a cache line. On real hardware, these would produce `_XABORT_CONFLICT` failures, but to find such access pairs would

require extending `mem_shm_intersect()`'s set intersection algorithm to consider a certain degree of (N-byte-aligned) fuzziness when comparing addresses, as well as adding a command-line option to configure said N, the cache-line size to simulate. For now, Landslide (wrongly) lumps false sharing in among the "spurious" `_XABORT_RETRY` failure reasons. On HTM, even if the user wraps all such spurious failure reasons in a retry loop, false sharing (i.e., disjoint memory accesses that share a cache line) should still produce a non-retryable abort, begetting a discrepancy with STM, which aborts only when the memory addresses match exactly. Accordingly, the `-S` option described above provides STM semantics as currently implemented. Future work could extend Landslide with an option to configure false sharing conflicts to remain faithful to HTM semantics and still abort even under `-S`. Likewise, Landslide does not check for a transaction's memory footprint exceeding the CPU's cache capacity, which on real hardware would trigger a `_XABORT_CAPACITY` abort. Landslide's memory tracking could simulate this check as well, perhaps with a configurable cache size, but would likely be theoretically uninteresting (all programs in the upcoming evaluation have trivial memory usage), and so is left unimplemented for now. Finally, `_XABORT_NESTED`, the last abort code specified by [47], depends on currently-unsupported `xbegin` nesting, which I discuss further in §6.4.

Injecting abort codes

When traversing the state space (§3.3.5), in addition to performing DPOR to select non-independent thread interleavings (§3.4.2), the abort codes under each `xbegin` preemption point are also considered "marked" paths which must be tested. Hence `explore()`, by way of `any_tagged_child()`, will pop off the `xabort_codes_todo` queue when it's time to explore that preemption point in the usual depth-first manner[4]. The optional abort code is then passed through `arbiter_append_choice()/arbiter_pop_choice()` to `cause_transaction_failure()`, which edits the simulation state (`%eip` and `%eax`) to force `_xbegin()` to return the provided code.

6.2.3 Data Race Analysis

When a thread returns `_XBEGIN_STARTED` from `_xbegin()` (analogous to `mutex_trylock()`), Landslide's scheduler sets the `user_txn` action flag for that thread (§3.3.2), and if using Pure Happens-Before, applies FT ACQUIRE (§3.4.4) using a dummy lock address to represent the abstract global lock. When a thread reaches `_xend()`, the flag is cleared, and under Pure Happens-Before, FT RELEASE is applied. Then when `check_locksets()` compares an access pair, under Limited Happens-Before, it is considered a data race only if at least one thread's `user_txn` was not set in addition to the usual conditions; under Pure Happens-Before, the vector clocks are simply checked as usual.

[4]The search order prioritizes abort codes before scheduling other threads at such preemption points, which is just an implementation detail, not theoretically necessary.

6.2.4 Retry independence

Finally, I identified a specific pattern of transactional code where existing state-space reduction algorithms will fail to identify and prune equivalent thread interleavings. Figure 6.3 shows a minimal example program which exhibits this problem. In this program, each thread's transactional path conflicts with the other thread's abort path, while the two transactional paths (disjoint memory accesses) and the two abort paths (reads only) are pairwise independent.

Initially `int foo, bar = 0;`

Thread 1

```
if (_xbegin() == STARTED) {
    foo++;
    _xend();
} else {
    assert(foo + bar < 2);
}
```

Thread 2

```
if (_xbegin() == STARTED) {
    bar++;
    _xend();
} else {
    assert(foo + bar < 2);
}
```

Figure 6.3: Motivating example for retry independence reduction.

Ordinarily, in the state space subset which schedules thread 1 before thread 2, there would be 4 combinations of each thread succeeding or aborting their respective transactions; among those, at least one would show a memory conflict, causing DPOR to explore into the other half of the state space which schedules thread 2 first, where the same 4 combinations of transaction reuslts would be tested in the other order, even though 2 are redundant under schedule reordering. Consequently, naïvely testing both success and retry aborts in both threads regardless of reordering will unnecessarily execute both equivalent interleavings from each such pair; to identify such equivalences, DPOR must somehow remember which combination of `xbegin` results led to the memory conflict in the first place.

While Figure 6.3's example may seem contrived (what program's transactions would just give up and do no work if aborted?), it is easy to imagine a larger transactional data structure, two insertions into which might operate on disjoint nodes or array indices, allowing simultaneous transactions to usually succeed, while the more uncommon abort paths might take the opportunity to assert a full consistency check of all elements, ultimately resulting in a similar conflict pattern. Also, the evaluation will later show (§6.3.2) that even programs with fully conflicting success/abort paths may still exhibit some equivalent thread interleavings of this pattern after their failure paths are split apart into smaller transitions by data-race preemption points.

To address this, I extended Landslide with *retry set reduction* (commit 86657c7), an experimental feature named after prior work's analogous *sleep sets* ([1, 46, 49], §3.4.2). Whenever DPOR tags a new branch for exploration (§3.4.2), if either or both sides of a memory conflict were part of a transaction, it records a *retry set*, i.e., the pair of `xbegin` results executed by the conflicting threads, to accompany that branch. Like the `xabort_codes`

lists, the state space estimators (§3.4.3) check the number of retry sets when counting the number of marked children. Later, when traversing the new branch, Landslide remembers those threads' expected `xbegin` results and refuses to test any others (unless DPOR separately found them, too, to conflict), thereby skipping over (i.e., pruning) reorderings of any other `xbegin` results that would be independent. Like sleep sets, it also considers the preempted thread "retry-set blocked" (like sleep-set blocking [1]), and refuses to run it until the conflicting thread runs its transaction first, unless such would result in deadlock[5]. Upcoming in the evaluation, Figure 6.6 will visualize the reduction achieved in two test cases.

6.3 Evaluation

While prior work has focused on verifying implementations of transactional memory themselves [33, 53, 54, 111], Landslide is to the best of my knowledge the first model checker to support transactional client code. Accordingly, there is no baseline against which to compare Landslide's performance. Likewise, since Landslide's HTM semantics emulation relies on the equivalences proved in §6.1, I did not actually implement an HTM-style speculative-execution-and-rollback simulation mode. On this count, at least, I hope the reader finds it self-evident that the equivalence proofs provide exponential state space reduction compared to actually testing (and thereafter aborting) every combination of instructions within transactions. Beyond those, this chapter's evaluation will take a relatively green-field and exploratory approach. I pose the following evaluation questions.

1. How quickly does Landslide find bugs in incorrect transactional programs?
2. Does Landslide find any previously-unknown bugs in real-world transactional code?[6]
3. How does Iterative Deepening's (§4.2) performance compare to Maximal State Space mode (§3.1.2)?[7]
4. How well does Landslide's verification scale with increasing thread/iteration count for correct transactional programs?
5. What further reduction can be achieved beyond the baseline provided by the global-lock/failure-injection equivalences?

[5] Before the search ordering update to Landslide's normal sleep sets implementation (§3.4.2), I observed false-positive retry-set-blocked deadlocks fairly often; after the update, it took until 981773 interleavings (>9 hours) into `htm2`(3,2) to find one and confirm the need to still explicitly avoid them by abandoning the retry set in such cases.

[6] The savvy reader will realize that whether or not the author poses this evaluation question to begin with spoils its answer.

[7] Not necessarily related to HTM, but the latter was implemented well after Chapter 4's conference paper was published, so this was the most convenient test suite to evaluate it on.

Experimental setup

The evaluation suite comprises several unit tests hand-written by yours truly, microbenchmarks and transactional data structures from [29], a transactional spinlock from [123], and various combinations thereof, as follows.

- Unit tests and microbenchmarks
  - `htm1`: The bug from Figure 2.3/6.1.
  - `htm2`: The fixed version as in Figure 6.2.
  - `counter`: Microbenchmark version of `htm2` which replaces the complex locking failure path with an atomic `xadd`, from [29].
  - `swap`: Microbenchmark that swaps values in an array, from [29].
  - `swapbug`: `swap` modified to introduce circular locking in the failure path.
  - `fig63`: Generalized version of Figure 6.3, contrived to induce as much reduction as possible from the retry sets optimization.
- Data structure tests
  - `avl_insert`: AVL tree concurrent insertion test [29].
  - `avl_fixed`: `avl_insert` with the AVL bug fixed (spoilers!!).
  - `map_basic`: Separate-chaining hashmap concurrent insertion test [29].
  - `map_basicer`: `map_basic` modified with a larger initial size to skip the resizing step.
  - `avl_mutex`: `avl_fixed` with transactional sections simplified by abstraction into a mutex.
  - `map_mutex`: `map_basic` simplified likewise.
- Lock abstraction tests
  - `lock()`: Checks that multiple threads using a transactional lock cannot access the critical section simultaneously.
  - `lock_fast()`: Checks that a transactional lock's fast path will not suffer conflict aborts if its client threads' critical sections are independent.

  These are each parameterized over implementations `spinlock` (from [123]), `spin_fixed` (spoilers!!), and `mutex` (replaces the spinlock with a Landslide-annotated P2 mutex to reduce state space size).

The notation `testname`(K, N) will denote a test configuration of K threads, each running N iterations of the test logic. All tests were run on an 8-core 2.7GHz Core i7 with 32 GB RAM. Reported CPU-times include time spent on all state spaces Quicksand saw fit to run, not just the maximal or the buggy state space; for verification tests (run with `-M`), this still includes abandoned smaller jobs that were run to saturate the set of data-race preemption points (Chapter 4). To minimize variance in CPU-time measurements, I ensured the test machine was not loaded beyond normal web browser use, and ran only one instance of Landslide at a time; for further discussion of variance see [13]. The number

of interleavings in each state space are, of course, deterministic and do not vary across runs.

I investigated the popular transactional benchmark suite STAMP [98] to include in this test suite, but found that all transactional code therein was written without failure paths, so would likely not contribute any theoretical depth to the evaluation. STAMP uses the OpenTM interface [8], which allows the programmer to specify transactional code regions to be implemented atomically, however the underlying architecture may require (whether HTM or STM; presumably with retry loops on HTM). On one hand, this lends credence to my upcoming conclusion that HTM programs should be written at a higher abstraction level than calling `_xbegin()` directly; on the other, OpenTM's requirement of *virtualized transaction* semantics (i.e., being unconstrained by memory footprint, able to make system calls, and able to nest arbitrarily) is more suited to STM and glosses over many pitfalls of HTM programming.

Finally, the keen-eyed reader will notice the state space sizes reported here differ from those reported in [13]. All experiments have been re-run on account of three updates to Landslide's exploration algorithm implemented since then: the sleep sets optimization for DPOR (§3.4.2, commits 0447666 and 588687c), the `thrlib_function` and `TRUSTED_THR_JOIN` directives to mark internal thread library logic as trusted (§3.2.1, commits 64a02e4 and a50d4ea), and fixing a soundness bug in which Landslide could neglect to inject transaction failures immediately after a thread switch (commit dcae85b). On account of the former two updates, some state spaces may be smaller than before; on account of the third, some may be larger. These updates do not discredit the bug-finding results (a bug is a bug), but the previously-published verification results should be considered outdated.

6.3.1 Bug-finding

Table 6.1 presents the bug-finding results. I configured Landslide to run Quicksand's Iterative Deepening algorithm on 8 cores, shown left, as well as to prioritize the maximal state space, shown right, each with a time limit of 1 hour. Tests `htm1`, `swapbug`, and `avl_insert` were run with `landslide -X` (i.e., retry aborts enabled and different abort codes not distinguished); `lock_fast` was run with `landslide -X -A -S` (i.e., suppressing retry aborts, due to the spinlock's use of a retry loop confirmed with manual inspection).

Finding bugs quickly

As the test parameters increase, the multiplicative factor in bug-finding speed (2-4x, eyeballing) is generally smaller than that of the total number of interleavings (10-1000x). In other words, should transactional bugs exist, Landslide is likely to find them reasonably quickly despite prohibitive exponential explosion in total state space size. This corroborates the results from Chapter 4, extending its good news to the world of HTM.

buggy test	K,N	Quicksand mode cpu (s)	wall (s)	int's	Maximal state space mode (-M) cpu (s)	wall (s)	int's	SS size (est.)
`htm1` (assertion)	2,1	26.48	6.48	5	*8.21	*5.78	5	*12*
	2,2	45.23	8.28	9	*8.42	*5.97	9	*102*
	2,3	53.88	9.92	17	*8.80	*6.31	17	*819*
	2,4	90.24	14.63	33	*9.72	*7.19	33	*6553*
	3,1	39.75	8.21	5	*8.15	*5.71	5	*76*
	3,2	46.91	8.40	9	*8.37	*5.91	9	*3686*
	3,3	41.39	8.50	17	*8.71	*6.26	17	*176947*
	3,4	92.98	15.19	33	*10.36	*7.15	33	*8493465*
	4,1	45.13	8.56	5	*7.77	*5.30	5	*460*
	4,2	64.99	11.33	9	*8.32	*5.59	9	*132710*
`swapbug` (deadlock)	2,1	*26.25	*6.42	*6	47.80	13.43	33	*73*
	2,2	*18.08	*4.98	*10	51.37	16.78	85	*860*
	2,3	*20.93	*5.57	*18	57.87	23.99	217	*9120*
	2,4	*38.92	*8.61	*34	82.95	48.31	537	*91239*
	3,1	*38.59	*8.50	*32	88.28	72.41	1016	*3543*
	3,2	*1572.83	*199.98	*262	–	*>1h*	–	*1683509*
`avl_insert` (segfault+)	2,2	2494.77	315.46	*29	*95.53	*30.56	79	*158505*
	2,3	308.10	*43.95	*33	*249.42	144.33	835	*13664203*
	2,4	*2979.29	*390.34	*1457	–	*>1h*	–	*61882736*
	3,1	*87.10	*14.81	*14	94.08	23.60	24	*207575*
	3,2	*3672.84	*475.03	*145	–	*>1h*	–	*1635075071*
`lock_fast` (perf)	2,1	18.33	5.19	2	*3.12	*3.12	2	*4*
	9,9	22.43	6.24	2	*4.71	*4.71	2	*inf*

Table 6.1: Landslide's bug-finding performance on various test configurations. Iterative Deepening (§4.2), optimized for fast bug-finding, is compared against Maximal State Space mode (§3.1.2), optimized for fast verification. For each, I list the CPU-time and wall-clock time elapsed, plus the number of interleavings tested in the ultimately buggy state space until the bug was found. * marks the winning measurements between each series. Lastly, state space estimation (§3.4.3) confers a sense of the exponential explosion.

Finding new bugs

In addition to the bugs I intentionally wrote in `htm1` and `swapbug`, Landslide also found two bugs in the "real-world" transactional algorithms I tested.

- **Atomicity violation.** `avl_insert` with any parameters higher than (2,1) exposed a previously-unknown bug in the transactional AVL tree. Figure 6.4 shows the root cause, essentially the `htm1` bug in disguise. This manifested alternately as a segfault (for test parameters (2,2) and (3,1)) and as a consistency-check assertion failure (for test parameters (2,3)). The presence of `while (_retry);` makes the necessary preemption window extremely small (between it and `_xbegin()`), making the bug extremely unlikely to manifest under stress testing, but Landslide is blind to such matters of chance.

  As a matter of full disclosure, I noted that the loop does not affect the test's possible behaviours, only its *likely* ones, and so removed it to make the test more Landslide-friendly. To dispel any doubt about bias or test hacking, I confirmed that Landslide

```
while (_retry);
if (_xbegin() == SUCCESS) {
    tie(_root,inserted) = _insert(_root,n);
    _xend();
} else {
    pthread_mutex_lock(&_tree_lock);
    _retry = true;
    tie(_root,inserted) = _insert(_root,n);
    _retry = false;
    pthread_mutex_unlock(&_tree_lock);
}
```

Figure 6.4: Unmodified code from `htmavl.hpp` showing the previously-unknown bug Landslide found in `avl_insert`. The transaction path fails to check `_retry`, leading to data races and corruption just as in `htm1`.

still finds the bug with the spin loop unmodified, on (3,1) in the same 53 interleavings, although it suffers resource exhaustion on parameters of (2,2) or greater.

- **Spurious spinlock abort.** `lock_fast` discovered a spurious transactional-path write conflict in the `spinlock` HTM-lock implementation.[8] This "performance bug" causes the lock to suffer slow-path spin-locking even in cases where the user's thread transitions are completely independent (for example, locking the root of an AVL tree, then traversing in different directions to make disjoint modifications), which the test case detects with `_xtest()`. Figure 6.5 shows the root cause: the `isfree()` routine (corresponding to the AVL's `_retry`) used an atomic compare-and-swap that would always write to memory even without modifying it. I corrected this in the `spin_fixed` implementation by replacing it with a normal read (being used only in the transactional path, no barriers are required to protect it). A cursory search on Github found one user of this code, a transactional LevelDB implementation [137], whose author had also noticed and corrected this problem in the same way.

  As another matter of full disclosure, I noticed this bug through manual inspection while adapting the spinlock's client code to be Landslide-friendly, then wrote `lock_fast` specifically to target this behaviour, so unlike `avl_insert`, it does not count as Landslide finding a previously-unknown bug. However, I feel in retrospect that how and when an HTM-backed concurrency abstraction will fall into its slow path is a reasonable performance property for a user to want to verify, so I consider Landslide confirming the bug (and later verifying its absence, in §6.3.2) a positive result anyway.

Note that in the AVL tree bug, the code's author was the very same person who proposed the protocol in Figure 6.2, yet still got it wrong once, having to write it out by hand

[8] `lock_fast`'s unusual (9,9) parameter shows that this state space size is constant: DPOR will always either deem all thread transitions independent and end exploration immediately, or the test's assertion will trip as soon as the first conflict is found.

```
bool hle_spinlock_isfree(spinlock_t *lock) {
    // XXX: should be "return lock->v == 0;"
    return __sync_bool_compare_and_swap(&lock->v, 0, 0);
}
void rtm_spinlock_acquire(spinlock_t *lock) {
    if ((tm_status = _xbegin()) == _XBEGIN_STARTED) {
        if (hle_spinlock_isfree(lock)) return;
        _xabort(0xff);
    } else {
        // ... retrying &c abbreviated for brevity ...
        hle_spinlock_acquire(lock);
    }
}
void rtm_spinlock_release(spinlock_t *lock){
    if (hle_spinlock_isfree(lock)) {
        _xend();
    } else {
        hle_spinlock_release(lock);
    }
}
```

Figure 6.5: Code from `spinlock-rtm.c`, modified only to remove unrelated logic for brevity, showing the performance bug Landslide found in `lock_fast(spinlock)`. The `isfree()` routine uses an atomic read-and-write operation where just a read would suffice, which leads to superfluous memory conflicts in the transactional path (seen at both of its callsites below).

throughout both data structures. This motivates the need for model checking such programs, no matter how much of a concurrency expert the author may be. It also suggests HTM primitives should be encapsulated behind higher-level abstractions, such as lock elision [66] or a simple spinlock [123], which can be verified in isolation with smaller state spaces then trusted in turn when checking their client programs [126]. §6.3.2 explores this further.

Regarding the spinlock bug, as HTM is fundamentally a performance-minded concurrency extension, the user may also care about more probabilistic properties of her code, such as requiring a transaction abort rate below a certain threshhold owing to the nature of its workload. Landslide cannot in general test for performance degradation bugs, because all interleavings are equal in Landslide's eyes, and probability is no object. However, `lock_fast` illustrates that model checking can still check some interesting performance properties as long as the element of probability can be removed. Future work may attempt to verify a wider range of performance properties, but with a hybrid approach between model checking and what other technique is not yet known.

Performance

Quicksand's ability to find bugs in fewer distinct interleavings (i.e., overall smaller state spaces) does not necessarily correlate with better performance in terms of CPU-time. Comparing Table 6.1's trends to the break-even point in Quicksand's evaluation (§4.3), most of these tests are too small for its approach to pay off, with `swapbug` and `avl_insert` as its notable wins. While plenty more wins were observed in §4.3, this suggests future MCs could prioritize state spaces using not just size estimation but a hybrid approach also conisdering state space maximality and preemption bounds [101] to soften the trade-off both for smaller tests and for verification.

6.3.2 Verification

For the test cases with no bugs found, I sought to provide Landslide's verification guarantee (§4.2.1) for up to as many threads and test iterations as possible under a reasonable time limit. The results, obtained using the same `-X -M` configuration options as in the previous section, are shown in Table 6.2 in the "Baseline DPOR" column. For test configurations which could not be verified within 10 wall-clock hours, I report their estimated state space size and runtime measured after that timeout instead, †*typeset thusly*.

Interpreting the verification guarantee

Landslide was able to verify most of these tests for a fair range of thread and iteration counts, often reaching up to 2 threads with 3-4 iterations each, 3 with 2 each, or 4 with 1. In the case of `htm2`, for example, verifying up to (K, N) represents a guarantee that, even repeating Figure 6.2's atomicity protocol N times in any scheduling sequence or combination of transaction aborts, it is impossible for K threads to violate the intended atomicity property (i.e., get 2 threads in the critical section simultaneously).

It is difficult to discern from prior work a concrete standard for what values of (K, N) constitute a "good" degree of verification. One recent paper [147], which likewise extended DPOR with a new dimension of concurrency (weak memory orderings), reported verifying programs with up to 10 concurrent events, presumably shared memory accesses. Another [1] reported test cases with as many as 19 threads, although with what must be very little synchronization or memory conflicts, as even their baseline DPOR checked only 4096 interleavings on that test. Table 6.3 confers a sense of the complexity of this evaluation's test suite, with the final column showing the approximate maximum number of preemption points reached among Landslide's verifications (i.e., $(\mathsf{txn}+\mathsf{sync}+\mathsf{race}) \times K \times N$ for the highest $K \times N$ completed).

In the case of `htm2`, it is easy to look at the program with human intuition and judge that, because the protocol's only state is stored in a single boolean, with no unbounded-capacity data structures or contention-dependent exponential backoff loops, it would be unimaginable that adding a 5th thread to the system could make any difference in correctness where 4 threads could not, or that a 4th repetition between 2 threads could make a difference where 3 could not, and ultimately that it must safely generalize to all (K, N).

test	K,N	Baseline DPOR cpu (s) (or †est.)	Baseline DPOR SS size (or †est.)	Retry sets (-R) cpu (s) (or †est.)	Retry sets (-R) SS size (or †est.)	STM (-A -S) cpu (s) (or †est.)	STM (-A -S) SS size (or †est.)
htm2	2,1	18.26	22	17.99	15	3.25	4
	2,2	63.61	1446	26.40	334	28.04	286
	2,3	3429.04	86536	196.18	6366	997.91	24740
	2,4	†29d 8h	†2710056	4771.17	123140	†5d 1h	†1792330
	3,1	43.24	774	24.02	224	22.79	140
	3,2	†1y 299d	†4510472	†todo	†todo	†10d 16h	†2322150
	4,1	9225.52	212146	376.70	11973	2158.93	44995
counter	2,1	6.92	10	6.81	8	3.28	4
	2,2	13.27	190	10.59	102	8.73	48
	2,3	155.09	3970	67.82	1558	40.08	904
	2,4	3664.66	86950	1150.34	25398	805.85	19128
	3,1	11.26	120	9.30	64	8.26	40
	3,2	2572.13	60606	2363.41	44862	639.62	14304
	4,1	129.25	3006	64.91	1296	40.78	848
swap	2,1	65.72	99	66.05	59	3.40	4
	2,2	18124.06	277824	1030.23	19542	703.82	11600
	3,1	3820.64	60912	608.48	10706	89.69	1014
fig63	2,1	7.12	10	6.92	6	3.40	1
	2,2	9.24	108	8.61	76	3.46	1
	2,3	54.68	1934	31.76	977	3.39	1
	2,4	1054.80	36600	417.99	14512	3.58	1
	3,1	11.15	148	7.76	22	3.53	1
	3,2	717.53	21642	217.30	6467	3.52	1
	4,1	111.83	3064	11.32	130	3.49	1
avl_insert	2,1	672.68	15125	307.94	6287	136.60	2774
avl_fixed	2,1	739.60	20459	332.32	9675	122.40	2774
	2,2	†58880y	†46150466	†todo	†todo	†36393y	†35094477
	3,1	†5y 292d	†2873664642	†todo	†todo	†97y 100d	†1698185036
map_basic	2,1	1950.48	30719	867.68	13237	367.91	5446
	2,2	†1y 176d	†75516602	†todo	†todo	†13d 11h	†565334
	3,1	†6y 133d	†1847957714	†todo	†todo	†4y 156d	†888242178
map_basicer	2,1	28.40	150	26.55	94	14.88	9
	2,2	†11h 17m	†727759	16455.24	283756	1285.06	21684
	3,1	†26h 15m	†1451708	21153.64	366030	705.04	12707

Table 6.2: Transactional tests verified (or not) by Landslide. Run with -M -X, plus any additional reduction options listed. "Baseline DPOR" always tests every abort path, without distinguishing among failure reasons (i.e., injecting only _XABORT_RETRY). "Retry sets" skips equivalent success path and/or retry abort reorderings (§6.2.4); "STM" suppresses retry aborts and dynamically detects when to inject conflict aborts and so on (§6.2.2). State space estimates measured after a timeout of 10 hours (and include those 10 hours in the predicted total).

test	#txn	#sync	#race	max events verified
htm2	1	2	4	42
counter	1	0	1	16
swap	1	4	8	52
fig63	1	0	K-1	18
avl_insert	1	2	7	20
avl_fixed	1	2	9	24
map_basic	1	4	13	36
map_basicer	1	2	5	32
lock(spinlock)	1	2	4	28
lock(spin_fixed)	1	2	4	28
lock(mutex)	1	2	2	30

Table 6.3: Number of concurrency events per iteration of each test case. Note that no test used any synchronization besides mutexes (the P2 thread API was annotated as trusted (§3.2.1) and so does not contribute to state space size). Also note that "#race" means the number of unique accesses identified as racy, rather than racing pairs (the other half of a pair might well be within a transaction, which cannot be preempted on).

Generalizing the verification is not so straightforward for more complicated algorithms, which may involve complex conflict patterns such as tree rebalancing or map resizing. Other formal verification approaches aside, the user must ultimately be content with the probabilistic assurance that as verified K and N increase, the likelihood that a bug exists which requires more threads or iterations to expose grows ever lower (for example, none of the bugs in §6.3.1 required any higher parameters than (2,2) to expose).

Nevertheless, pushing K and N higher is obviously desirable, even if it means applying reductions that require human intuition to trust are sound. Moreover, much as verifying `htm2`'s soundness is a positive result, the attempts at larger data structures quickly suffered exponential explosion for even small thread/iteration counts, in one case failing to verify whether or not a previously-found bug had actually been fixed. In the following subsections I explore three possible mitigation approaches.

Retry set reduction

Firstly, the middle column of Table 6.2 shows the impact of Landslide's experimental retry set reduction (§6.2.4). Despite its conservative implementation, it provides roughly 2-6x reduction in most tests, with up to 17x in extreme cases. The biggest win is apparent in `fig63`, the test contrived to induce as much reduction as possible using transactional paths that conflict only with aborts and not with each other, and vice versa. In `fig63`(2,1), corresponding to Figure 6.3, retry-set-enabled DPOR correctly prunes down to the optimal 6 interleavings, while the baseline treats it identically to the fully-conflicting `counter`(2,1). Figure 6.6 depicts the difference between the state spaces explored by the two approaches.

Perhaps surprisingly, retry sets also provide reduction even when transactional success and abort paths are fully conflicting, (i.e., all tests besides `fig63`). With just syn-

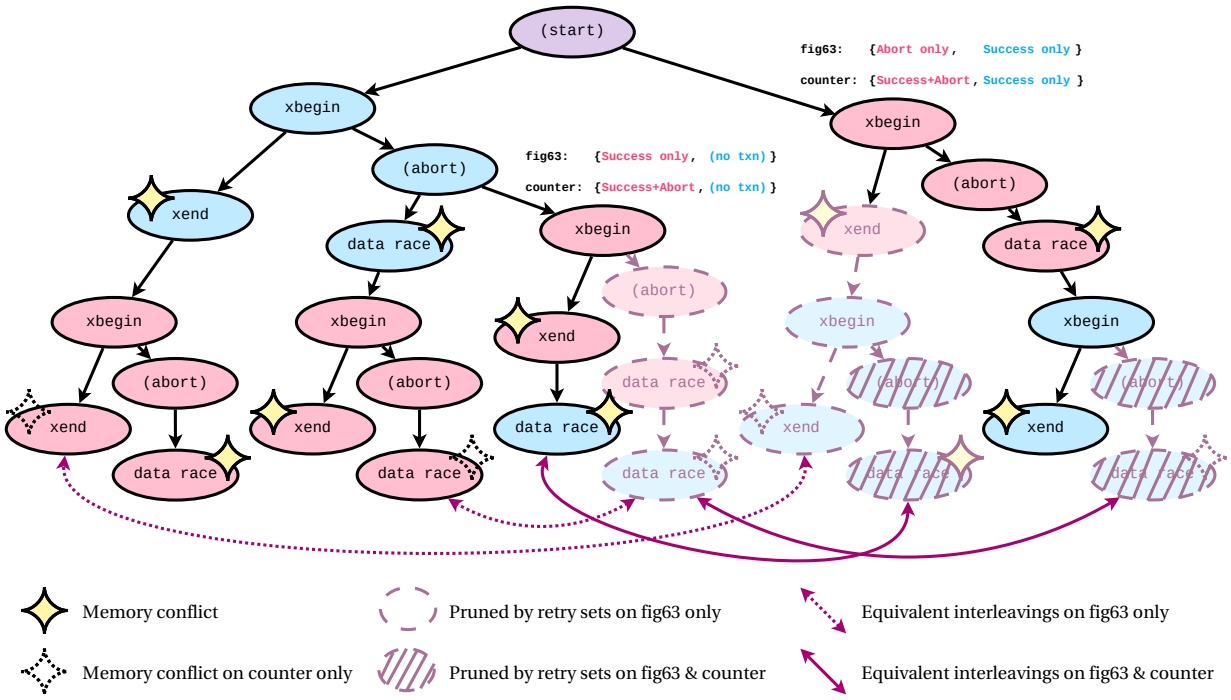


Figure 6.6: Visualization of retry set state space reduction. On both `counter`(2,1) and `fig63`(2,1), baseline DPOR tests all 10 interleavings pictured, with the middle 2 arising from the data-race preemption point within the abort path. With the reduction enabled, after the 4th and 6th branches (i.e., when preempting to reorder threads), Landslide activates the retry set indicated at the top of the next upcoming subtree, allowing it to identify and skip 2 redundant branches in `counter` and 4 in `fig63`.

chronization preemption points (including `_xbegin()` and `_xend()`), both the baseline and retry sets would explore exactly all 8 permutations of success and abort between two threads. However, in the presence of data-race preemption points, even for example on `counter`(2,1) (whose abort path is just 1 `xadd` operation), and whose optimal state space size should be 8 regardless of data-race preemption points), baseline DPOR tests reorderings of one thread's transaction both with the other's failure path, and with the other's data race therein[9]. Retry sets on the other hand identify and skip that equivalence (technically speaking, retry set DPOR reorders with the data race first during depth-first search, then skips generating a retry set for reordering the full abort path).

[9] Not pictured in Figure 6.6 is the symmetric subtree of branches 5-6 in the right half of the state space, which would occur after branch 10, reordering the blue thread before the pink's data race. Such would be equivalent to branches 2 and 4, and is pruned by the normal sleep set algorithm (§3.4.2, `equiv_already_explored()`) even in baseline DPOR, with no need for retry sets.

STM (abort code) reduction

Secondly, the state spaces could be reduced simply by restricting the concurrency model to only a subset of nondeterministic `xbegin` outcomes possible under HTM. Concretely speaking, the `-X -A -S` combination of Landslide options suppresses retry aborts (§6.2.2), which must be checked at every transactional preemption point, replacing them with explicit and conflict aborts, which Landslide injects only after identifying memory conflicts through DPOR or encountering an `xabort`, respectively, ultimately simulating the semantics of STM rather than HTM. This can allow for state space reduction when transactions happen to be non-conflicting, but more impactfully, conflict aborts can occur only *after* the other thread's conflicting access, so between a pair of transactions only the success, success and success, abort sequences need be tested; abort, success and abort, abort may (in fact, must) be skipped. The final column in Table 6.2 shows the result of STM semantics verification, which always results in at least 2x reduction compared to the baseline, although retry sets can make up some of the lost ground in some cases.

Abstraction reduction

Visual inspection of the AVL tree and separate-chaining map implementations [29], after correcting the former's atomicity bug (Figure 6.4), reveals that every use of HTM followed exactly the same pattern: running identical data structure logic in both the transactional and abort paths, as though HTM were merely a mutual exclusion lock with fancy performance characteristics. Prior work [126] proposed *abstraction reduction*, in which the user identifies program components that can be separated by a well-understood API, then tests each one against the API individually, effectively turning multiplicative state space size factors into additive ones.

In this case, I split the lock-like HTM use and the mutually-exclusive data structure code into separate tests, `lock`, which checks that the use of HTM guarantees mutual exclusion, and `avl_mutex/map_mutex`, which replace the open-coded HTM use with an already-trusted P2 mutex. `lock_fast`, a bonus test, checks the transactional lock's performance by asserting that its internal logic won't trigger conflict aborts even when the client's accesses are independent. Figure 6.7 shows their core logic. Finally, I parameterized them over how the lock was implemented: a real-world `spinlock` implementation from [123], `spin_fixed`, the same with the performance bug from §6.3.1 fixed, and `mutex`, using Landslide-annotated P2 mutexes (as the AVL and map implementations do).

Table 6.4 shows the new resulting levels of verification Landslide reached before the same 10-hour timeout. Provided that one trusts the `lock` tests correctly check the desired properties, and that open-coding hadn't introduced any new bugs (such as Figure 6.4's), the benefit is clear: the data structure tests' state spaces become much more tractable, their state space growth now defined only by internal conflicts from tree rebalancing, map resizing, and so on. In total, summing the testing times of `lock(mutex)`(K, N) and `avl_mutex`(K, N) produces the same verification as `avl_fixed`(K, N) far more cheaply. Furthermore, `lock`'s verification can be reused, whereas `avl_fixed` and `map_basic` effectively duplicated the mutex verification between them.

```
static int num_in_section = 0;
for (int i = 0; i < NITERS; i++) {
    rtm_spinlock_acquire(&lock);
    num_in_section++;
    if (!_xtest())
        thr_yield(-1);
    assert(num_in_section == 1);
    num_in_section--;
    rtm_spinlock_release(&lock);
}
```

(a) `lock()`, tests mutual exclusion.

```
for (int i = 0; i < NITERS; i++) {
    rtm_spinlock_acquire(&lock);
    assert(_xtest());
    rtm_spinlock_release(&lock);
}
```

(b) `lock_fast()`, tests for no spurious aborts.

Figure 6.7: Abstraction reduction test cases.

Note two curiosities: firstly, the impact that fixing Figure 6.5's performance bug (changing a read+write to a read only) had on even the correctness tests: `lock(spin_fixed)`'s state spaces were reduced by nearly half compared to `lock(spinlock)`, on account of DPOR no longer needing to reorder the (now) read-read access pairs. Secondly, `spinlock` and `spin_fixed` take longer to test per interleaving than `mutex` (roughly 9 interleavings per second for the former, 22 for the latter), because while `mutex` abstracts away threads needing to wait their turn for the critical section behind an API Landslide understands, the spinlock's wait loop is open-coded, and Landslide must fall back on its costlier heuristic synchronization detection (§3.4.6). In this way (and also, of course, because `mutex`'s state spaces are smaller overall), `mutex` can be seen in turn as a further abstraction reduction of `spinlock`.

6.4 Discussion

In this section I review some of the evaluation's results in a broader context, list the current limitations of Landslide's implementation, and discuss open problems for future work.

Retry set optimality

For all the reduction retry sets demonstrated in Table 6.2, some inefficiencies remain in its strategy. For example, it is not clear how to prune soundly when three or more threads must be reorderd around one transactional preemption point, or when a second pair of partially-independent transactions interleaves while an existing retry set is already active. Accordingly, I implemented the optimization as conservatively as possible in these cases, "saturating" the retry sets to fall back to no pruning (`update_hax_abort_set()` and `update_hax_abandon_abort_set()`, respectively).

Likewise, the cases of `htm2`(2,3), (2,4), and (4,1), in which retry sets achieved better reduction than STM mode, show that the latter does not necessarily subsume the former,

test	K,N	STM (-A -S) cpu (s) (or †est.)	STM (-A -S) SS size (or †est.)
lock (spinlock)	2,1	3.41	4
	2,2	198.77	1702
	3,1	35.28	246
lock (spin_fixed)	2,1	3.55	4
	2,2	105.57	998
	2,3	†33h 13m	†321553
	3,1	28.27	186
	3,2	†13y 281d	†1443676
	4,1	†16h 26m	†432628
lock (mutex)	2,1	3.44	4
	2,2	16.83	180
	2,3	405.21	9372
	2,4	24999.68	489480
	3,1	15.00	132
	3,2	†26h 42m	†1223955
	4,1	665.89	15064
lock_fast (spin_fixed)	2,1	3.25	1
	9,9	4.61	1
lock_fast (mutex)	2,1	3.19	1
	9,9	4.62	1

(a) Verifying HTM locks alone.

test	K,N	Non-transactional cpu (s) (or †ETA)	Non-transactional SS size (or †est.)
avl_mutex	2,1	3.48	7
	2,2	6.25	85
	2,3	24.69	561
	2,4	217.98	4984
	3,1	8.26	129
	3,2	1403.46	30653
	3,3	†todo	†todo
	4,1	199.96	4488
	4,2	†todo	†todo
map_mutex	2,1	39.81	83
	2,2	†26h 21m	†1085126
	3,1	†173d 17h	†12572187

(b) Verifying the lock's client code.

Table 6.4: Continuation of Table 6.2, demonstrating abstraction reduction [126] on the `avl_fixed` and `map_basic` tests by verifying HTM mutex implementations separately. Tested with STM semantics, as both lock implementations include a retry loop.

and that combining the two could in theory achieve further reduction still. However, `xbegin` results other than `_XABORT_RETRY` may depend on the execution logic (explicit aborts may be conditional on some change by a conflicting thread, and conflict aborts cannot occur before their conflicting transaction to begin with[10]), so how to soundly prune either success paths or retry aborts while other abort codes are in play remains an open problem.

While motivated by straightforward analogy to the known-sound sleep sets, the intersection of retry sets with Landslide's other exploration features may cause unforeseen problems. For now, its use is prohibited in conjunction with other state-space-affecting features such as ICB (§3.4.5) as well as multiple abort codes. I personally believe retry set reduction to be sound under these restrictions, having carefully scrutinized its behaviour while constructing Figure 6.3 and from inspecting state spaces arising from larger test parameters as well; nevertheless, this falls well short of formal proof, which I must leave to future work.

[10]See `410user/progs/htm_causality.c`; that was a fun realization to have already halfway into implementing retry sets the wrong way at first.

STM reduction soundness

In §6.3.2 I showed that state spaces could be reduced even further than with retry sets by assuming an HTM interface which abstracts away `_XABORT_RETRY` behind a loop. However, suppressing retry aborts is not guaranteed to faithfully test all possible behaviours observable under HTM. As an example, note in Table 6.2 how STM semantics reduced `fig63`'s state space on all (K, N) configurations to 1. Because its transactional paths are all mutually independent, DPOR identifies no need either to inject conflict aborts or to reorder threads. However, this skips the slow-path consistency assertion completely. If the programmer had intended it to run "every so often" at the whim of the timer interrupt, applying this reduction would be unsound. Also note the state space size of 4 for many (2,1) test configurations, corresponding exactly to the aforementioned success, success and success, abort sequences (times two ways to interleave the two threads). Because of the scheduling dependency for conflict aborts, Landslide cannot recognize the failure path's data races without a third freely-reorderable iteration; STM mode must be run with $K \times N \geq 3$ to meaningfully test conflicts between failure and success paths at all. A user wishing to distinguish conflict aborts, retry aborts, and so on during testing without glossing over any of HTM's peculiarities could supply the `-X -A` options without `-S`; however, this will inevitably result in state spaces at least as large as the baseline.

On the other hand, some programs may clearly annotate their intention for abort paths to be executed only in case of actual memory conflicts. `swap`, `avl_insert`, and `map_basic` abstract their `_xbegin()` calls behind an interface which can be implemented either with or without retry loops, while `lock(spinlock)` and `lock(mutex)` implement the retry loop directly. In these cases, the user can assure herself of STM reduction's soundness by visual inspection. In fact, Landslide's current implementation gets stuck in infinitely deep interleavings whenever it encounters a retry loop (bypassing even its heuristic infinite loop detection), so for now the user must inspect the test case to determine which testing mode to use. Future work could automatically identify a program's retry loops and give up on `_XABORT_RETRY` by switching to STM mode on-the-fly, much like Landslide's heuristic synchronization detection does for yield loops (§3.4.6).

Nested transactions

Whenever `_xbegin()` is called with a transaction already active, or `_xend()` while not, Landslide's current implementation immediately stops and reports a bug. However, just as concurrent programs often hold multiple locks simultaneously, one may wish to conduct multiple transactional routines simultaneously, especially as they may be abstracted across different code modules as a project grows in scale. Recent work [19, 34] has developed both implementations and formal semantics for executing nested transactions, so future work should extend the verification concurrency model to permit such programs. For now, the best Landslide can offer is to check the transactional components and their client code separately against their APIs with abstraction reduction (§6.3.2), then check the rest of the program with (for example) traditional mutexes that can nest safely.

Relaxed memory orderings

Section 6.1's formalization of thread interleavings does not account for read/write reorderings possible on relaxed consistency architectures [5]. In fact, even after [29]'s proposed fix to the atomicity protocol in Figure 6.2, it is still incorrect on Total Store Order (TSO) architectures such as x86, let alone on weaker memory models. Despite stores being totally-ordered, x86 may still reorder stores after subsequent loads [133]. Accordingly, an execution of lines $8, 9a, 9b, 9c$ may be locally visible to another thread as $9a, 8, 9b, 9c$, and hence an apparent interleaving of

$$\textbf{T1}@1, \textbf{T2}@1-5, \textbf{T1}@7, \underline{\textbf{T1}@9a}, \textbf{T3}@1-5, \underline{\textbf{T1}@8}, \textbf{T1}@9b-B$$

is possible (reordered accesses underlined for emphasis). An acquire barrier is needed between lines 8 and 9 to solve this problem on TSO [15] (on x86, either `mfence` or `xchg/xadd`). Recent work [21] also demonstrated unsoundness in a similar lock elision implementation on ARMv8 (PSO), in which the transactional path reads the lock's internal state directly rather than using a separate flag. In Figure 6.2, a release barrier before line A is also necessary under PSO.

Because Landslide's concurrency model includes only instruction-level thread nondeterminism, not per-CPU memory buffer reorderings, its current HTM implementation cannot find this bug. In fact, it erroneously verifies the corresponding test `htm2`(3,1) in 40 CPU-seconds, with 774 interleavings in total, none of which include the above-listed sequence. Recent work has extended DPOR to support TSO and PSO memory nondeterminism [147], as well as proposed formal execution semantics for HTM on these architectures [21, 34]; if both of these advances were incorporated into Landslide's concurrency model, it could find or verify the absence of such bugs. Visual inspection of [29]'s HTM data structures found no barriers used in this implementation pattern; I would urge any reader interested in using those to add them in by hand first. The test case `lock`(`mutex`) (`410user/progs/htm_mutex.c` in the repository) provides an example of how to use compiler intrinsics to emit the necessary barriers.

Chapter 7

Related Work

This field is built of the contributions of many a brilliant mind trying to carve out a presentable space in an overall impossible problem, each making their own tradeoffs along the way. While previous chapters cited prior work as necessary in background discussions, algorithm descriptions, and so on, this chapter aims to comprehensively tour the field, orienting the reader's understanding of Landslide in the space of said tradeoffs.

7.1 Stateless Model Checking

Equal partners in concurrency testing are the practical and the theoretical: the former meaning tool implementations that target specific problem domains and help users as best one can, and the latter meaning algorithmic advances to bring ever-larger state spaces within the realm of computational feasibility. I discuss my most closely related works split in two sections accordingly.

Tools

Systematic concurrency testing dates back to Verisoft [50], the 1997 tool which first attempted to exhaustively explore the possibile ways to interleave threads. Since then, researchers have built many tools along the same lines to test many kinds of programs. One of the best-known stateless model checkers is Microsoft Research's CHESS [102], a checker for userspace C++ programs which preempts on synchronization APIs by default, supporting compiler instrumentation to preempt on memory accesses as well, and which pioneered the ICB search strategy discussed below.

Many checkers exist which target programs written for various different types of concurrent execution and/or programming environments. MaceMC [73], MoDist [142], SAMC [84], ETA [128], and Concuerror [22], focus on distributed systems, where concurrent events are limited to message-passing and may span across multiple machines. R4 [69] and EventRacer [11] check event-driven concurrent programs typical in mobile applications. Like Landslide, SimTester [145] is a Simics [92]-based tool for kernel-level code, although it focuses on interrupt nondeterminism for testing device drivers, and is

limited to injecting at most one interrupt per test run (as if under ICB with a bound of 1). dBug [127], another CMU original similar to CHESS, tests natively-executing programs using a dynamic library preload to insert preemption points at pthread and MPI interface boundaries. Inspect [143] uses a static alias analysis to instrument and preempt all memory accesses to potentially-shared data at compile time, in addition to common synchronization APIs. RacePRO [79] targets multi-process programs using system calls such as the filesystem API as preemption points to find bugs which can corrupt persistent system resources. SPIN [61] tests algorithms defined in the PROMELA domain-specific language, instruments every memory access, uses explicit state tracking rather than the stateless approach (§2.2), and specializes in verifying synchronization primitives such as RCU [97]. TLC [146] checks formal models of concurrent program behaviour written in the specification language TLA+ [82], and is arguably one of the only true concurrency *model checkers* as it checks specifications separate from the programs themselves rather than attempting to exhaustively exercise every thread interleaving directly. Déjà Fu [141] is a model checker for the Haskell language, whose strong type system guarantees that thread communication be confined to trusted, type-safe APIs. It instruments these interfaces (STM among them) to check for deadlocks or nondeterministic behaviour in general, which either may arise despite the static no-data-race guarantee.

The problem of relaxed memory nondeterminism alone has inspired the creation of several new model checkers in the past few years. Relacy [140], a header-only C++ model checking library for checking synchronization primitives, was the first to broach this field, although requires custom annotations for non-atomic memory accesses and does not fully model all possible relaxed memory behaviours. CDSChecker [109] extends DPOR with a *reads-from* relation to capture most of the C++11 memory model's new behaviours. Nidhugg [2] is a checker for TSO and PSO which instruments LLVM abstract assembly, although does not yet support the C++11 memory model. rInspect [147] offers further heuristic state space reduction using buffer bounding (described below). RCMC [76] models a "repaired" version of the C++11 memory model known as RC11 [80], and professes to achieve the best state space reduction to date. These tools each use various heuristics to account for spin-wait loops, ranging from delay bounding [24] to a rigid rewrite rule, and provide only limited support so far for read-modify-write atomics (at best, supporting them by introducing some redundant exploration). No relaxed-memory model checker has yet proposed a satisfactory model for the "thin-air" problem [133], which can cause state space cycles in a way not yet well-understood and remains future work. They also identify all data races (under the C++ definition rather than §4.2.1's) as bugs immediately, rather than checking them for benign or buggy outcomes. All the tools in this paragraph are notably open-source – an encouraging recent trend in the field.

If I might indulge by listing Landslide in its own related work section [14], I would distinguish it by its ability to find shared memory preemption points via dynamic tracing, rather than relying on user annotations or imprecise compiler instrumentation as other tools do. Compared to all other tools I know of, it implements a wider range of exponential explosion coping techniques, some theoretical and some heuristic, some inherited and some novel, to help the user receive meaningful results as promptly as possible. Its choice of a familiar pthread-like synchronization API makes it suitable for inexpert users, and its

recent extension to HTM adds support for more modern concurrency patterns as well.

Algorithms

To date a number of techniques have been proposed to mitigate exponential explosion, the Sisyphean rock of stateless model checking. The notion that some interleavings of concurrent threads could lead to indistinguishable program states and be therefore redundant, known as *partial order reduction* (POR), was first proposed in [48] and explored in detail in [49]. *Dynamic POR* (DPOR) was later developed in [46], proposing to track communication events between threads on-the-fly (i.e., dynamically) rather than to rely on imprecise static alias analyses, and is now considered the baseline for all subsequent state space reduction approaches in the field. That paper includes the *sleep sets* extension, which Landslide includes in its implementation. It is a *sound* reduction algorithm, meaning it will never fail to test a possible program behavior, despite skipping many execution sequences. §3.4.2 provides a detailed walk-through of how DPOR works, as many of this thesis's contributions build directly upon it.

DPOR has since been extended in several ways to achieve further reduction and to incorporate new concurrency models. Optimal DPOR [1] extends sleep sets into the more expressive *wakeup trees*, which provably tests exactly one interleaving from each equivalence class, i.e., the optimal possible reduction, at least under the memory independence definition of equivalence. Extending the equivalence relation itself to capture not just memory *address* conflicts but also the *values* read and written, SATCheck [31] and Maximal Causality Reduction (MCR) [63] use an SMT solver [28] to identify additional pruning opportunities. Implementing wakeup trees or SMT-driven exploration in Landslide is left to future work.

Several other recent advances extend DPOR to new concurrency models, beyond the shared-memory-threading model outlined in §3.4.2. TransDPOR [134] provides extra domain-specific reduction for message-passing actor programs by exploiting the fact that the dependency relation is transitive in the absence of shared state. The R^4 algorithm [69] (corresponding to the R4 checker mentioned above) extends DPOR to event-driven programs by separating the notion of enabled events from that of multiple threads. TaxDC [85], a taxonomy study of distributed systems concurrency bugs, showed that for completeness distributed model checkers must incorporate many forms of nondeterminism, including message reordering, timeouts, network disconnections, and crashes and reboots, in addition to local threads. DPOR for TSO and PSO [147] extends the concurrency model using *shadow threads*, which interleave with traditional threads to represent store buffer nondeterminism, which can expose bugs not even possible in the strong consistency model such as discussed in §6.4. It also introduced a heuristic *buffer bounding* technique, analogous to ICB, to mitigate the corresponding increase in state space size. The same year, Nidhugg [2] proposed a DPOR extension to account for TSO and PSO using *chronological traces*. MCR was recently extended to support relaxed memory models likewise [64]. Just this year, RCMC [76] proposed to replace the interleaving model entirely with *execution graphs*, which precisely model the executions legal under the RC11 memory model, offering further reduction still. Somewhat analogously for HTM, this work's Chapter 6

extended DPOR's concurrency model to include failure injection, and proposed three reduction strategies, one sound and two heuristic, to keep state spaces manageable.

Of course, no matter how optimal a sound reduction, there will always be programs too large to test. To provide even partial results for state spaces that exceed the testing budget (whether as predicted by automatic estimation [129] or by a human's wild guess), various heuristic exploration strategies have been proposed. Preemption Sealing [9] allows programmers to manually exclude preemption points arising from trusted source code modules; Landslide implements this as the `without_function` command (§3.4.1). Iterative Context Bounding (ICB) [101] (§3.4.5) orders the search space by increasing number of preemptions in each branch, which is empirically more likely to expose bugs sooner should they exist; BPOR [24] extends DPOR to preserve soundness thereunder. Landslide implements ICB and BPOR for Chapter 4's control experiments, although does not yet incorporate it into this thesis's own contributions (as discussed in Chapter 8). Chapter 4's Quicksand algorithm is, effectively, another such heuristic search strategy, focusing on preemption point subsets rather than context switch bounding. DeMeter [57] adapted abstraction reduction to distributed systems verification under the name Dynamic Interface Reduction, while dBug [126] applied abstraction reduction to synchronization primitives, and I showed how it could be applied similarly to transactional memory in §6.3.2. Each of these approaches is compatible (and indeed, throughout this thesis used often in concert) with the sound reduction analyses listed above.

7.2 Data race analysis

Data race analysis, originating with the lockset-only analysis of Eraser [120], has since grown into a mature field in its own right, which Landslide more borrows as building blocks for its own methods rather than contributing new techniques to. Race detectors are largely distinguished by their particular flavour of the Happens-Before (HB) relation, as discussed in §2.3.2. Djit+ [116] and FastTrack [45] are among those which soundly avoid false positives using "Pure" HB, tracking Lamport-style vector clocks [81] for each lock and each thread to compute a global partial order on shared state accesses, and flag any access pair not related thereby. FastTrack optimizes Djit+'s analysis rules to remove $O(K)$ runtime factors (i.e., linear in the number of threads) from several common read and write tracing events; however, because K is relatively small in model checking's use cases, Landslide uses the Djit+ rules for the sake of implementation simplicity. Meanwhile, the "hybrid" approach which combines DPOR-style happens-before with locksets [110], used in tools such as ThreadSanitizer [125], compute a more relaxed partial order to find more potential races in a single pass at the cost of false positives. I called this "Limited" HB on account of how it excludes only those access pairs separated by blocking synchronization, not those separated by just locks or barriers, as compared to Pure HB. Landslide's Limited HB implementation piggy-backs on DPOR's computed happens-before relation, supplemented with straightforward lock-sets and heuristic treatment of lock hand-off (often common in kernels).

Since these foundational algorithms, many more recent works have contributed to

make data-race analysis more precise, more performant, and/or more domain-specific. The Causally-Precedes relation [131] is a refinement of Limited HB which avoids the most common cases of false positives, including §4.2.1's reallocation false positives. It could strike a middle ground in the bug-finding/verification tradeoff between Pure and Limited HB (§4.3) that would be a welcome enhancement in Quicksand. IFRit [40] improves the performance of Pure HB using an interference analysis, which could allow future work to avoid tracing every memory access in a simulator such as Bochs [83] or Simics [92]. DroidRacer [93] and CAFA [62] find data races in Android applications, using domain-specific heuristics (orthogonal to Quicksand's method) to reduce false positives. DataCollider [42] finds data races in kernel code by using hardware breakpoints and random sampling to achieve high performance.

Although many checkers listed in the previous section are content to report any data races as outright bugs, RacerX [41] showed that tools must be careful not to overwhelm users with benign warnings they don't care about fixing. This has motivated replay analysis to classify data-race candidates by their impact on program behaviour by extending single-pass data-race analysis to many thread interleavings. It was first introduced in [105], which compares the program states immediately after the access pair for differences, preferring still to err on the side of false positives (as different program states might not necessarily lead to a failure). RaceFuzzer [124] avoids false positives by requiring an actual failure be exhibited, as Quicksand does, although it uses random schedule fuzzing rather than systematic testing for its concurrency coverage. Portend [71] is closest in spirit to Quicksand: it tests alternate executions based on single-pass data-race candidates to classify them in a taxonomy of likely severity, including non-failing races which nevertheless cause nondeterministic output in addition to obvious failures. However, it does not test alternate interleavings in advance of knowing any specific data races, which §4.3 showed is necessary to find certain bugs. Quicksand builds on Portend's approach by introducing a feedback loop between the data-race analysis and model checking, which results in a stronger verification property when the test can be fully completed (§4.2.1). Portend also uses symbolic execution to test input nondeterminism as well as schedule nondeterminism, while Quicksand remains at the mercy of manual test case design. Future work could incorporate Portend's taxonomy to better help the user understand any non-failing data races when the test is too large to complete, as well as its symbolic execution to help user-provided tests achieve better coverage automatically.

7.3 Concurrency in education

The operating systems curriculum at CMU has used the Pebbles project infrastructure and assigned the thread library [36] and kernel [35] projects in something recognizably close to their modern forms since the Fall 2003 semester. I chose Pebbles to target with Landslide because it is closest to home, naturally. To indulge my bias as a former member of 15-410 course staff, I also believe that Pebbles's open-ended, design-oriented project structure is best suited to train students to design robust concurrent code and debug it efficiently, as it forces them to consider interactions between many different parts of their

design simultaneously. However, the difficulty of its concurrency problems (mostly having to do with thread lifecycle) leaves little time left in the semester to cover more modern topics such as multicore scheduling let alone transactions or relaxed memory (all relegated to lecture material not reinforced by the assignments).

Pintos [114] has recently emerged as the most popular educational kernel (by count of top CS schools in the United States who teach by it); it trades off the prevalence of its concurrency challenges to cover various OS topics more broadly, especially advanced scheduling algorithms and filesystems. Pintos is the stand-alone evolution of its predecessor, Nachos [23], which originally ran as a UNIX process with simulated device drivers. Its popularity motivated me to extend Landslide to support it as an additional kernel architecture (an unfortunately arduous task) to prove Landslide's mettle beyond CMU's walls. Xv6 [25], from MIT, is another major educational kernel, which is also UNIX-like and runs in QEMU, and a natural target for model checking in future work. Recently, Columbia introduced a new Android-focused OS course [7], which perhaps highlights the importance of related work on model-checking event-driven applications [69].

To my knowledge, this is the first study of model checking in an educational setting, although teaching concurrency is not itself an unstudied problem. [88] surveyed how students think about testing and debugging during a concurrent programming project, finding that unguided, students often approach testing haphazardly, not understanding the goal of good concurrency coverage, and also had difficulty understanding single failing executions. In fact, the study explicitly recommended tool support for testing many thread interleavings automatically and for execution traces to communicate sequences of important events (preemption traces), which I dare say I have achieved in this thesis. A more recent study [6] examined in detail the students' thought process during the diagnosis and fixing phases, although its participants were drawn from novice-level programming classes, and the experiment was set up with more elementary bugs like syntax and logic errors correspondingly. Nevertheless, the authors recommended teaching debugging skills explicitly via systematic exposure to different kinds of bugs, which suggests future work for even advanced operating systems curricula to offer a "warm up" Landslide assignment (for example, the `atomic_*` tests from §5.1.3) that could ultimately lead to a higher solve rate on Landslide's bug reports during P2 (§5.3.1).

Willgrind [104] is a tool recently developed at Virginia Tech that targets a fork-join parallelism project and checks for memory errors (using the Valgrind [107] framework) as well as deadlocks, assertion failures, and data races, similarly to Landslide, although unlike Landslide, its thread interleaving coverage is as yet limited to stress testing. Its GUI-based debugging output is perhaps more friendly than Landslide's HTML preemption traces, and its user survey found that students appreciated detailed debugging info especially for deadlocks (future work for Landslide), but also that students had little patience for even a 5-minute stress test when no assurance against false negatives could be provided. This suggests motivating students with Landslide's verification guarantee, although it is tricky to avoid accientally encouraging them to limit possible interleavings by just using one global lock for everything, which is counter to 15-410's educational goals.

7.4 Transactional memory

Transactional memory (TM), first introduced in 1993 [60], has received renewed attention in recent years with the announcement of Intel's Haswell architecture [59], which supports hardware transactions (HTM) using new x86 instructions. Since then, many studies have evaluated the increased performance it offers over traditional locking and/or STM [29, 32, 144]. HTM's performance comes at an increased cost in complexity to the programmer, who must avoid system calls or transaction nesting, respect the CPU cache capacity, and consider retry loops for spurious failure. SI-TM [86] introduces techniques for reducing HTM's abort rates for performance's sake, but without eliminating them altogether, any full verification must still consider them possible anywhere. For programmers who wish to avoid such concerns, the simpler STM programming model remains relevant. One recent work [19] enhances STM transactions to nest with HTM ones, while another [55] adds support for relaxed memory models. Meanwhile, two recent papers [21, 34] have proposed formal models of HTM's execution semantics under relaxed memory likewise. Such extensions come with the challenge of even more complicated behavioural semantics for stateless model checking to accurately model and verify in future work.

Testing approaches for transactional programs are sparsely represented in the literature so far. Although several related works [33, 53, 54] are building up to formal proofs of the correctness of underlying TM *implementations*, Landslide is the first I know of to verify client programs thereof. McRT STM [111] uses SPIN [61] to model check an STM implementation up to 2 threads running 1 transaction each with up to 3 memory accesses. This kind of verification, analogous to §5.1.3's `mutex_test`, is an important stepping stone for trusting the results Landslide will provide. STAMP [98] is a benchmark suite transactional programs, implemented using the OpenTM interface [8], used by many papers in the field to evaluate the performance of both STM and HTM implementations alike, although as discussed in §6.3, focuses more on performance than on interesting concurrency properties. Even so, the more recent Stampede suite [108] argues that STAMP's benchmarks were constructed under a programming model poorly-suited to fully take advantage of HTM's performance, and that scalable HTM programs should seek to minimize incidental conflicts and to handle aborts more flexibly than with blind retry loops. The programming complexity needed to achieve these goals calls, of course, for correspondingly advanced verification approaches such as Landslide. Finally, TxRace [148] tests non-transactional programs for data races by inserting HTM calls via compiler instrumentation, relying on conflict aborts to point out access pairs that would be unsafe in the original program. This citation perhaps half belongs in §7.2 as well; I include it here to highlight the importance of Landslide's ability to distinguish different abort reasons (§6.2.2).

An article from relatively early in the timeline of TSX [94] warns of several false equivalence pitfalls when converting conventionally-locking code to use transactions, although these pitfalls depend on multiple existing locks used locally and disjointly, so this does not invalidate the equivalence proved in §6.1.3. Rather, Landslide could be used to ensure that freshly-converted transactional code avoids the warned-of pitfalls. *Learning from Mistakes* [90], a survey of the characteristics of many types of concurrency bugs, found that TM could potentially fix some, but not all, of the studied bugs, while in other cases it must

be combined with other concurrency primitives to be fully correct. A subsequent paper [139] found a majority of bugs in their study to be easily fixable with hand-written transactions, while others remained out of scope due to blocking `cond_wait()` operations and the like; more recently, the tool BugTM [20] aims to deploy such repairs in production code fully automatically. However, these studies all optimize for empirical correctness at best, as well as maintaining good performance, which motivates the use of tools like Landslide to ensure these rewrites are actually correct, rather than merely shrinking the necessary preemption window required to expose them.

7.5 Other concurrency verification approaches

Naturally, many avenues of research towards writing correct programs have been explored apart from just executing them a bunch of times to check all the interleavings. Though not as directly related as the works referenced above, this section explores such approaches, ranging from expressing safety guarantees in a language's type system to checking, proving, and/or enforcing execution properties post-hoc.

Programming language design

While C's extremely rudimentary type system allows the compiler to statically check programs for properties such as not accidentally dereferencing raw integer values as if they were pointers, more advanced programming languages may make guarantees about concurrent execution. Erlang [138], an early concurrent functional language, introduced the actor model for concurrency, in which threads share no state and must communicate only by message-passing. While this statically guarantees the absence of data races, programs may still execute nondeterministically, so concurrency bugs, especially deadlocks, are not ruled out. Concuerror [22], discussed above, is a model checking tool for Erlang programs. Haskell [65] offers a more sophisticated interface to concurrency: threads may reference the same objects and even update shared references using monads that encapsulate mutation, but at the (garbage-collected) execution level all data is immutable once created, which preserves type soundness and data-race freedom. The aforementioned Déjà Fu [141] checks concurrent Haskell programs. Rust [95] presents a type system with more explicit memory management, in-place mutation, and mutable references to appear familiar and approachable to those already versed in C++. It proposes a borrow-check analysis to ensure memory and type safety despite mutable references, and a trait system to ensure no shared state between threads by default. Its concurrency libraries then offer interfaces which relax this restriction, allowing threads even to simultaneously reference shared mutable state, using the type system to enforce sound locking discipline across such accesses, again preserving type soundness and data-race freedom[1]. I know of no existing model checker for Rust as of yet. The Relaxed Memory Calculus [133] proposes to extend C++ with annotations for weak memory atomics, which allows for static

[1]The author themself contributed the original design for this latter feature.

formal analysis of memory access reorderings. Although not ruling out data races, this approach is an important step towards compilers which can statically reason about program execution under more advanced concurrency models. Finally, LVish [78] features a type system that enforces deterministic behaviour by construction, using shared state called LVars which allow writes only in ways that update order is not observable. This renders thread interleavings entirely irrelevant, obviating any need for runtime verification, but at the cost of a more restrictive programming model.

Deterministic multithreading

Coming at nondeterminsm from the opposite angle as this thesis, which aims to push the frontier of testing coverage to expand as many interleavings as possible, is deterministic multithreading, which reduces the number of interleavings possible to begin with enough that said frontier can reach it more easily. Unlike LVish, described above, these systems provide deterministic execution even for the familiar, C-like, shared-state multithreading programming model. Kendo [112] and CoreDet [10] were among the first systems to implement this, but were limited in which sources of nondeterminism they could control and suffered high performance overhead. DThreads [87] then extended the scope of determinization to include data races, while Peregrine [26] improved performance by using record-and-replay to compute a set of possible safe schedules. Parrot [27] later integrated with the aforementioned dBug [127] to offer a partially-determinizing runtime scheduler that offered near-baseline performance by allowing the programmer to manually annotate speed-critical nondeterministic sections and then check the resulting state spaces using dBug as normal. Most recently, Sofritas [30] proposed the Ordering-Free Region execution model which restricts nondeterminism to only order-enforcing operations such as blocking, and automatically suggests refinement annotations to the programmer when that would be too aggressive for the intended behaviour. These determinizing runtimes serve a different purpose than model checking: they seek to preserve the stability of existing code already running in production, whether or not concurrency bugs may exist, while this thesis aims to eradicate as many such bugs as possible beforehand. As Parrot demonstrated, the two approaches are compatible in cases where either extreme be infeasible.

Symbolic execution

Analogous to stateless model checking, which seeks good coverage of possible thread execution paths under schedule nondeterminism, another popular testing approach is symbolic execution [74], which seeks good coverage of possible flow control paths under input nondeterminism. Symbolic executors abstract a program's variables and use constraint solvers such as Z3 [28] to work backwards and synthesize combinations of test inputs which can lead to a failure. KLEE [18], one well-known and open-source implementation, offers over 90% code coverage on average across many tests, often outdoing that achieved by programmers' own hand-written tests. Later, Contessa [77] extended symbolic execution to include concurrency nondeterminism as well, by using a DPOR-

like analysis on individual execution traces then including reordering possibilities in its SMT constraints. This simultaneously exercises both input and schedule nondeterminism, but does not provide the same verification guarantees as repeated DPOR iterations with explicit scheduling. Exploring both kinds of state space at once thoroughly enough to provide strong verification is undoubtedly subject to further state space explosion, and remains future work. Symbiosis [91] starts from the known root cause of an existing failure and uses symbolic execution to synthesize a schedule to reproduce it, then further searches for a non-failing schedule and compares them to produce a minimum sequence of events necessary for the failure. This approach skips the initial verification step entirely, but greatly reduces the diagnosis effort required of the user, which was a common complaint about Landslide's preemption traces.

Kernel verification

seL4 [75] is a microkernel fully designed and specified in Haskell and translated into C. Its proofs guarantee not only standard security properties such as process isolation and bounded interrupt latency, but also that the C code faithfully implements the specification. It addresses concurrency by enabling system interrupts only at carefully-chosen code points, and proving bounded runtime besides to ensure good preemptibility. This degree of verification must however come at a cost: seL4's authors reported over 2 person-years of development effort, with the majority spent on the Haskell specification. CertiKOS [52] extends this approach to include full concurrency and fine-grained locking in the scope of verification, using a proof in Coq that also took 2 person-years to complete. Its safety properties hold under all possible interleavings, and include data-race freedom as well as standard sequential properties such as no null dereference and no buffer or integer overflow, although it stops short of reasoning about relaxed memory orderings or the TLB cache. Many programmers would find a verification cost measured in person-years far too prohibitive, while others might argue that for safety-critical kernel code you can't afford *not* to verify so thoroughly. More recently, Hyperkernel [106] extended the xv6 educational kernel [25] to allow for partial, case-by-case verification of system call behaviour using state-machine specification in Python checked by an SMT solver. To limit verification complexity, it assumes not only uniprocessor execution but also that interrupts be perpetually disabled, taking concurrency entirely out of the equation to allow for greater extensibility and lessen the programmer's verification burden. Heroic as such end-to-end formal verification projects are, this thesis finds that trading off thoroughness for accessibility is also acceptable if it means helping more users overall. Future work could check preemptive and/or multiprocessor kernels by first checking safety properties in the absence of concurrency, then checking with Landslide that concurrency introduces no new program behaviour not already verified, to provide a less formal, but still hopefully useful, verification guarantee.

Chapter 8

Future Work

Chapter 9

Conclusion

Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535845. URL http://doi.acm.org/10.1145/2535838.2535845. 2.2.2, 3.4.2, 3.4.2, 6.1.2, 6.2.4, 6.3.2, 7.1

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In *Proceedings of the 21st International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9035*, pages 353–367, Berlin, Heidelberg, 2015. Springer-Verlag. ISBN 978-3-662-46680-3. doi: 10.1007/978-3-662-46681-0_28. URL https://doi.org/10.1007/978-3-662-46681-0_28. 7.1, 7.1

[3] Michael J. Accetta, Robert V. Baron, William J. Bolosky, David B. Golub, Richard F. Rashid, Avadis Tevanian, and Michael Young. Mach: A new kernel foundation for unix development. In *USENIX Summer*, pages 93–113, 1986. 2.4.1

[4] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133985. URL http://doi.acm.org/10.1145/1133981.1133985. 2.1.4, 6

[5] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996. ISSN 0018-9162. doi: 10.1109/2.546611. URL http://dx.doi.org/10.1109/2.546611. 2.1.1, 6.4

[6] Basma S. Alqadi and Jonathan I. Maletic. An empirical study of debugging patterns among novices programmers. In *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '17, pages 15–20, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4698-6. doi: 10.1145/3017680.3017761. URL http://doi.acm.org/10.1145/3017680.3017761. 7.3

[7] Jeremy Andrus and Jason Nieh. Teaching operating systems using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 613–618, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-

1098-7. doi: 10.1145/2157136.2157312. URL http://doi.acm.org/10.1145/2157136.2157312. 7.3

[8] Woongki Baek, Chi Cao Minh, Martin Trautmann, Christos Kozyrakis, and Kunle Olukotun. The OpenTM transactional application programming interface. In *Proceedings of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, pages 376–387, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2944-5. doi: 10.1109/PACT.2007.74. URL https://doi.org/10.1109/PACT.2007.74. 6.3, 7.4

[9] Thomas Ball, Sebastian Burckhardt, Katherine E. Coons, Madanlal Musuvathi, and Shaz Qadeer. Preemption sealing for efficient concurrency testing. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 420–434, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12001-6, 978-3-642-12001-5. doi: 10.1007/978-3-642-12002-2_35. URL http://dx.doi.org/10.1007/978-3-642-12002-2_35. 2.2.2, 3.4.1, 7.1

[10] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. Coredet: A compiler and runtime system for deterministic multithreaded execution. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736029. URL http://doi.acm.org/10.1145/1736020.1736029. 7.5

[11] Pavol Bielik, Veselin Raychev, and Martin Vechev. Scalable race detection for Android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814303. URL http://doi.acm.org/10.1145/2814270.2814303. 7.1

[12] Ben Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2012. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf. 1, 3.1.3, 7, 8, 3.4.1, 5, 5.4.1

[13] Ben Blum. Transactional memory concurrency verification with Landslide. In *Proceedings of the 12th ACH SIGBOVIK Conference in Celebration of Harry Q. Bovik's 2^6th Birthday*, SIGBOVIK '18, pages 143–151, Pittsburgh, PA, USA, 2018. ACH. URL http://sigbovik.org/2018/proceedings.pdf. 6.2.2, 6.3

[14] Ben Blum. *Practical Concurrency Testing, or: How I Learned to Stop Worrying and Love the Exponential Explosion*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2018. 7.1

[15] Ben Blum. Demonstration of the need for a barrier in TSX failure paths. https://gist.github.com/bblum/85f64858a35a74641be228f191144911, 2018. 6.4

[16] Ben Blum and Garth Gibson. Stateless model checking with data-race preemption

points. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 477–493, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984036. URL http://doi.acm.org/10.1145/2983990.2984036. 6.1.2, 6.1.3

[17] R. Bryant and D. O'Hallaron. Introducing computer systems from a programmer's perspective. In *Proc. of the 32nd Technical Symposium on Computer Science Education (SIGCSE)*, Charlotte, NC, February 2001. ACM. 2.4.1

[18] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855756. 7.5

[19] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid STM/HTM for nested transactions on OpenJDK. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 660–676, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984029. URL http://doi.acm.org/10.1145/2983990.2984029. 6.4, 7.4

[20] Yuxi Chen, Shu Wang, Shan Lu, and Karthikeyan Sankaralingam. Applying hardware transactional memory for concurrency-bug failure recovery in production runs. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 837–850, 2018. ISBN 978-1-931971-44-7. 7.4

[21] Nathan Chong, Tyler Sorensen, and John Wickerson. The semantics of transactions and weak memory in x86, Power, ARM, and C++. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2018, pages 211–225, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-5698-5. doi: 10.1145/3192366.3192373. URL http://doi.acm.org/10.1145/3192366.3192373. 6.1.2, 6.4, 7.4

[22] Maria Christakis, Alkis Gotovos, and Konstantinos Sagonas. Systematic testing for detecting concurrency errors in Erlang programs. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, ICST '13, pages 154–163, Washington, DC, USA, 2013. IEEE Computer Society. ISBN 978-0-7695-4968-2. doi: 10.1109/ICST.2013.50. URL https://doi.org/10.1109/ICST.2013.50. 7.1, 7.5

[23] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The Nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 4–4, Berkeley, CA, USA, 1993. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267303.1267307. 7.3

[24] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 833–848, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509556. URL http://doi.acm.org/10.1145/2509136.2509556. 5, 6.1.2, 7.1, 7.1

[25] Russ Cox, Cliff Frey, Xiao Yu, Nickolai Zeldovich, and Austin Clements. Xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2012/xv6.html. 7.3, 7.5

[26] Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. Efficient deterministic multithreading through schedule relaxation. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 337–351, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043588. URL http://doi.acm.org/10.1145/2043556.2043588. 7.5

[27] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522735. URL http://doi.acm.org/10.1145/2517349.2522735. 7.5

[28] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792766. 7.1, 7.5

[29] Mario Dehesa-Azuara and Nick Stanley. Hardware transactional memory with Intel's TSX. http://www.contrib.andrew.cmu.edu/~mdehesaa/, 2016. 6.1.1, 6.3, 6.3.2, 6.4, 7.4

[30] Christian DeLozier, Ariel Eizenberg, Brandon Lucia, and Joseph Devietti. SOFRITAS: Serializable ordering-free regions for increasing thread atomicity scalably. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 286–300, New York, NY, USA, 2018. ACM. ISBN 978-1-4503-4911-6. doi: 10.1145/3173162.3173192. URL http://doi.acm.org/10.1145/3173162.3173192. 7.5

[31] Brian Demsky and Patrick Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 20–36, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814297. URL http://doi.acm.org/10.1145/2814270.

2814297. 2.2.2, 6.1.2, 7.1

[32] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508263. URL http://doi.acm.org/10.1145/1508244.1508263. 3, 7.4

[33] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259:245–261, December 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2010.01.001. URL http://dx.doi.org/10.1016/j.entcs.2010.01.001. 6.3, 7.4

[34] Brijesh Dongol, Radha Jagadeesan, and James Riely. Transactions in relaxed memory architectures. *Proceedings of the 45th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2(POPL):18:1–18:29, December 2017. ISSN 2475-1421. doi: 10.1145/3158106. URL http://doi.acm.org/10.1145/3158106. 6.4, 6.4, 7.4

[35] David Eckhardt. Pebbles kernel specification. http://www.cs.cmu.edu/~410-f16/p2/kspec.pdf, 2016. 2, 2.4.1, 2.4.1, 7.3

[36] David Eckhardt. Project 2: User level thread library. http://www.cs.cmu.edu/~410-f16/p2/thr_lib.pdf, 2016. 2, 2.1.3, 2.4.1, 2.4.1, 7.3

[37] David Eckhardt. Personal communication, 2018. 2.4.1

[38] David Eckhardt. Paradise lost. http://www.cs.cmu.edu/~410-s18/lectures/L13a_Lost.pdf, 2018. 5.1.1

[39] David Eckhardt. Synchronization (2). http://www.cs.cmu.edu/~410-s18/lectures/L08b_Synch.pdf, 2018. 5.1.2

[40] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384650. URL http://doi.acm.org/10.1145/2384616.2384650. 7.2

[41] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945468. URL http://doi.acm.org/10.1145/945445.945468. 2.2.1, 2.3.1, 5.3.1, 7.2

[42] John Erickson, Madanlal Musuvathi, Sebastian Burckhardt, and Kirk Olynyk. Effective data-race detection for the kernel. In *Proceedings of the 9th USENIX conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–16, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?

id=1924943.1924954. 7.2

[43] Eric Feigelson and G. Jogesh Babu. Beware the Kolmogorov-Smirnov test! https://asaip.psu.edu/Articles/beware-the-kolmogorov-smirnov-test. 6

[44] R. A. Fisher. On the interpretation of χ^2 from contingency tables, and the calculation of P. *Journal of the Royal Statistical Society*, 85(1):87–94, 1922. 5.3.2

[45] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542490. URL http://doi.acm.org/10.1145/1542476.1542490. 3.4.4, 6.1.3, 7.2

[46] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040315. URL http://doi.acm.org/10.1145/1040305.1040315. 2.2.2, 3.4.2, 3.4.2, 3.4.2, 3.4.2, 6.1.2, 6.2.4, 7.1

[47] The GNU Foundation. X86 transaction memory intrinsics. https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html, 2016. 2.1.4, 6, 6.2.2

[48] P. Godefroid and P. Wolper. A partial approach to model checking. In *Papers Presented at the IEEE Symposium on Logic in Computer Science*, pages 305–326, Orlando, FL, USA, 1994. Academic Press, Inc. doi: 10.1006/inco.1994.1035. URL http://dx.doi.org/10.1006/inco.1994.1035. 7.1

[49] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg, 1996. ISBN 3540607617. 3.4.2, 3.4.2, 6.2.4, 7.1

[50] Patrice Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Proceedings of the 9th International Conference on Computer Aided Verification*, CAV '97, pages 476–479, London, UK, UK, 1997. Springer-Verlag. ISBN 3-540-63166-6. URL http://dl.acm.org/citation.cfm?id=647766.733607. 1, 2.2, 7.1

[51] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. 2.2.1

[52] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, pages 653–669, Berkeley, CA, USA, 2016. USENIX Association. ISBN 978-1-931971-33-1. URL http://dl.acm.org/citation.cfm?id=3026877.3026928. 7.5

[53] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-795-7. doi: 10.1145/1345206.1345233. URL http://doi.acm.org/10.1145/1345206.1345233. 6.3, 7.4

[54] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Completeness and nondeterminism in model checking transactional memories. In *Proceedings of the 19th International Conference on Concurrency Theory*, CONCUR '08, pages 21–35, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85360-2. doi: 10.1007/978-3-540-85361-9_6. URL http://dx.doi.org/10.1007/978-3-540-85361-9_6. 6.3, 7.4

[55] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 321–336, Berlin, Heidelberg, 2009. Springer-Verlag. ISBN 978-3-642-02657-7. doi: 10.1007/978-3-642-02658-4_26. URL http://dx.doi.org/10.1007/978-3-642-02658-4_26. 7.4

[56] Haryadi Gunawi. Cmsc 23000 (cs 230): Operating systems (autumn 2014). https://www.classes.cs.uchicago.edu/archive/2014/fall/23000-1/, 2014. 2.4.2

[57] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043582. URL http://doi.acm.org/10.1145/2043556.2043582. 7.1

[58] Mike Hachman. Intel finds specialized TSX enterprise bug on Haswell, Broadwell CPUs. *PCWorld*, 2014. URL https://www.pcworld.com/article/2464880/intel-finds-specialized-tsx-enterprise-bug-on-haswell-broadwell-cpus.html. 6

[59] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, (2): 6–20, 2014. 3, 2.1.4, 6, 7.4

[60] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164. 2.1.4, 7.4

[61] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. ISSN 0098-5589. doi: 10.1109/32.588521. URL http://dx.doi.org/10.1109/32.588521. 2.2.1, 7.1, 7.4

[62] Chun-Hung Hsiao, Jie Yu, Satish Narayanasamy, Ziyun Kong, Cristiano L. Pereira,

Gilles A. Pokam, Peter M. Chen, and Jason Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594330. URL http://doi.acm.org/10.1145/2594291.2594330. 7.2

[63] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 165–174, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737975. URL http://doi.acm.org/10.1145/2737924.2737975. 2.2.2, 3.4.2, 6.1.2, 7.1

[64] Shiyou Huang and Jeff Huang. Maximal causality reduction for TSO and PSO. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 447–461, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984025. URL http://doi.acm.org/10.1145/2983990.2984025. 7.1

[65] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M. Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, Dick Kieburtz, Rishiyur Nikhil, Will Partain, and John Peterson. Report on the programming language haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Not.*, 27(5):1–164, May 1992. ISSN 0362-1340. doi: 10.1145/130697.130699. URL http://doi.acm.org/10.1145/130697.130699. 7.5

[66] Intel. Hardware lock elision overview. https://software.intel.com/en-us/node/683688, 2013. 6, 6.3.1

[67] Intel. 6th generation Intel processor family specification update. https://www3.intel.com/content/dam/www/public/us/en/documents/specification-updates/desktop-6th-gen-core-family-spec-update.pdf, 2017. 6

[68] Intel. Transactional synchronization extensions (TSX) overview. https://software.intel.com/en-us/node/524022, 2018. 2.1.4, 6, 6.1.2

[69] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814282. URL http://doi.acm.org/10.1145/2814270.2814282. 7.1, 7.1, 7.3

[70] Anthony Joseph. CS162: Operating systems and systems programming. https://cs162.eecs.berkeley.edu/, 2016. 2.4.2

[71] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012.

ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150997. URL http://doi.acm.org/10.1145/2150976.2150997. 7.2

[72] Philip Kilby, John Slaney, Sylvie Thiébaux, and Toby Walsh. Estimating search tree size. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1014–1019. AAAI Press, 2006. ISBN 978-1-57735-281-5. URL http://dl.acm.org/citation.cfm?id=1597348.1597350. 3.4.3, 25

[73] Charles Killian, James W. Anderson, Ranjit Jhala, and Amin Vahdat. Life, death, and the critical transition: finding liveness bugs in systems code. In *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1973430.1973448. 7.1

[74] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7): 385–394, July 1976. ISSN 0001-0782. doi: 10.1145/360248.360252. URL http://doi.acm.org/10.1145/360248.360252. 7.5

[75] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, SOSP '09, pages 207–220, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-752-3. doi: 10.1145/1629575.1629596. URL http://doi.acm.org/10.1145/1629575.1629596. 7.5

[76] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL):17:1–17:32, December 2017. ISSN 2475-1421. doi: 10.1145/3158105. URL http://doi.acm.org/10.1145/3158105. 6.1.2, 7.1, 7.1

[77] Sudipta Kundu, Malay K. Ganai, and Chao Wang. CONTESSA: Concurrency testing augmented with symbolic analysis. In *Proceedings of the 22Nd International Conference on Computer Aided Verification*, CAV'10, pages 127–131, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-14294-X, 978-3-642-14294-9. doi: 10.1007/978-3-642-14295-6_13. URL http://dx.doi.org/10.1007/978-3-642-14295-6_13. 7.5

[78] Lindsey Kuper, Aaron Todd, Sam Tobin-Hochstadt, and Ryan R. Newton. Taming the parallel effect zoo: Extensible deterministic parallelism with LVish. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 2–14, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594312. URL http://doi.acm.org/10.1145/2594291.2594312. 7.5

[79] Oren Laadan, Nicolas Viennot, Chia-Che Tsai, Chris Blinn, Junfeng Yang, and Jason Nieh. Pervasive detection of process races in deployed systems. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 353–367, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.

1145/2043556.2043589. URL http://doi.acm.org/10.1145/2043556.2043589. 7.1

[80] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, pages 618–632, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4988-8. doi: 10.1145/3062341.3062352. URL http://doi.acm.org/10.1145/3062341.3062352. 7.1

[81] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563. 2.3.2, 3.4.2, 3.4.4, 7.2

[82] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. ISBN 032114306X. 7.1

[83] Kevin P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux J.*, 1996(29es), September 1996. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=326350.326357. 3, 7.2

[84] Tanakorn Leesatapornwongsa, Mingzhe Hao, Pallavi Joshi, Jeffrey F. Lukman, and Haryadi S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 399–414, Berkeley, CA, USA, 2014. USENIX Association. ISBN 978-1-931971-16-4. URL http://dl.acm.org/citation.cfm?id=2685048.2685080. 7.1

[85] Tanakorn Leesatapornwongsa, Jeffrey F. Lukman, Shan Lu, and Haryadi S. Gunawi. TaxDC: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 517–530, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872374. URL http://doi.acm.org/10.1145/2872362.2872374. 7.1

[86] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 383–398, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541952. URL http://doi.acm.org/10.1145/2541940.2541952. 7.4

[87] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: efficient deterministic multithreading. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0977-6. doi: 10.1145/2043556.2043587. URL

http://doi.acm.org/10.1145/2043556.2043587. 7.5

[88] Jan Lönnberg, Anders Berglund, and Lauri Malmi. How students develop concurrent programs. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 129–138, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc. ISBN 978-1-920682-76-7. URL http://dl.acm.org/citation.cfm?id=1862712.1862732. 7.3

[89] Chris Lu. unaligned writes are bad. https://gist.github.com/kalenedrael/323bab7e624f88d0edde, 2014. 2.1.1

[90] Shan Lu, Soyeon Park, Eunsoo Seo, and Yuanyuan Zhou. Learning from mistakes: A comprehensive study on real world concurrency bug characteristics. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 329–339, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-958-6. doi: 10.1145/1346281.1346323. URL http://doi.acm.org/10.1145/1346281.1346323. 7.4

[91] Nuno Machado, Brandon Lucia, and Luís Rodrigues. Concurrency debugging with differential schedule projections. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 586–595, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737973. URL http://doi.acm.org/10.1145/2737924.2737973. 7.5

[92] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002. ISSN 0018-9162. doi: 10.1109/2.982916. URL http://dx.doi.org/10.1109/2.982916. 3, 7.1, 7.2

[93] Pallavi Maiya, Aditya Kanade, and Rupak Majumdar. Race detection for Android applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 316–325, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2784-8. doi: 10.1145/2594291.2594311. URL http://doi.acm.org/10.1145/2594291.2594311. 7.2

[94] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17–17, July 2006. ISSN 1556-6056. doi: 10.1109/L-CA.2006.18. URL https://doi.org/10.1109/L-CA.2006.18. 7.4

[95] Nicholas D. Matsakis and Felix S. Klock, II. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, pages 103–104, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3217-0. doi: 10.1145/2663171.2663188. URL http://doi.acm.org/10.1145/2663171.2663188. 2.1.1, 7.5

[96] A Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*, pages 279–324, New

York, NY, USA, 1987. Springer-Verlag New York, Inc. ISBN 0-387-17906-2. URL http://dl.acm.org/citation.cfm?id=25542.25553. 2.2.2

[97] Paul McKenney and Jonathan Walpole. What is RCU, fundamentally? https://lwn.net/Articles/262464/, 2007. 7.1

[98] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proceedings of the IEEE Workload Characterization Symposium*, IISWC '08, pages 35–46. IEEE Computer Society, 2008. ISBN 978-1-4244-2778-9. doi: 10.1109/IISWC.2008.4636089. URL http://csl.stanford.edu/~christos/publications/2008.stamp.iiswc.pdf. 6.3, 7.4

[99] Gordon E. Moore. Readings in computer architecture. chapter Cramming More Components Onto Integrated Circuits, pages 56–59. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2000. ISBN 1-55860-539-8. URL http://dl.acm.org/citation.cfm?id=333067.333074. 1

[100] Randall Munroe. Significance. https://xkcd.com/882/. 5.3.2

[101] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250785. URL http://doi.acm.org/10.1145/1250734.1250785. 2.2.2, 3.4.5, 5, 6.3.1, 7.1

[102] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855760. 5.4.3, 7.1

[103] Brad A. Myers. The importance of percent-done progress indicators for computer-human interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '85, pages 11–17, New York, NY, USA, 1985. ACM. ISBN 0-89791-149-0. doi: 10.1145/317456.317459. URL http://doi.acm.org/10.1145/317456.317459. 3.4.3

[104] William Naciri. Bug finding methods for multithreaded student programming projects. Master's thesis, Virginia Tech University, June 2017. URL https://vtechworks.lib.vt.edu/bitstream/handle/10919/78675/Naciri_WM_T_2017.pdf. 7.3

[105] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250738. URL

http://doi.acm.org/10.1145/1250734.1250738. 7.2

[106] Luke Nelson, Helgi Sigurbjarnarson, Kaiyuan Zhang, Dylan Johnson, James Bornholt, Emina Torlak, and Xi Wang. Hyperkernel: Push-button verification of an OS kernel. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 252–269, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-5085-3. doi: 10.1145/3132747.3132748. URL http://doi.acm.org/10.1145/3132747.3132748. 7.5

[107] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250746. URL http://doi.acm.org/10.1145/1250734.1250746. 7.3

[108] Donald Nguyen and Keshav Pingali. What scalable programs need from transactional memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '17, pages 105–118, New York, NY, USA, 2017. ACM. ISBN 978-1-4503-4465-4. doi: 10.1145/3037697.3037750. URL http://doi.acm.org/10.1145/3037697.3037750. 7.4

[109] Brian Norris and Brian Demsky. CDSchecker: Checking concurrent data structures written with C/C++ atomics. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 131–150, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509514. URL http://doi.acm.org/10.1145/2509136.2509514. 7.1

[110] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM. ISBN 1-58113-588-2. doi: 10.1145/781498.781528. URL http://doi.acm.org/10.1145/781498.781528. 2.3.2, 6.1.3, 7.2

[111] John O'Leary, Bratin Saha, and Mark R. Tuttle. Model checking transactional memory with Spin. In *Proceedings of the Twenty-seventh ACM Symposium on Principles of Distributed Computing*, PODC '08, pages 424–424, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-989-0. doi: 10.1145/1400751.1400816. URL http://doi.acm.org/10.1145/1400751.1400816. 6.3, 7.4

[112] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, pages 97–108, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508256. URL http://doi.acm.org/10.1145/1508244.1508256. 7.5

[113] John Ousterhout. CS 140: Operating systems (winter 2016). http://web.

stanford.edu/~ouster/cgi-bin/cs140-winter16/index.php, 2016. 2.4.2

[114] Ben Pfaff, Anthony Romano, and Godmar Back. The Pintos instructional operating system kernel. In *Proceedings of the 40th ACM Technical Symposium on Computer Science Education*, SIGCSE '09, pages 453–457, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-183-5. doi: 10.1145/1508865.1509023. URL http://doi.acm.org/10.1145/1508865.1509023. 2, 2.4.2, 7.3

[115] Rob Pike, Dave Presotto, Ken Thompson, and Howard Trickey. Plan 9 from Bell Labs. In *In Proceedings of the Summer 1990 UKUUG Conference*, pages 1–9, 1990. 2.4.1

[116] Eli Pozniansky and Assaf Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Principles and Practice of Parallel Programming*, PPoPP '03, pages 179–190. ACM, 2003. 6.1.3, 7.2

[117] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. URL https://www.R-project.org/. 5.3.2, 5.3.2

[118] Ravi Rajwar and James R. Goodman. Speculative lock elision: Enabling highly concurrent multithreaded execution. In *Proceedings of the 34th Annual ACM/IEEE International Symposium on Microarchitecture*, MICRO 34, pages 294–305, Washington, DC, USA, 2001. IEEE Computer Society. ISBN 0-7695-1369-7. URL http://dl.acm.org/citation.cfm?id=563998.564036. 6

[119] Susmit Sarkar, Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Tom Ridge, Thomas Braibant, Magnus O. Myreen, and Jade Alglave. The semantics of x86-CC multiprocessor machine code. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 379–391, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-379-2. doi: 10.1145/1480881.1480929. URL http://doi.acm.org/10.1145/1480881.1480929. 6.1.2

[120] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997. ISSN 0734-2071. doi: 10.1145/265924.265927. URL http://doi.acm.org/10.1145/265924.265927. 2.3.1, 7.2

[121] F. W. Scholz and M. A. Stephens. K-sample Anderson–Darling tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987. doi: 10.1080/01621459.1987.10478517. URL https://doi.org/10.1080/01621459.1987.10478517. 5.3.2

[122] Fritz Scholz and Angie Zhu. *kSamples: K-Sample Rank Tests and their Combinations*, 2018. URL https://CRAN.R-project.org/package=kSamples. R package version 1.2-8. 5.3.2

[123] Austin Seipp. Transactional memory on Haswell. https://gist.github.com/thoughtpolice/7123036, 2013. 6.3, 6.3.1, 6.3.2

[124] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375584. URL http://doi.acm.org/10.1145/1375581.1375584. 7.2

[125] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. doi: 10.1145/1791194.1791203. URL http://doi.acm.org/10.1145/1791194.1791203. 1, 2.3.1, 2.3.2, 6.1.3, 7.2

[126] Jiri Simsa. *Systematic and Scalable Testing of Concurrent Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2013. (document), 6.3.1, 6.3.2, 6.4, 7.1

[127] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1929004.1929007. 3.3.2, 3.3.5, 5.4.3, 7.1, 7.5

[128] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Efficient Exploratory Testing of Concurrent Systems. Technical Report CMU-PDL-11-113, Carnegie Mellon University, November 2011. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-11-113.pdf. 7.1

[129] Jiri Simsa, Randy Bryant, and Garth Gibson. Runtime estimation and resource allocation for concurrency testing. Technical Report CMU-PDL-12-113, Carnegie Mellon University, December 2012. URL http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-12-113.pdf. 1, 3.4.3, 3.4.3, 25, 26, 27, 7.1

[130] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Concurrent systematic testing at scale. Technical Report CMU-PDL-12-101, Carnegie Mellon University, May 2012. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-101.pdf. 2.2.2

[131] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103702. URL http://doi.acm.org/10.1145/2103656.2103702. 2.3.2, 7.2

[132] Michael J. Sullivan. pebwine – the worst thing I have ever done. Unpublished (implementation private to CMU 15-410 course staff), 2017. 2.4.1

[133] Michael J. Sullivan. *Low-level Concurrent Programming Using the Relaxed Memory Calculus*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2017. 6.4, 7.1, 7.5

[134] Samira Tasharofi, Rajesh K. Karmani, Steven Lauterburg, Axel Legay, Darko Marinov, and Gul Agha. TransDPOR: A novel dynamic partial-order reduction technique for testing actor programs. In *Proceedings of the 14th Joint IFIP WG 6.1 International Conference and Proceedings of the 32nd IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems*, FMOODS'12/FORTE'12, pages 219–234, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-30792-8. doi: 10.1007/978-3-642-30793-5_14. URL http://dx.doi.org/10.1007/978-3-642-30793-5_14. 7.1

[135] Paul Thomson, Alastair F. Donaldson, and Adam Betts. Concurrency testing using schedule bounding: An empirical study. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 15–28, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2656-8. doi: 10.1145/2555243.2555260. URL http://doi.acm.org/10.1145/2555243.2555260. 5

[136] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 1937. doi: 10.1112/plms/s2-42.1.230. URL http://plms.oxfordjournals.org/content/s2-42/1/230.short. 2.1.2

[137] vamsikc. leveldb-tsx. https://github.com/vamsikc/leveldb-tsx/, 2014. Spinlock implementation found at ext/xsync/include/spinlock-rtm.hpp. 6.3.1

[138] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2Nd Ed.)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. ISBN 0-13-508301-X. 7.5

[139] Haris Volos, Andres Jaan Tack, Michael M. Swift, and Shan Lu. Applying transactional memory to concurrency bugs. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 211–222, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150999. URL http://doi.acm.org/10.1145/2150976.2150999. 7.4

[140] Dmitry Vyukov. Relacy race detector. https://github.com/dvyukov/relacy, 2011. 7.1

[141] Michael Walker and Colin Runciman. Déjà fu: A concurrency testing library for haskell. In *Proceedings of the 2015 ACM SIGPLAN Symposium on Haskell*, Haskell '15, pages 141–152, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3808-0. doi: 10.1145/2804302.2804306. URL http://doi.acm.org/10.1145/2804302.2804306. 5, 7.1, 7.5

[142] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558992. 7.1

[143] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 288–305, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85113-4. doi: 10.1007/978-3-540-85114-1_20. URL http://dx.doi.org/10.1007/978-3-540-85114-1_20. 7.1

[144] Richard M Yoo, Christopher J Hughes, Konrad Lai, and Ravi Rajwar. Performance evaluation of Intel® transactional synchronization extensions for high-performance computing. In *2013 SC-International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11. IEEE, 2013. 7.4

[145] Tingting Yu, Witawas Srisa-an, and Gregg Rothermel. SimTester: a controllable and observable testing framework for embedded systems. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, VEE '12, pages 51–62, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1176-2. doi: 10.1145/2151024.2151034. URL http://doi.acm.org/10.1145/2151024.2151034. 7.1

[146] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking TLA+ specifications. In *Proceedings of the 10th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, CHARME '99, pages 54–66, London, UK, UK, 1999. Springer-Verlag. ISBN 3-540-66559-5. URL http://dl.acm.org/citation.cfm?id=646704.702012. 7.1

[147] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 250–259, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737956. URL http://doi.acm.org/10.1145/2737924.2737956. 2.2.2, 6.1.2, 6.3.2, 6.4, 7.1, 7.1

[148] Tong Zhang, Dongyoon Lee, and Changhee Jung. TxRace: Efficient data race detection using commodity hardware transactional memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '16, pages 159–173, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4091-5. doi: 10.1145/2872362.2872384. URL http://doi.acm.org/10.1145/2872362.2872384. 7.4