Practical Concurrency Testing

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

Ben Blum

CMU-CS-18-128

December 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.

Copyright © 2018 Ben Blum

This research was sponsored by the U.S. Army Research Office under grant number W911NF0910273 and by Intel ISTC-CC. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government, or any other entity.

Keywords: concurrency, testing, debugging, verification, model checking, data races, education, transactional memory

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 of reproducing and reasoning about bugs. Stateless model checking is a 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 honor here the many who supported me during these 7 long years and made this journey possible.

Thesis

Professor Garth Gibson is largely responsible for shaping me into the researcher I am today. Despite having primary research interests in other fields, Garth enthusiastically took on the project, and always pushed me to explore new problem domains, to design scientifically thorough experiments, to explain difficult concepts approachably, to appreciate related work as charitably as possible, and to seek guidance from industry people with relevant experience, despite my often stubborn refusal to leave my own little comfort zone. So, despite the times I needed to send 10+ email reminders to get your attention: truly, thanks for everything, Garth.

Professor David A. Eckhardt provided unending guidance on education, writing, career, and life in general, helped revise each semester's recruiting lecture with a keen eye for the student mind, contributed several clever survey questions, and supplied constant encouragement about Landslide's value to 15-410, not to mention the red-ink-encrusted printed draft of this document I received immediately after my defense. My debt of thanks to Dave is of such magnitude that I may never hope to pay it back, but perhaps forward instead.

Jiří Šimša, another former student of Garth's, gave invaluable support during my first few years. Jiří's mentorship oriented me in the world of research, from teaching me how DPOR works to helping revise my paper drafts to offering the kind of candid career advice you can't get from a professor. Had I not met Jiří and learned about dBug during grad school applications way back, none of this would have happened.

Many have collaborated with me directly on the research itself, whom I thank here in approximate chapter order. Michael J. Sullivan and Nathaniel Wesley Filardo were Landslide's first other users besides myself back in its prototype days. Joshua Wise, Miriam Zimmerman, and Glenn Willen helped revise my explanation of DPOR in Chapter 3. Professor Brandon Lucia provided valuable revisions to Chapter 4's content that ultimately led to its publication at OOPSLA 2016. Michael J. Sullivan double-checked the proofs therein. Professor Timothy Zhu from Penn State University and Professor Haryadi Gunawi and TA Kevin Zhao from the University of Chicago graciously allowed me to use their students as guinea pigs and helped to improve Landslide's stability and robustness for use beyond CMU's walls, greatly enriching Chapter 5's evaluation. David Blum and Skye Toor provided invaluable advice on statistical analysis that helped me evaluate the student grade distributions therein. Mario Dehesa-Azuara generously contributed the transactional

memory data structures that make up about half of Chapter 6's evaluation suite. Carlo Angiuli, Evan Cavallo, Ziv Scully, Jim McCann, Matthew Maurer, Stefan Muller, Guillaume Didier, Sol Boucher, and Michael J. Sullivan helped me rehearse and polish the snake-fight talk.

Most importantly of all, I thank every student who ever used Landslide. Though the introduction section may say differently, their smiles and gratitude were the true motivation of this thesis.

Academia

The community of 15-410 TAs has nurtured my love of teaching since well before I took the class myself. 15-410 could not be the world's best OS class without course staff's unwavering dedication to thorough grading, the Socratic method, and empathy to struggling students. It was a blessing to serve, and later to do research, among you all. Thanks especially to Stanley Zhang, the 2018 TA who who worked with me to ensure 15-410 could keep grading projects with Landslide even after I leave. Thanks to Wind River for generously providing Simics educational licenses. Thanks also to Adam Blank for founding and organizing the class Great Practical Ideas, which teaches oft-neglected programming skills such as version control and debugging, which I had the opportunity to teach as head instructor one semester.

I thank the Parallel Data Lab, my research group at CMU, for supporting me at the PDL retreat every fall, pushing me to refine my presentation skills and to network with strangers (now familiar faces) from industry. I also thank the industry attendees for their interest and enlightening conversations. Joan Digney and Karen Lindenfelser make a lot of gears turn behind the scenes at PDL, and Deb Cavlovich and Catherine Copetas do likewise for CSD as a whole. Thanks to Nicholas D. Matsakis for the mentorship during my two summers as a Rust research intern at Mozilla, and to the rest of the Rust community as well for bringing better concurrent programs into the world. And to everyone at large who insisted how cool it was that my research helps students: thank you for helping me keep my head up during darker times.

The ThursDz Council, my group of now-former grad student, Shadyside-bound (in spirit if not in person) restaurant-hoppers, surrounded me with both friendship and mentorship over the years. In particular, Chris Martens, jcreed, Jim McCann, Rob Simmons, Tom 7, and William "wjl" Lovas: you each have been role models to me far more than you know.

Without SIGBOVIK's unique brand of humorous, self-aware, and *definitely legitimate* research, I probably would not have been interested in grad school in the first place, or perhaps would have grown too frustrated along the way for lack of a cathartic outlet for conference paper woes. I thank all who've worn the mantle of Harry Bovik over the years, from the conference's original founders, to every paper author, to all future organizers who'll keep those "mainstream" conferences on their toes well into the future.

Life

My parents, Eve and David Blum, and my grandparents, Margie Granach and Elsie Blum, provided bottomless emotional support throughout my career as a student. I appreciate beyond words for the privileged life they provided that allowed me the opportunity to pursue higher education.

Looking back on my 11 years at CMU, I am especially grateful for the early friendships that continued well past first graduation. From those who stayed in Pittsburgh to those who kept in touch, from regular tabletop, board, and/or dance gaming partners to those who offered company and a couch to sleep on while I traveled, I thank 8 Gianfortoni, Adam Blank, Alan Vangpat, Alex Yuschik, Anand Subramanian, Andrew Krieger, car bauer, Carolyn Sawyer, Elizabeth "Kempy" Kemp, Elly Fong-Jones, E. Forney, Emily J Leathers, Emma Cating-Subramanian, Eric Faust, Gabriel Mengde Routh, Greg Hanneman, hazel virdó, Jack Ferris, Jake Lengyel, Jason "Wyrm" Deng, Joshua Keller, Joshua Wise, Josiah Boning, Julia Tuttle, Kartik Null Cating-Subramanian, Kellie "K2" Medlin, Laura Abbott, Maija E. Mednieks, Margaret Meyerhofer, Michael Arntzenius, Michael "Sully" Sullivan, Miriam Zimmerman, Naomi Saphra, Nathaniel Wesley Filardo, Rhett Lauffenburger, Richard Ha, rglenn, Ryan Pearl, Todd Eisenberger, and Zachary McCord for their wonderful friendship. I am equally grateful to the new friends I made during my graduate years, whether through board/card/dance gaming, tea drinking, movie watching, and/or as more permanent Pittsburgh residents that I am extremely sad to move away from: Alexandra Lee Falk, Alexis Dyer, Barbara Jensen, Brendan McShane, Brian E. Saghy, Carlo Angiuli, Cassie Orr, Charlie McGuffey, Dan Guzek, Danny Gratzer, David Renshaw, Evan Cavallo, Gabriele Farina, Grant Wu, Isaac Grosof, jennylin, Karl Schulz, Kevin Nguyen, Kristy Gardner, Lea Albaugh, Marlena N. Abraham, Matt Stanec, Matthew Maurer, Priya Donti, Roie Levin, Ryan Kavanagh, Sarah R. Allen, Sol Boucher, Sophia Wu, Stefan Muller, Zachary Waldman, Ziv Scully, and the rest of ThursDz already named previously. And to anyone who may read this list and feel even slightly dejected that their name was left out: I owe you a nice dinner. Get in touch.

日本語を一緒に勉強したり、会話したり、色々教えてくださったりした仲間の学習者さんたちとネイティブスピーカーさんたちへ、心から感謝を申し上げます。外国語を学ぶのは一生の二番達成感だと考えます。

Lastly, a shout out to the Android: Netrunner community. Netrunner is a deckbuilding strategy card game that depicts cyberspace hacking battles (completely different from real-life computer security) alongside dystopian social issues (becoming strangely more familiar with each passing year). Its characters have a diversity of ethnicity and gender identity unparalleled in any other game I know of, and the community that has grown around it reflects those values. I thank especially the other Stimhack moderators, #rainbow-coalition, Nisei, and the local Pittsburgh crew for their support, encouragement, and mutual exchange of inclusive ideals in these turbulent yet hopeful times.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Please hold on. This thesis will be departing shortly for the Landslide terminal, baggage claim, ground transportation, and ticketing.

—Pittsburgh International Airport (paraphrased)

Pick up any recent concurrency-related research paper, and chances are its very first sentence will tell you that concurrency bugs are notoriously difficult to find, reproduce, diagnose, and/or fix [AAA+15, AAJS14, BAD+10, BBC+10, BG16, Blum12a, Blum18d, CGS13, CMM13, CWG+11, DL15, EA03, EMBO10, FF09, HH16, Hua15, KAJV07, KGW10, LCB11, LLLG16, LTQZ06, MLR15, MQ07, MQB+08, NWT+07, OAA09, SBGH12, Sul17b, YCW+09, YNPP12, YSR12]. How could a single programming technique be so troublesome as to inspire multiple decades of verification research? Allow me to begin this thesis by answering this question in layperson's terms, using two parables.

My favourite way to explain concurrency to a non-technical audience is by analogy with unlocking car doors. In many models of cars, the door handle and locking mechanism are intertwined in such a way that the door cannot unlock while its handle is being pulled [TT18]. Most car users, then, have suffered the experience of trying to open the passenger door at exactly the same moment as the driver turns the key in the lock: it fails to unlock, and they must beseech the driver to turn the key again! If they had pulled the handle any later, it would have opened as normal; any earlier, and it would have been as though not having the key at all, but would at least have unlocked properly at the first turn of the key later. One would reasonably expect to never have to turn the key more than once, so this violation of expected behaviour is a *bug*, and its dependence on the two travelers' actions *interleaving* in a particular way makes it a *concurrency bug*. This example involves only three events (pulling the handle, releasing the handle, and turning the key), but what of a system with hundreds or thousands of times more? Unfortunately, the number of possible interleavings grows *exponentially* with the number of events.

The Islamic historian Ibn Khallikān told perhaps the first cautionary tale of exponential explosion [Kha1274, KdS1868]. Sissa Ibn Dàhir, credited with the invention of chaturanga (the precursor to modern chess), was invited by King Shihrâm to request any reward he desired. Sissa requested that a grain of wheat be placed on the first square of a chessboard, two in the second, and so on, doubling the number of grains in each previous

square until all 64 be filled. The king at first laughed, thinking it a pittance, realizing only later that to fulfill this would require more than all the wheat in all the cities on Earth. Modern programmers will recognize this sum as 2^{64}-1, the largest value representable by an unsigned 64-bit integer, or approximately 18 billion billion. The number of interleavings in a concurrent system grows at a similarly intractable rate (although not necessarily exactly doubling each time), with the number of chessboard squares filled with grain corresponding to the number of events in a program's execution. A concurrency bug, then, is a single poisoned grain of wheat, which must be rooted out from the stock before feeding the townspeople. And Sissa himself is the programmer, whose software is invariably large enough to be far beyond the reach of any exhaustive verification strategy.

1.1 Motivation

To take advantage of multiple cores for performance, programmers must write software to execute concurrently, i.e., using multiple threads to run several 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, just as when multiple travelers interact with car doors at the same time. When an unexpected interleaving produces undesirable program behaviour, for example, by corrupting shared data structures, or by leaving the door locked as in the car example, we call it a *concurrency bug*. The specific interleaving required to expose such a bug 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, she will chance upon such an interleaving should one exist. 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 interrupts it if her patience runs out before it finds a bug.

Stateless model checking [God97] is an alternative way to test for concurrency bugs, or to verify their absence, which provides more reliable coverage, reproducibility, and verification than stress testing. A stateless model checker tests programs by forcing them to execute a new unique thread interleaving on each iteration, capturing and controlling the randomness in a finite *state space* of all possible interleavings. However, attempting to exhaustively check the entirety of such state spaces is akin to Sissa's reward: 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 think she cares 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 a checker must be told how to identify and control the execution of each thread. Many undergraduate computer science curricula culminate in project-oriented systems classes [Eck18a, Eck18b, PRB09] in which students implement these very types of programs, but are left to their own devices when it comes time to root out bugs. Some expert concurrency research wizards may be willing to add manual annotations to their code for the sake of verification, but requiring manual effort is a serious downside for anyone with a looming deadline, and especially so for students who are still learning basic concurrency principles in the first place.

Finally, in the struggle to meet ever-increasing performance demands, hardware and software alike has grown more and more complicated features for fine-grained concurrency control. Recent stateless model checking research is already beginning to address some such cases, allowing for new sources of nondeterminism beyond simply the ability to interleave threads arbitrarily. For example, relaxed memory consistency allows for multicore systems to reorder accesses to main memory [AG96], which introduces store buffer nondeterminism in addition to thread nondeterminism [ZKW15], and event-driven programming models, useful especially in mobile applications to improve responsiveness and reduce power consumption, allow for threads to reenter themselves via event handlers [JMR+15]. Transactional memory, which allows the programmer to specify arbitrary atomic execution sequences, has been supported by software libraries for decades [HM93], but only recently have processor manufacturers introduced hardware-backed transactions [HKO+14], replete with new complexity, and no verification tool which can accurately model the concurrency of programs that use them yet exists.

1.2 Contributions

This thesis will address each of the problems introduced above, establishing stateless model checking as a modern verification technique that can meet realistic human needs. My thesis statement is as follows:

> *Combining theoretically-founded automatic reduction techniques and user-informed heuristic ones, stateless model checking can sufficiently mitigate exponential explosion to be a practical testing technique for inexperienced users and real-world programs alike.*

The foundation of this work is Landslide, a stateless model checker I have built over the last seven years, first debuting in my M.S. thesis [Blum12a], so named because it tests the stability of Pebbles programs students write in 15-410 at CMU. Since then, I have extended it with many features and algorithms, some which work behind the scenes and others which rely on human feedback to be effective, in pursuit of this thesis statement.

The first half of the statement will serve as the overarching theme of this work: that as impossible a problem as perfect formal verification may be, one may still hope to achieve meaningful test results using an algorithm that knows its own limits, and compensates for them with heuristics informed by the user's own human intuition. The second half of the statement can be broken down into three parts: coping with exponential explosion, helping students, and addressing modern concurrency models; which correspond to the three major contributions of this thesis, as follows:

1. **Data-race preemption points and Iterative Deepening (Chapter 4).** I will present Quicksand, a stateless model checking execution framework that manages multiple simultaneous Landslide instances to automatically cope with exponential explosion. Quicksand is powered by *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 [SBG12] to decide on-the-fly whether each state space is worth pursuing, and uses data race analysis [SI09, FF09] to find new preemption point candidates based on a program's dynamic behaviour. This chapter will include a soundness proof, showing that testing only synchronization and data-race preemption points still constitutes a full formal verification of the test, and a large evaluation, comparing its performance to three prior-work approaches across 600+ unique tests. I will show that Quicksand outperforms prior work in terms of both finding bugs quickly and verifying correctness when no bug exists.
2. **Educational use (Chapter 5).** For the past seven semesters, I have offered a fully-automated version of Landslide to students in 15-410, CMU's undergraduate Operating System Design and Implementation class, for use as a debugging aid during the thread library project [Eck18a]. I have also extended Landslide to handle Pintos kernel projects from other universities [PRB09]. In the two most recent semesters, I collaborated with Operating Systems course staff at the University of Chicago, which uses Pintos, and at The Pennsylvania State University, which recently adopted CMU's thread library project, to provide their students an opportunity to benefit from Landslide as well. 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 and prove that stateless model checking is suitable for use in an educational setting.
3. **Hardware transactional memory (HTM) (Chapter 6).** I will introduce a new concurrency model for stateless model checkers to emulate HTM's execution semantics in terms of existing concurrency primitives. This model is accompanied by a soundness proof which allows checkers to avoid simulating conflict abort rollbacks, reducing the state space substantially, while preserving the formal verification guarantee. I have implemented a new transactional testing mode in Landslide, with support for the many different abort reasons HTM introduces, optional weak atomicity semantics for simulating software TM instead, and a "retry set" reduction algorithm to identify and prune new types of equivalences. The evaluation includes several real-world HTM programs and benchmarks, and shows that Landslide can both find bugs and verify correctness using the new model with reasonable performance.

1.3 Meta

Dear reader, whether you have come to read this whole document, to pick and choose whichever sections interest you, or just to skim the figures and footnotes looking for easter eggs, it is truly an honor to have you here. Before we get underway, allow me to present some notes on reading this thesis that may soften the blow of these 200+ pages.

Experience

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 easy reading to all. 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.

Vision

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. Should any vision-impaired reader require fully-textual figure descriptions, I would be happy to provide them upon request.

Language

Pronoun use will vary between more specific and more ambiguous to convey additional nuance. The singular "I" connotes 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, as well as to challenge readers' unconscious bias about gender in computer science. Gender-neutral pronouns will be used to refer to individual students who participated in the user studies, as well as on the author themself.

Fonts

The body of this thesis is set in Bitstream Charter, the monospace font is Inconsolata, the epigraph font is Alexa, the Japanese font is Heisei Mincho, and the Arabic font is Amiri.

Code

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 Landslide codebase is open-source under the BSD license, available at `https://github.com/bblum/landslide`. Chapter 3 provides more detail and 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.

1.4 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, concluding with a glossary of terminology for the reader's convenient reference.
- **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 introduces Quicksand, a wrapper framework for Landslide which incorporates data race analysis at run-time to find new preemption points and uses the new Iterative Deepening algorithm to more intelligently choose which state spaces to test, corresponding to contribution 1 above.
- **Education:** Chapter 5 discusses my evaluation of Landslide in CMU's and The Pennsylvania State University's class environment using thread libraries based on the Pebbles kernel, and in the University of Chicago's OS class environment using the Pintos kernel, corresponding to contribution 2 above.
- **Transactions:** 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 identifies stateless model checking's remaining shortcomings and how new research might address them.
- **Conclusion:** Chapter 9 offers some thoughts on the future of the field.

46

2 MID PHD

HOW I LEARNED TO STOP WORRYING*

Set the State Space Size at any level you want (10^1-10^5). This event counts as 5 Research Operations for the purpose of required Research Operations.

Graduate from CMU if used as an event.

Chapter 2

Background

We can't be free until we learn to laugh at ourselves. Once you look in the mirror and see just how foolish we can be, laughter is inevitable. And from laughter, comes wisdom.

—G'Kar, Babylon 5

This chapter introduces the necessary background material on concurrency (§2.1), stateless model checking, (§2.2), data race analysis (§2.3), and the relevant undergraduate operating systems classes (§2.4). It concludes with a glossary of terminology (§2.5) that I hope the reader may find convenient as a quick reference while reading later chapters as well.

2.1 Concurrency

This section presents an elementary introduction to concurrent programming. I have taken care to make it accessible to readers with even only basic programming experience, familiar with flow control, functions, memory, and objects at most, although I cannot avoid the inherent difficulty of multiple layers of abstraction building upon one another. For an explanation to be "elementary" means that it requires very little be known ahead of time to approach it, not necessarily that it be easy [Fey64].

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 [Lu14], and weak memory consistency models [AG96] – 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*.

Shared state

When a programming language offers multithreaded parallelism but forbids access to any shared state between threads [VWW96, KN18], 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 which 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 [Tur37]).

[1] With some exceptions in kernel-level programming, discussed later.
[2] Fun exercise for the reader: Show why 2 is a possible output, but 1 is not!

```
int x = 0;
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 (a)'s compiled assembly, in which two concurrent iterations of the loop yield one net increment of `x`.

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

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 which thread interleavings can arise surrounding their use. When building a tool for testing concurrent programs, one 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, et cetera; see §3.2.1) at the cost of increasing the implementation and theoretical complexity of the analysis (§6.3.3). 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 [Eck18a]).

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

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

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 block (i.e., wait) 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 [HM93] offers a more optimistic approach: critical sections of code 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 logic, or in software, via library calls and a log-based commit approach. Software transactions (STM) [ATLM+06] can be used on any commodity processor, but must impose runtime overhead associated with logging. Hardware

```
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.

transactions (HTM) [Intel18] 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 [HKO+14] 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 [GNU16].

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 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 [GW94, God97] 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

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

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.

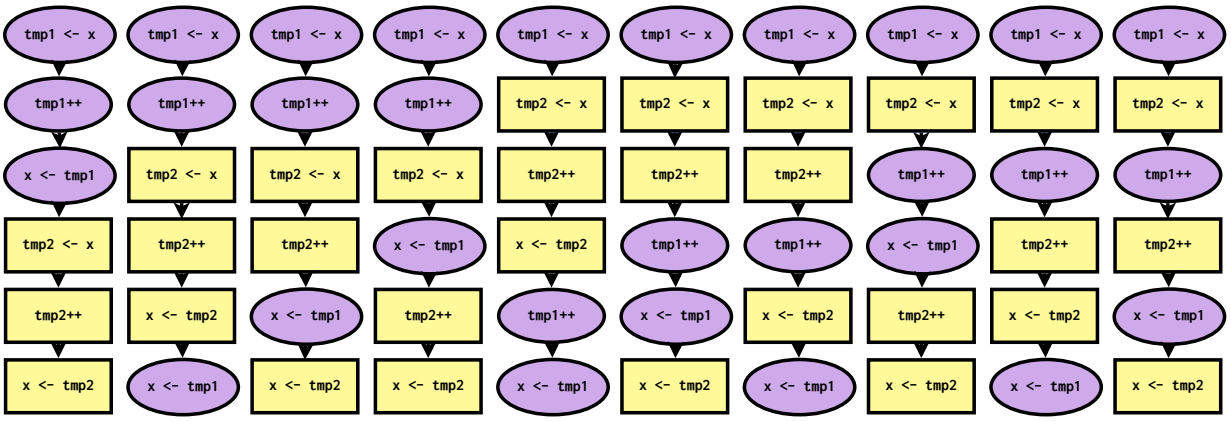
Static versus dynamic analysis

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 [GÖ1]. 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 [EA03]. Dynamic analysis, our approach, identifies program behaviours that are definitely wrong, so each bug report is accompanied by concrete evidence of the violation. Assertions, access violations, use-after-free of heap memory, and deadlock are examples of commonly-checked failures, although a checker may also include arbitrary program-specific predicates.

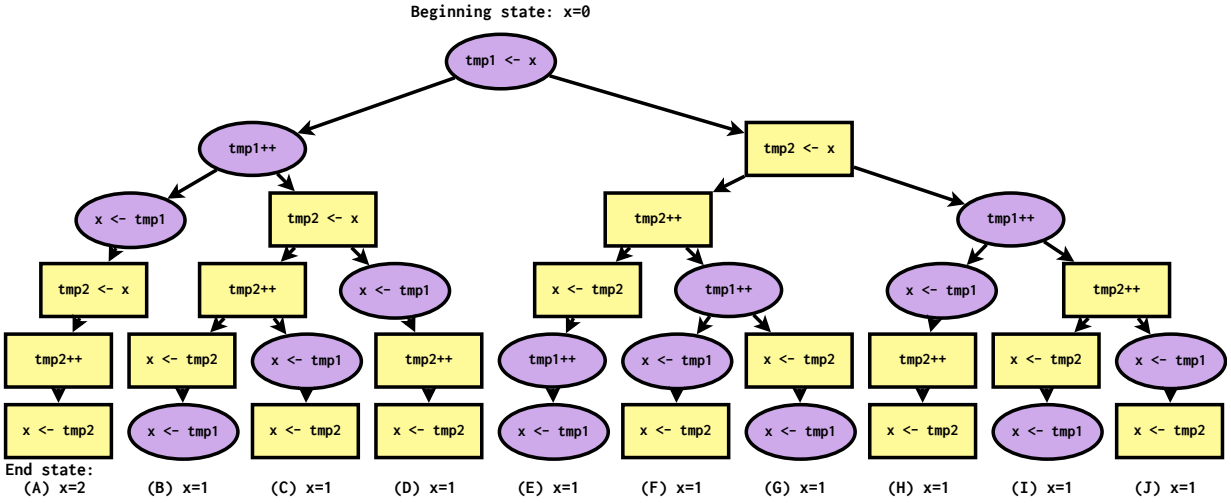
Preemption points

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 synchronization API boundaries (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 possible behaviours of a program, we must check the executions corresponding to every branch of the tree. Addressing the scaling problem in



(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.

this exponential relation is the central research problem for all model checkers, which this thesis will address in Chapter 4.

Some model checkers, such as SPIN [Hol97], explicitly store the set of visited program states as a means of identifying equivalent interleavings. 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 often 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: *reduction techniques*, i.e., removing redundant interleavings from the state space when we can prove they are equivalent to some interleaving already tested, and *search heuristics*, i.e., prioritizing interleavings judged as more likely to contain bugs (should bugs exist) in case we are unable to exhaustively test all interleavings after all.

Reduction techniques

Dynamic Partial Order Reduction [FG05] (henceforth, DPOR) is the most popular algorithm for mitigating the exponential explosion that arises as program size increases.

To define it abstractly, 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 [Maz87], and tests at least one execution from each equivalence class.

For a friendlier, more concrete example, consider Figure 2.5, which 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). Any events executed before the thread 1/thread 2 sequence, any other runnable threads besides these two, 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

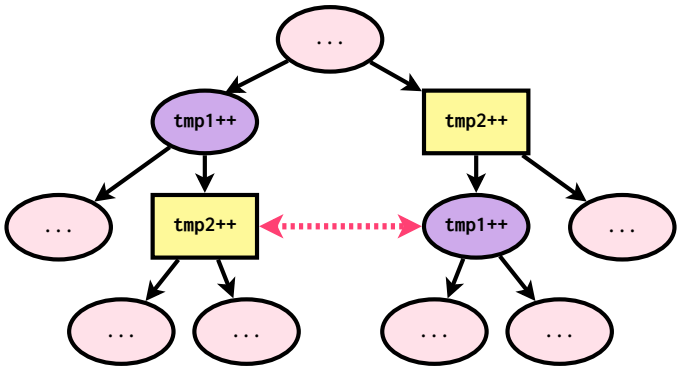


Figure 2.5: DPOR identifies independent transitions by different threads which can commute without affecting program behaviour. Here, because each thread's local operations on `tmp1`/`tmp2` are not shared memory conflicts, the states marked with the red arrow are guaranteed identical. Hence, only one of the subtrees need be explored.

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 [AAJS14], parallelizable DPOR [SBGH12], SAT-directed model checking [DL15], Maximal Causality Reduction [Hua15], and DPOR for relaxed memory architectures [ZKW15].

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 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 [MQ07] 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 [BBC+10] 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

Data race analysis [SBN+97] identifies pairs of unsynchronized memory accesses between threads.

2.3.1 Definition

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 [Lam78, OC03], described below).

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

```
   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.

A data race analysis may be either *static* (inspecting source code) [EA03] or *dynamic* (tracking individual accesses arising at run-time) [SI09]. 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* [Lam78] as the order relation between accesses. [SES+12] and [OC03] 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*.
- The *Limited Happens-Before* relation [OC03] reports 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 [SI09], but at the cost of necessarily reporting some such *false positives*.
- In recent work, the *Causally-Precedes* relation [SES+12] 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 Implementation [Eck18b, Eck18a]. 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 [BO01, BO10]. Both P2 and P3 are built using the *Pebbles* system call specification, outlined in Table 2.1.

P2

The thread library project [Eck18a] has two main components: implementing concurrency primitives, and implementing thread lifecycle and management routines. Students implement it on top of a reference Pebbles kernel implementation (named Pathos [SFJ09]), which is distributed as an obfuscated binary to protect the subsequent challenges of P3. The required concurrency primitives for P2 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 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 also called *P* and *V* in other literature, from Dutch, originally *passering* and *vrijgave* [EWD35], later often explained as *proberen* and *verhogen*). 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 (also called R/W-locks or 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 con-

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.

dition 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 custom, fixed 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)` completes execution of the calling thread, recording an exit status value. This function does not return to its callsite. 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 specified thread ID exits, then returns, collecting its exit status.

Other than `thr_init()` (which is necessarily single-threaded), 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 [ABB+86] 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 `rfork` model of Plan 9 [PPTT90], or the `clone` model of Linux.

Although the interfaces are not mandated as 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"), and some approach to blocking threads indefinitely;
- A virtual memory implementation (page directories and tables, frame allocator, et cetera);
- A program loader;
- Lifecycle management code for creation and destruction of threads and tasks;
- Other miscellany such as a suite of fault handlers to ensure no user program can cause the kernel itself to crash.

Because user programs can invoke any combination of system calls or fault handlers simultaneously, concurrency bugs can arise from the interaction of any subset of kernel components with each other. The most common bugs students 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 [Eck18f]. 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 amplify the probability of a particular thread interleaving, likely to expose the bug, which is otherwise very unlikely to arise in normal execution. The reference kernel specification [Eck18b] 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 [Sul17a] 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 [PRB09] is used at several universities, including the University of Chicago, Berkeley, and Stanford. Its 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 1-initialized semaphore), `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, et cetera.

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 [Ous16, Gun14], or supplemental HTTP server and `malloc()` assignments [Jos16]. This thesis is not concerned with these assignments. as they are are largely architectural/algorithmic projects rather than concurrency-oriented ones. 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. Use of *italics* within definitions will indicate terms also defined elsewhere in the list.

1. **Actor.** (Also "actor concurrency model".) An independent execution component of a concurrent system. In *pthread*-like multithreaded programs, each *thread* is an actor. In distributed systems, each machine is an actor.
2. **API.** Short for "application programming interface" (which nobody ever says). A set of function signatures which comprise the interface to a body of code. For example, the standard *mutex* API consists of `mutex_init()`, `mutex_lock()`, `mutex_unlock()`, and optionally `mutex_trylock()` and `mutex_destroy()`.
3. **Atomic.** Said of a sequence of operations within a single thread, whose intermediate state cannot be observed by or interfered with by certain conflicting operations from another thread. Must always be qualified as "atomic *with respect to* X", where X is the set of possibly-conflicting operations to be protected against. Apart from hardware-provided atomic assembly instructions (e.g., `xchg`), i.e., for any multi-instruction sequence, setting X as "everything" can be surprisingly difficult to achieve safely in practice; for example, disabling interrupts in kernel code does not prevent other CPUs' simultaneous execution.
4. **Atomicity violation.** A term introduced by AVIO [LTQZ06] to refer to a particular class of *concurrency bug* in which the programmer's expectation of *atomicity* around a certain sequence of operations is violated by another thread interleaving in between. This is distinct from language-level *data races*, which may result in bugs by being reordered by the compiler or hardware (see *relaxed memory*); however, this thesis checks programs at the executable level and assumes *sequential consistency*, so all data races that still result in bugs (e.g., unprotected `x++`) must technically also be atomicity violations. Also, this thesis is focused on identifying failures, leaving root cause diagnosis to the user, so will simply avoid the term entirely.
5. **Benign.** Said of a *data race*, or of *nondeterministic* behaviour in general, which does not lead to a *concurrency bug* in any *interleaving*.
6. **Branch.** Another term for *interleaving*, with an emphasis on the *execution tree* interpretation of *state spaces*. May also refer to an entire subtree outside of the current interleaving, characterized by the choice to run a certain *thread* at a certain past *preemption point*, comprising all possible interleavings which may arise with that as a prefix.
7. **Bug.** See *concurrency bug*.
8. **Completeness.** In logic, said of an inference system in which any proposition consistent with the axioms can be proved. When speaking of *state space reduction*, could mean opposite things depending on perspective: complete from an interleaving coverage perspective would mean not skipping any buggy interleavings; complete from a state space reduction perspective would mean not testing any two equivalent interleavings without pruning one. I define *soundness* as the dual from the reduction perspective (i.e., not pruning anything unsafe to prune), which is actually identical to the coverage perspective definition of completeness. To avoid confusion, this thesis avoids this term altogether, using *optimality* to refer to the reduction perspective definition instead.

9. **Concurrency bug.** A program behaviour which manifests *nondeterministically* depending on the thread scheduling pattern, which is not desired or intended by the programmer. This thesis further assumes these to be identifiable by computable predicates on the execution state, excluding some real bugs which may result in data corruption that never actually leads to crashes or assertion failures under the test case in question.
10. **Conflict.** See *memory conflict*.
11. **Data race.** An unsynchronized pair of memory accesses between two threads that can, may, or did execute concurrently. *Race* for short (see also *race condition*). Distinguished further in this thesis as *limited happens-before* races and *pure happens-before* races; see §2.3. Distinct from the more nebulously-defined *concurrency bug*, in that they are identified precisely by analysis on execution traces of one (or all) thread interleavings. This thesis considers any memory accesses not protected by a known trusted *mutex API* to be potential races; this may include atomic instructions such as `xchg`, which most all-data-races-are-bugs research does not include, but are necessary for *Iterative Deepening*'s *verification* guarantee (§4.3).
12. **Deadlock.** An execution state among a set of N *threads*, where each is blocked waiting for an event that can be caused only by another thread in the set [Eck18e]. May arise either deterministically or concurrently; *Landslide* considers it a bug in any case.
13. **Deterministic bug.** As *concurrency bug*, but manifesting in all (or at least most) thread interleavings, making it easy for programmers to find and debug without advanced tooling. This thesis will occasionally use "manifests on the 1st interleaving Landslide tested" as an approximation, without necessarily checking for its presence or absence in other branches.
14. **Disjoint.** Said of a pair or of two sets of memory accesses when there are no two elements that share an address and also are not both reads.
15. **DPOR.** *Dynamic partial order reduction*.
16. **Dynamic analysis.** Program analysis technique which involves analyzing concrete execution(s) of a program, either through tracing (via compiler-inserted instrumentation, simulation, or other means) or through post-hoc analysis of event logs. Searches for concrete evidence of undesirable program behaviour, and is blind to what the source code looks like (or even what language it be written in). Opposite of *static analysis*.
17. **Dynamic partial order reduction** (DPOR). Exploration algorithm for *stateless model checkers* which identifies alternate interleavings to explore likely to result in reorderings of *memory conflicts* observed in the current branch. Runs iteratively upon completion of each branch, tagging one or more sibling branches for future exploration, skipping over ones guaranteed to be *independent*. See §3.4.2.
18. **Equivalent.** Said of two *interleavings* where one can be transformed into the other using permutations of only *independent interleavings*. Note the difference in "type

signature" with *independent*, which applies to *transitions* instead.

19. **Estimation.** An algorithm whose input is the known structure of a partially-explored *state space* and whose output is a guess at the ultimate size of the state space when exploration finish. Output may be in units of either time or branches. See §3.4.3.
20. **ETA.** Short for *estimation* (or "state space estimate"), typically with nuance for units of time rather than branches. From "estimated time of arrival".
21. **Execution tree.** Representation of the *state space* by joining interleavings with common prefixes into a subtree with a branch at each point they diverge, forming a tree overall. See §2.2.1; Figure 2.4.
22. **False negative.** Said of a tool's failure to produce a bug report when a bug does exist after all. In theory this may mean that the *test case* might have been insufficient to expose the bug (which the programmer may discover later via manual inspection, or when her code fails in production, or which may remain unnoticed), however, for pragmatic, tool-oriented discussion, this thesis will use the term to mean bugs observable under the given test case which the tool itself fails to find.
23. **False positive.** Said of a bug report which, upon subjective manual inspection by the programmer, does not indicate a bug after all. In principle, Landslide's bug-detection predicates always indicate something truly wrong, so it cannot produce false positive bug reports; in practice, however, infinite loop detection and deadlock detection in the presence of ad-hoc synchronization are both *heuristic* (§3.4.6), so may produce false positives after all, although I have never witnessed one to date.
24. **Happens-before.** Relation between two events executed by different threads in a single execution trace which captures the notion of concurrent execution. Comes in two flavours, *pure happens-before* and *limited happens-before*.
25. **Heuristic.** A testing strategy informed by empirical evidence rather than theoretically-founded algorithms. For example, *ICB* is a heuristic way of prioritizing *interleavings* with fewer *preemptions*, based on the observation that most concurrency bugs in practice can be exposed with as few as 2-3 preemptions. For another example, *Landslide* heuristically judges the program to be stuck in an infinite loop when it has executed at least 4000 times longer than previously-observed "safe" executions, based on the observation that no student has ever written correct thread library or kernel code that can exhibit such extreme interleaving-dependent variance.
26. **ICB.** Short for "Iterative Context Bounding". See §3.4.5.
27. **Independent.** Said of two *transitions* by different threads, when they share no *memory conflicts*; i.e., they can have no effect on each other's behaviour even if reordered. Note the difference in "type signature" with *equivalent*, which applies to *interleavings* instead.
28. **Interleaving.** A sequence of *transitions* of many threads representing the complete or partial execution of a *test case*.

29. **Iterative Deepening.** The *heuristic* search ordering strategy and framework for incorporating *data-race preemption points* used by *Quicksand*. One of this thesis's major contributions. See §4.2.
30. **Kernelspace.** Opposite of *userspace*. Execution mode in which code has the ability to modify critical system state. In addition to having access to privileged CPU instructions, kernelspace code is also responsible for implementing safe concurrent scheduling, safe virtual memory protection and isolation guarantees, et cetera.
31. **Landslide.** Your friendly local *stateless model checking* implementation, written by yours truly. The framework upon which all this thesis's contributions are built. Documented in Chapter 3.
32. **Landslide-friendly.** Said of a *test case* that is well-suited for *model checking*, i.e., contains few or no loops which, under normal circumstances, would increase the probability of a bug manifesting, but in principle do not make any new behaviours possible that would be impossible without the loop.
33. **Limited happens-before.** Variant of *happens-before* that relates two events only when a blocking/signalling synchronization affecting the second thread's runnability, such as `cond_wait()`/`cond_signal()` or `thr_fork()` occurs in between. In contrast with *pure happens-before*, indicates not just that the events were observed to happen non-concurrently in the execution in question, but also that they could not be reordered even in alternate interleavings.
34. **Lock.** A data structure used to ensure *atomic* access to shared state with respect to other *threads* using the same lock. See §2.1.3.
35. **Memory conflict.** A pair of memory accesses between two *threads*, where at least one is a write, meaning that executing them in a different order from the one observed may produce different program behaviour. If both are writes, reordering them may cause a different value to be globally visible in that location to subsequent parts of the program. If one is a read and one is a write, reordering them may cause the read access to return a different value, possibly affecting that thread's subsequent logic.
36. **misbehave.** An optional system call offered by *Pebbles* kernels, especially the official reference kernel, that changes the scheduler's behaviour, according to a numeric mode argument, to amplify the likelihood of specific illustrative low-probability *thread interleavings*. Used in several *P2* test cases, although exactly how thread scheduling changes according to each specified mode is kept secret to preserve the intended challenge of debugging (see §2.4.1).
37. **mutex.** Short for "mutual exclusion lock". The name *P2* and *pthreads* use for their *lock API*.
38. **Model checking.** In general, refers to the whole research field of checking program implementations against formal specifications in a theoretically rigorous way. In this thesis, used as shorthand for *stateless model checking*.

39. **Nondeterminism.** Property of a program which can exhibit different behaviours across multiple runs, despite no change in the input data.
40. **Optimality.** See *completeness*.
41. **P2.** Operating systems class project in which students must implement synchronization primitives and thread lifecycle functions. *API* is very similar to that of *pthreads*. Often used here to refer to the same thread library project at both CMU and PSU, even though the latter did not call it by that name. See §2.4.1.
42. **P3.** Operating systems class project, occuring after *P2*, in which students implement their own *Pebbles* kernel from the ground up.
43. **Partial store order.** See *relaxed memory*.
44. **Pebbles.** The educational kernel architecture used in 15-410 at CMU. See §2.4.1.
45. **Pintos.** The educational kernel architecture used in CMSC 23000 at the University of Chicago, as well as several other universities. See §2.4.2.
46. **pthreads.** POSIX thread library for Linux programs. The modern pthreads model traces its lineage back to DEC's Systems Research Center threads [Bir89] and Mach's C threads [CD88], and is now specified by ISO/IEC/IEEE [IEEE09].
47. **Preemption.** The act of interrupting one thread's execution to force a context switch to another. Can be performed either deliberately by the *model checker* or randomly by the *scheduler* as driven by system interrupts.
48. **Preemption point.** A single instance of a program state or code location where a *preemption point predicate* returns true. This thesis will also use this as shorthand for preemption point predicates, using the longer term only when the precise difference is important.
49. **Preemption point predicate.** A test of an intermediate execution state returning whether the *model checker* should inject *preemptions* to force another thread to interleave. Often simply "is it the beginning of `mutex_lock()` or `mutex_unlock()`?", but may be extended to include the program counter values of known *data races*, or to further restrict by predicates on the current stack trace. See §3.4.1. Often called *preemption point* for short, with context hopefully making the distinction clear.
50. **PSO.** *Partial store order*. See *relaxed memory*.
51. **Pure happens-before.** Variant of *happens-before* that relates two events when any synchronization operation introducing a memory barrier, which would prevent compiler or hardware access reorderings, occurs in between. In contrast with *limited happens-before*, indicates that the events were observed to execute non-concurrently in this execution, but could possibly be reordered to be concurrent in alternate interleavings.
52. **Quicksand.** Testing framework which manages the execution of multiple *Landslide* instances, each configured with a different set of *preemption points*, using the *Iterative Deepening* algorithm. Also supports various other execution modes in which only one Landslide instance is run at a time. See Chapter 4. In later chapters,

"Landslide" will refer to the project as a whole, including Quicksand, while the term "Quicksand" will be used only when specific reference to Iterative Deepening is required.

53. **Race condition.** Another term for *nondeterminism* in general, which may also carry more specific nuance depending on context. The term dates back (at least) to 1954, describing electrical signals' nondeterministic order of arrival in circuit design [Huf54]. That paper further distinguished race conditions as either "critical" or "non-critical", corresponding to *buggy* or *benign* in our terminology. CMU 15-410 teaches the term consistently with this definition, although may also refer to the buggy subset only (i.e., *concurrency bugs*) for brevity. Recently, many systems research papers have used the term to refer to *data races* in particular.[4] To avoid confusion, this thesis will avoid the term entirely, using the above two more precise terms instead.
54. **Reduction.** The act of making a *state space* smaller by finding multiple execution sequences that can be proved *equivalent*, such that all but one may be skipped over.
55. **Relaxed memory.** Also known as *weak memory*. A family of multiprocessor execution semantics that allows reads and/or writes of shared memory to become visible to different CPUs in different sequences. Under *total store order* (TSO), which x86 implements, one CPU's writes may appear reordered after its subsequent loads, from the perspective of another CPU. Under *partial store order* (PSO), either writes and/or reads may appear reordered after other subsequent writes and/or reads. On ARM's implementation of PSO, all such combinations are possible. Safe concurrent programming on such architectures requires careful use of acquire and/or release memory barriers [Sul17b]. By contrast, *sequential consistency* (SC) permits no such reorderings. For further discussion see §4.3, §6.4.4, and §7.5.
56. **Schedule.** Another term for *interleaving* with a more theoretical nuance.
57. **Scheduler.** The component of an operating system kernel responsible for tracking the set of runnable threads, handling system interrupts, and allocating execution time among them (in round-robin, priority-driven, or other manner). May also to refer to the analogous component of *Landslide* which overrides the decision of the kernel scheduler.
58. **Sequential consistency.** See *relaxed memory*.
59. **Soundness.** Said of a search algorithm which provides the property that if a bug exists under the given test case, given arbitrary testing time, it will eventually be found; in other words, admits no *false negatives*. When said of a *reduction* strategy as applies to an existing search algorithm, means that any bugs found by the original algorithm will still be found under the reduction, even if there were false negatives (e.g., applying *DPOR* without using *data-race preemption points*). Used by analogy with logic, in which said of an inference system, means contradiction cannot be

[4] In the author's personal experience, many readers of their M.S. thesis [Blum12a] have misinterpreted its title to believe that Landslide be only a data race detector, rather than a model checker.

derived from the axioms. See also *completeness*.

60. **Stateless model checking.** Concurrency testing *dynamic analysis* technique involving controlling a system's *nondeterminism* to exhaustively check all possible *thread interleavings*. See §2.2. Often "MC" for short in this thesis (to mean either "model checking" or "model checker" depending on context). This name is potentially confusing for tools which explore *state spaces*: the term "stateless" refers to the implicit way they use *DPOR* to ensure *sound* checking of all possible program states, without actually explicitly storing those states in memory. The second half of the name is another historical accident, as most modern MCs do not check "models" in the sense that programs be verified against external formal specifications, but rather the program's internal assertions serve as informal specifications, alongside the MC's own bug-detection predicates.
61. **State space.** The set of all possible behaviours of a program under any interleaving of threads. As opposed to modern *stateless model checkers*, older MCs stored the set of states explicitly, attempting to cover them by traversing as few edges (executing as few *transitions*) as possible; hence the name. Modern MCs instead cover these states implicitly by instead exhaustively executing all *interleavings*, using *DPOR* to prune redundant *equivalent* ones which could only revisit old states. The slightly misleading name is kept for consistency with prior work (sorry).
62. **Static analysis.** Class of program analysis techniques which process the source code, usually at syntax-tree or LLVM IR level, looking for suspicious implementation patterns, rather than actually observing behaviours that arise during execution. Opposite of *dynamic analysis*.
63. **Systematic testing.** Another name for *stateless model checking*.
64. **TA.** *Teaching assistant*.
65. **Teaching assistant.** A student who volunteers to help run a class, holding office hours, grading projects, and so on, compensated either with course credit or pay from the university. Probably has taken the class herself in the past.
66. **Test case.** A piece of code, separate from the code under test, to drive execution of the code under test according to a certain pattern designed to expose interesting behaviour. In the context of *P2*, the thread library is the code under test, and client code thereof, such as *paradise_lost.c*, is the test case.
67. **Thread.** Unit of concurrent execution in a typical modern program, concretely represented as a set of private CPU registers, including program counter. Probably has its own stack as well, allocated by the userspace thread library. A program may have as many parts of its code executing simultaneously as threads exist.
68. **Transition.** A sequence of execution steps by a single thread, bounded by the two different program states at beginning and end. In this thesis, will be used exclusively when the terminal program states are *preemption points*, and no intermediate state is a preemption point as well.
69. **Total store order.** See *relaxed memory*.

70. **TSO.** *Total store order*. See *relaxed memory.*
71. **Userspace.** Opposite of *kernelspace*. Execution mode in which programs, such as *P2* or client code thereof, run with limited privileges, such as no access to the CPU's control registers, interrupt descriptor table, ability to perform device I/O, et cetera. Assumed to run with the virtual memory protections standard in any modern kernel (such as crashing when dereferencing a null pointer).
72. **Vector clock.** Data structure for time-keeping in distributed systems which uses a list of epoch numbers to track when the execution of any given pair of *actors* should be considered simultaneous/concurrent. Used in *Landslide* to implement *pure happens-before*.
73. **Verification.** Formally certifying the correctness of a program under certain properties. In this thesis, will always be constrained by the set of behaviours the *test case* is capable of exposing in the first place.
74. **Weak memory.** See *relaxed memory.*

Chapter 3

Landslide

Somewhere is the promise of an uncharted trail, with 700 branching limbs and 700 ways to fail.

—ThouShaltNot, Cardinal Directions

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. Figure 3.1 visualizes Landslide in relation to its execution environment, showing how it communicates with each of its surrounding and/or simulated programs.

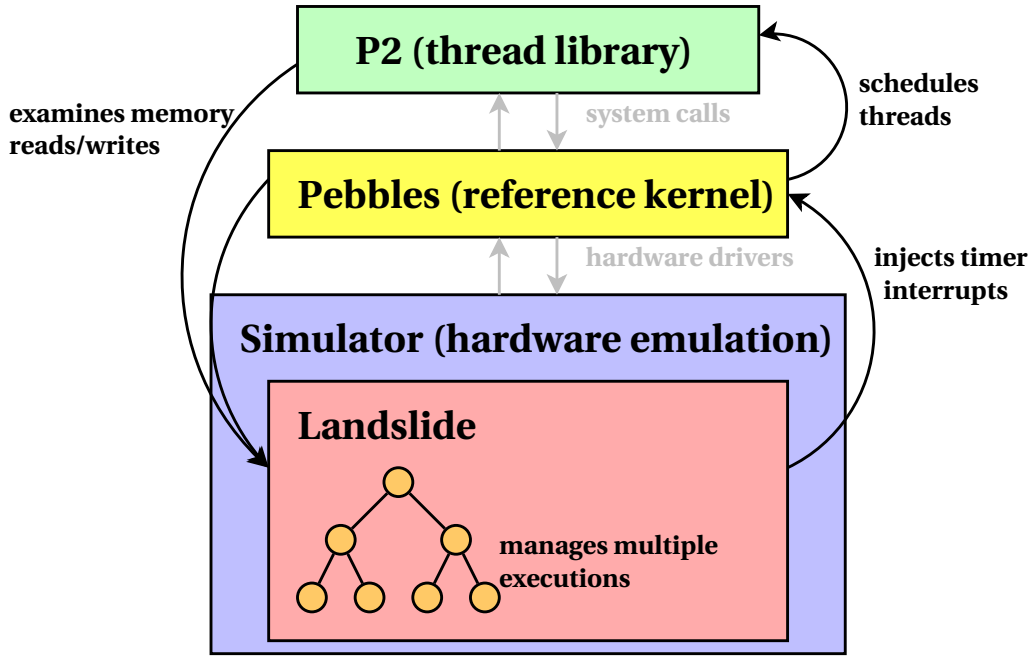


Figure 3.1: Landslide's execution environment when testing 15-410 student projects.

Overview

From an implementation point of view, Landslide's main "execution loop" is simply the simulated CPU's own fetch-decode-execute loop: each time it simulates an instruction or memory access, it invokes Landslide through the simulator's module interface. More abstractly speaking, the general sequence of events during a Landslide test runs as follows.

1. Instrument the test program by inspecting its binary, learning the addresses of important functions, global variables, et cetera (§3.3.9). Further interpret test-specific annotations as informed by configuration options (§3.1.3), test case annotations (§3.1.4), and kernel annotations (§3.2).
2. Execute the program under simulation.
   (a) At each instruction:
      i. Update the *scheduler*, a state machine which depicts the runnable and/or blocked threads currently existing in the simulated program, as well as a set of action flags to track what each thread is up to, such as waiting to lock a mutex at a given memory address (§3.3.2).
      ii. Record reads and writes to shared memory (§3.3.3).
      iii. Check whether the current program state constitutes a bug, and if so, emit a preemption trace (§3.1.5, §3.3.6) and halt Landslide's execution. Bug-detection predicates range from simply checking if the program tripped an assert or accessed invalid memory, to checking the above-mentioned memory accesses against the current heap allocation state for use-after-frees (§3.3.3), to querying the scheduler for deadlocks (§3.4.6), to checking heuristically for infinite loops and livelock (§3.4.6).
      iv. Check whether the current program state is a preemption point (§3.4.1).
      v. Query the scheduler to detect when the test has completed successfully, i.e., all threads have finished executing normally (§3.3.2).
   (b) At each preemption point, identified as in 2(a)iv above:
      i. Check the set of memory accesses since the previous preemption point, recorded as in 2(a)ii above, for conflicts with other threads (§3.3.3), and further for data races (§3.4.4).
      ii. Select which thread should run next (§3.3.5). Heuristically detecting yield-blocked threads may inform this decision (§3.4.6).
      iii. Checkpoint the execution state in case future executions should wish to rewind and try a different thread from the one first chosen here (§3.3.1, §3.3.5).
      iv. Force the chosen thread to run by injecting timer interrupts (§3.3.2, §3.3.4).

   Note that Landslide maintains the invariant that each transition between two preemption points consist of instructions executed by exactly one thread; i.e., every thread switch must be punctuated by a preemption point.

3. At the end of each execution, identified as in 2(a)v above:
   (a) Analyze the set of memory conflicts, computed as in 2(b)i above, using Dynamic Partial Order Reduction (DPOR) to decide which preemption point to backtrack to and which new thread to run (§3.4.2). This may be constrained by Iterative Context Bounding (§3.4.5).
   (b) Estimate the structure of the resulting state space to predict overall runtime and total interleavings needed to test (§3.4.3).
   (c) If DPOR selected an alternate interleaving to explore, backtrack the simulation state to that preemption point (§3.3.5) force the new thread to run (§3.3.2, §3.3.4), and repeat from step 2. Otherwise, halt Landslide's execution, declaring the explored state space free of bugs.

Implementation

As of this thesis's writing, Landslide supports the use of two possible simulators:

- **Simics** [MCE+02], a proprietary simulator licensed commercially by Wind River, used at CMU in 15-410 to run Pebbles thread libraries and kernels, and
- **Bochs** [Law96], 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 under the BSD license and available at `https://github.com/bblum/landslide`. The HEAD commit at the time of writing is 5e4082a. In the event of a GitHub outage or the above link otherwise not working, a backup of the repository checked out at that commit may be found at `https://www.cs.cmu.edu/~410/landslide/landslide.tar.bz2`. 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.

Disclaimer

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 §5.1.1.

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 `./quicksand` 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 (see §5.1.2).

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, et cetera)
- `-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
- `-s`: include "secret" options when printing help text

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)
- `-0`: enable preempt-everywhere mode (§4.4.4, §4.4.5, 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 §4.3's soundness proofs, this is equivalent to `-0` (and according to my experience, much 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 (§4.2; described in detail in `id/option.c`)
- `-E ETATHRESH`: configure heuristic threshold of state space progress for judging ETA stability (§4.2; 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 an allowlist 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 denylist instead of an allowlist
- `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 preemption points 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 §4.1.1.
  - `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.

1. **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 (§4.5); corresponds to `-0` 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 ([Blum12a, §8.7.1]). Not compatible with ICB.
2. **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.

3. **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 the `tell_landslide()` annotations (§3.2.2) 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`

Because Landslide tests the ultimate value of these variables after all threads have completed execution, 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 3.2, 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 3.2: Example demonstrating the need for `misbehave` to force the kernel to yield during `thread_fork`.

Though Iterative Deepening's soundness (§4.3) 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 3.3.

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 (the same as the stack trace in the cell corresponding to that thread in the bottom row of the table), 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] §8.1 discusses how these bug reports might be further improved by adding more information still, such as identifying data-race preemption points or listing memory conflicts that occurred during each transition.

3.2 Kernel annotations

My M.S. thesis [Blum12a] investigated the annotation overhead required for a user to instrument a Pebbles kernel for use with Landslide. On average, students who volunteered for the study (conducted during P3) required 2 hours for this process. While many of these students then went on to find and diagnose bugs, this was deemed an unacceptable burden for an educational tool to impose on struggling students.

[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 3.3: Example preemption trace bug report.

Hence, 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. An alternate step-by-step guide is also available at https://www.cs.cmu.edu/~410/landslide/landslide-guide-kernel.pdf.

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, 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. Those defined with an equals sign between name and value are implemented as bash variables, which the annotation scripts check after processing the configuration to emit a corresponding `#define` within Landslide itself; those defined with no equals sign are implemented as bash functions, which the annotation scripts define in advance of processing the configuration to do something more complicated.

- `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, et cetera 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] Whether or not a kernel requires this annotation could be auto-detected in future work.
- `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.

[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 [Blum12a, §6.2.3].

- `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`).
- `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

[8]These two options replace the deprecated `kern_ready_for_timer_interrupt()` annotation from `student.c` described in [Blum12a, §6.2.3].

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

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, et cetera) 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, 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 [SBN+97] 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 implementation) 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 nobe` 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 `nobe` 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, et cetera;
- 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 (chapter 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 which 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 several queues to track threads in various states of runnability (runnable queue, deschedule queue, and sleep queue), while per-thread state is stored in `struct agents` (named after the terminology of [SBG10]) 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. Landslide also queries the scheduler state after it updates every instruction, via `test_update_state()` (`test.c`), to check the existence and/or runnability of all the system's threads and determine whether or not the test case has completed execution.

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

[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.

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 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 ma-

[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.

chine 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, et cetera).

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 execution of the upcoming instruction is completed.
- `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" [12] 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 [SBG10]) 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.[13]

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 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 Bochs process and thus the simulation 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.[14]

[12] willan on-having switched to [Ada80]

[13] 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`.

[14] 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.

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_pp()` (`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.[15] 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.

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()`[16] which provides function names and hexadecimal offsets; while for line numbers, `pebsim/build.sh` (and `pebsim/pintos/build.sh` for Pintos) 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` using 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 each function allocates), 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

[15] 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.

[16] Does the same thing as the existing `bx_dbg_symbolic_address()`, but with a better type signature.

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 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.

[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.

[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.

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`.
- `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 unmapped all of its user memory, splice in a saved "pre-vanish" stack trace (saved previously in `sched_update_kern_state_machine()`) so that 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 distasteful 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.

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.

[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.

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 allow/denylist behaviour for the former two of those commands (in a similar manner to prior work's Preemption Sealing [BBC+10]). 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 "denylist" mode: the preemption point is by default valid as long as no `without_function` calls reject it. Otherwise, operate in "allowlist" 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. It can optionally be qualified by a thread ID, most recent system call number, et cetera, as described in §3.1.3, and is queried through `suspected_data_race()`.

In preempt-everywhere mode, Landslide instead marks all shared memory accesses as long as they are not part of either the mutex implementation or 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 preemption point requested by the user with `tell_landslide_preempt()` (relic of [Blum12a], largely obsoleted by data-race preemption points (§4.2.3))

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 nobe` 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 an allowlist, 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 denylist, 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) [FG05] 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 (switched to) 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 logically concurrent (despite the serialized nature of the simulated execution). This relation is expressed in the following definitions paraphrased from [FG05].

- **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 research 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 $S = t_1 \ldots t_n$ is the smallest relation $\rightarrow_S$ on $1 \ldots n$ 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 [Lam78] 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).

[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.

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 $O(\max(m,n))$ 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 $M(t_i)(a) = \text{write} \vee M(t_j)(a) = \text{write}$ 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, Landslide 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`.

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 nobes`, 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, $i < j$, in the current interleaving S. If $t_i \rightarrow_S t_j$ then they cannot be reordered, and if $t_i \not\leftrightsquigarrow_S t_j$ 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()`).

[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, et cetera) 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.

- Otherwise, there must be some third thread runnable at s;[23] then, return all $I_{ik} = (t_1 \ldots t_{i-1}, T_k)$ 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 [God96, FG05, AAJS14] 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 $t_1 \ldots t_{i-1}$; in other words, the set of all sequences $S' = t_1 \ldots t_{i-1}, u_i, \ldots u_{n'}$ 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 nobe` 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 nobe_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, and automatically included in the result of any future iterations of DPOR until they are tested. 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, mortal readers may prefer the following example of using DPOR on the program from Figure 2.1, 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 3.4.

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

[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 $t_i \not\rightarrow_S t_j$.

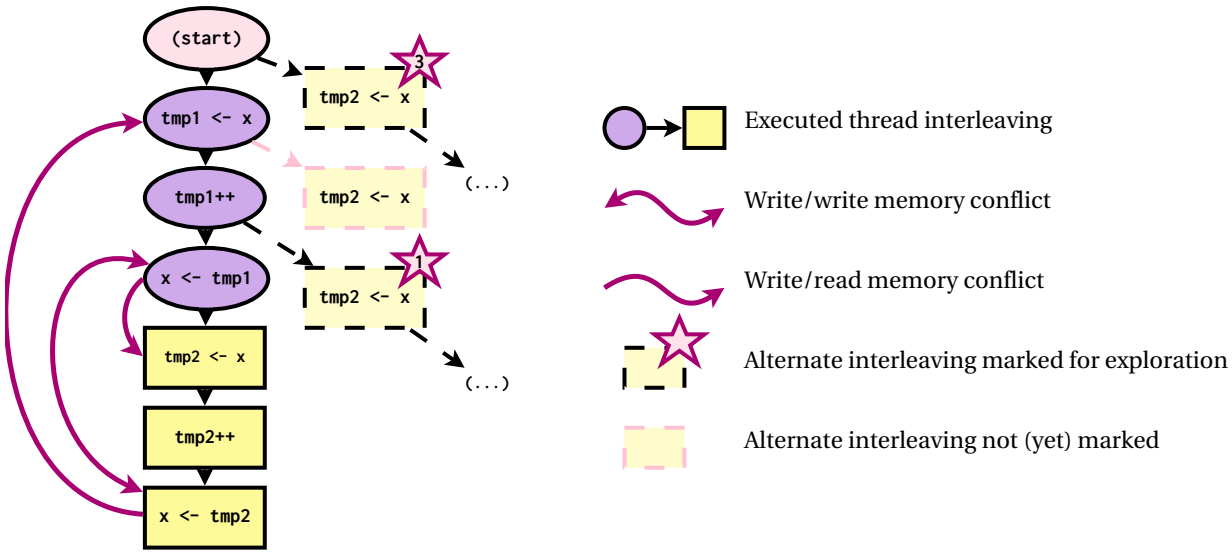


Figure 3.4: Result of a single iteration of DPOR.

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 ★3) to avoid storing in memory any representation of exponentially-sized subtrees outside of the current branch. Note also that in ★3, 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 3.5 shows the outcome (with the new ★2 appearing in ★1's subtree, to be explored before ★3).

After marking ★1 and ★3 from Figure 3.4'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 [Hua15] has extended DPOR to find such value-based equivalences, although is beyond this explanation's scope.

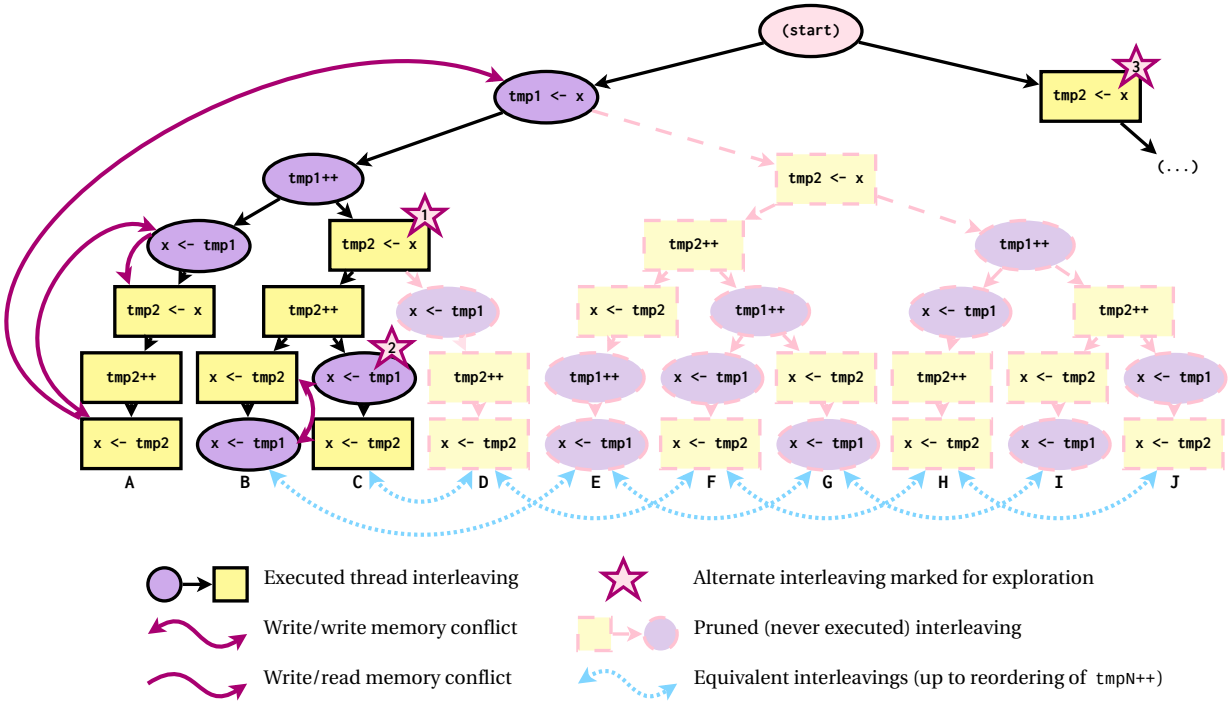


Figure 3.5: Result of 3 DPOR iterations, pruning 7 redundant branches.

Finally, let us consider the final result after DPOR runs out of remaining unexplored marked branches, shown in Figure 3.6.

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 3.6 hints, its reduced subset state space still contains redundancy, arising from the fact that one pair of those four 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 3.7 summarizes the relevant interleavings to highlight one such equivalence.

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

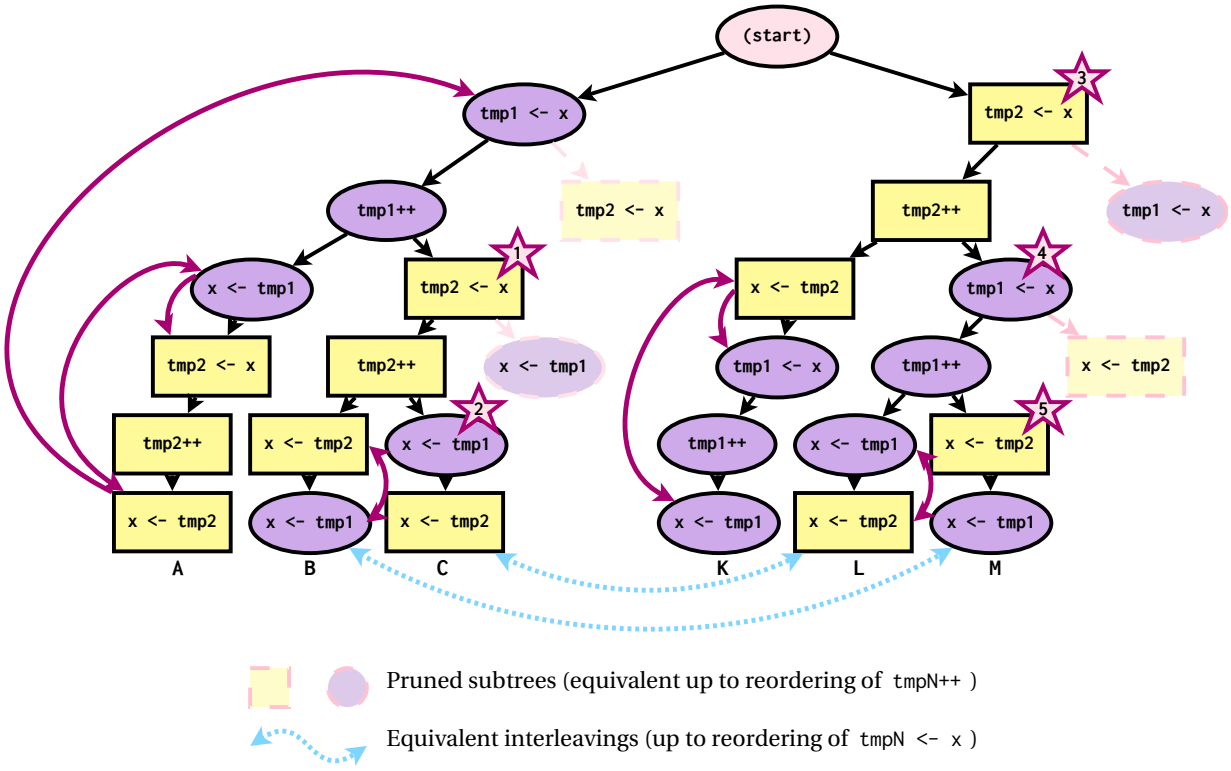


Figure 3.6: DPOR's termination state, having reduced 20 interleavings to 6.

```
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 3.7: Motivating example for the sleep sets optimization. Three of Figure 3.6'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 $\mathbf{T_1}$ therein was already checked, preserving DPOR's $O(n)$ memory footprint.

This corresponds to the *sleep sets* optimization described by prior work [God96, FG05, AAJS14], so named because it effectively puts $\mathbf{T_1}$ "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) [AAJS14] 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 optimization 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 3.6, 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 [KSTW06] and later adapted to DPOR by [SBG12]. 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 (§6.3.3).

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). §8.3 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 [Mye85] that represents the estimated percentage towards completion, approaching 100% (not necessarily monotonically) as exploration continues.

Summarizing prior work's formal definition [SBG12], the proportion at a terminal node v_n[25], preceded by an execution sequence $(v_1 \ldots v_{n-1})$, is computed as:[26]

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

where $\text{marked children}(v_i)$ 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 $b = (v_1 \ldots v_n)$ explored so far:[27]

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

It is easy to see how these 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,

[25] Prior work [KSTW06, SBG12] 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 [SBG12]: the missing $F(v_i)$ is 0, using the empty fit strategy.

[27] Simplified from [SBG12]: $t(b)$, the time for each branch, is 1, because we are counting them.

requiring them to be recomputed, which would require 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 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 $v_{i-1} \rightarrow v_i$. 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 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 3.6, 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 happens 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 [Lam78]. Landslide implements this via the vector clocks approach described by FastTrack [FF09]. 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 [PS03] (as presented in [FF09]) 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.

1. 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()`
2. 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 [MQ07] 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:

Algorithm 2: ICB exploration ordering.

```
1 for B ∈ [0..max_P(U)] do
2 |   foreach S ∈ U, P(S) ≤ B do
3 |   |   Execute(S)
4 |   end
5 end
```

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 would be to state "**foreach** $S \in \text{sort}_P(U)$" in pseudocode, implementing such an ordering would be much less straightforward. [28]

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.[29] The DPOR implementation then checks, for some I_{ij} it wants

[28] The pseudocode algorithms found in many conference papers often fail to be straightforward to translate into usable implementations [Blum17, Figure 5].

[29] 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.

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[30] 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[31], 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 [MQ07, TDB14, WR15], although §4.5'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. Preserving the full formal verification, i.e., continuing iteration until $B = \max_P(U)$ not only remains exponential in N, but also introduces a factor of $\max_P(U)$ repeated work. Landslide takes this approach for now (rather than stopping at any finite cutoff). Future work could memoize already-tested interleavings so that each iteration of B could test only those schedules with exactly B preemptions, restoring the original B-independent (but still exponential) complexity.

Bounded Partial-Order Reduction

Prior work [CMM13] 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

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

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

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 [CMM13].

3.4.6 Heuristic loop, synchronization, and deadlock detection

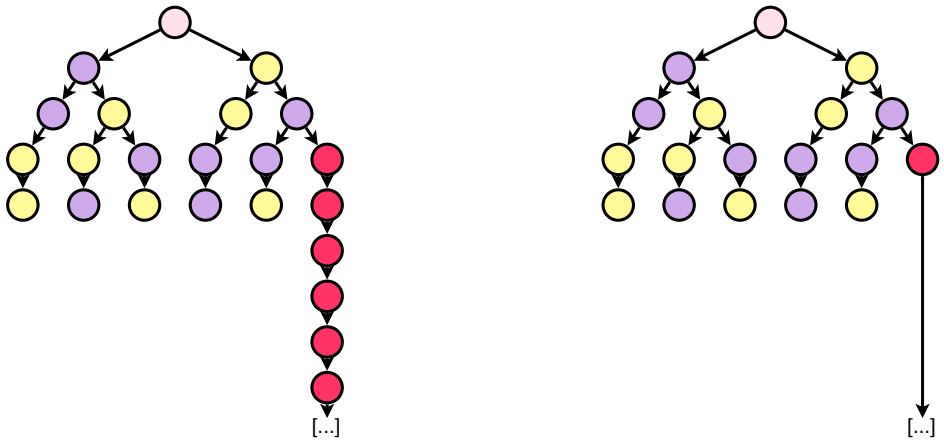
Despite the ease of automatically instrumenting a fixed concurrency API such as P2's, the variety of student implementations inevitably results in many behaviours outside Landslide's model. This section documents the heuristics Landslide uses to approximate a program's formal behaviour in such situations.

Infinite loop detection

All modern presentations of stateless model checking assume finite program length. Even though all test cases used in this thesis's experiments are hand-written to ensure runtime which is not just finite but also short (on account of exponential state space sizes), bugs may still cause a program to get stuck in an infinite loop unexpectedly. Detecting such loops in general is of course uncomputable [Tur37], but Landslide has the benefit of knowledge from past iterations to inform its sense of how long the program "should" run.

Landslide checks for two distinct categories of potentially-infinite loops, visualized in Figure 3.8. Firstly, it keeps a running average of how many preemption points deep each completed interleaving has been in the past. Whenever a new preemption point is reached, Landslide checks if its depth is greater than the heuristic constant factor of 20 (`PROGRESS_DEPTH_FACTOR`) times the previous average. If the depth exceeds that cutoff, it reports an infinite loop bug. Whether this represents a livelock or just a mundane sequential logic bug is for the user to decide. However, as different interleavings do often execute different program logic and vary in length accordingly, Landslide waits to test 10 interleavings (`PROGRESS_CONFIDENT_BRANCHES`) before applying this heuristic; in the case of fewer, it scales the depth factor by a heuristic exponential factor of 1.1 to represent its lower confidence (`PROGRESS_BRANCH_UNCERTAINTY_EXPONENT`). In the special case of the first branch ever tested, Landslide will abort after a fixed preemption point depth limit of 4000 (`TOO_DEEP_0TH_BRANCH`) – a program with such N would probably have an impossibly large state space anyway.

Secondly, a program may get stuck in a loop where no preemption points are encountered each new iteration. For example, race-induced data corruption may cause a list to end up circularly linked, leading a search or append operation to fail to terminate. Landslide also maintains a running average of instructions per transition, and each instruction, compares if more instructions have elapsed since the last preemption point than a constant



(a) Infinite loop around preemption points. (b) Stuck between preemption points.

Figure 3.8: Detecting infinite loops heuristically by comparison to past interleavings.

times that average. Because transitions may themselves vary greatly in number of instructions as each represents completely different program logic, this heuristic is much more lenient than the preemption-point-counting one, using a multiplicative factor of 4000 (`PROGRESS_TRIGGER_FACTOR`). If such a loop is encountered within the P2 synchronization primitives, which should generally be free of $O(n)$ operations, Landslide uses a more aggressive cutoff of 2000 (`PROGRESS_AGGRESSIVE_TRIGGER_FACTOR`). Also, as transitions between data-race preemption points may have as few as 1 instruction each, Landslide caps the average from below at a minimum of 1000 (`PROGRESS_MIN_TRIGGER_AVERAGE`) to keep the average relatively stable.

These checks are both implemented in `check_infinite_loop()` in `landslide.c`.

Yield-loop detection

One very common student implementation pattern in P2s and kernels is to open-code synchronization between threads using a loop that spins around a condition that the loop itself cannot fulfill, waiting for another thread to allow it to proceed, rather than using the established synchronization API.[32] If Landslide failed to recognize that the thread was in principle blocked, just as if it had called `deschedule` or `cond_wait()`, this would result in an infinitely deep branch as it keeps trying to schedule the waiting thread, blocking all of its attempted context switches to the thread that could make progress. These loops need not necessarily `yield` each iteration: they may spin blindly, assuming progress is being made on another CPU, which is not possible under Landslide as it serializes execution.

Landslide detects such ad-hoc *yield-loop blocking* by keeping a counter for each thread to track how many `yield`s it has invoked since the last interesting activity. "Interest-

[32]Even when the student doesn't open-code any such synchronization and always uses `deschedule` or primitives built thereupon, `mutex_test` (§5.1.3) often relies on this functionality, as data-race preemption points within `mutex_lock()` would otherwise disrupt Landslide's ability to recognize blocking on a contended lock.

ing" here is defined heuristically as any other known P2 API call, with the exception of `mutex_lock()` and `mutex_unlock()`.[33] Whenever this counter reaches the heuristic cutoff of 10 (`TOO_MANY_YIELDS`), Landslide declares the thread blocked, and treats it just as if it had invoked `deschedule()` for purposes of DPOR (`check_user_yield_activity()` in `user_sync.c`). In cases where `yield` itself is not involved, Landslide also counts the number of atomic instructions (`xchg`, `xadd`, `cmpxchg`, et cetera), and marks the thread blocked if it exceeds 100 such (`TOO_MANY_XCHGS_TIGHT_LOOP`).[34] When such a loop contains neither `yield` nor atomics, Landslide falls back on the standard infinite loop detector and reports a bug directly, as described above. Future work could extend this to include more modern ways of establishing memory safety such as acquire/release barriers and hardware transactions.

In order to detect when a thread should be unblocked from its yield loop, Landslide simply leverages DPOR's existing computation of memory conflicts: whatever condition the blocked thread was waiting for will show up in the conflicts between it and whichever thread fulfills it.[35] Hence, every memory conflict detected during DPOR is also checked against any currently yield-loop-blocked threads, unblocking them in the case of a match (`check_unblock_yield_loop()` in `user_sync.c`). If that memory access is not sufficient to let the blocked thread start making progress again, it will simply trigger the yield-blocking heuristic again. In theory, this could result in a livelock between two threads, each in principle blocked in a yield loop, but where the conditions of the loop happen to conflict with each other, causing the threads to keep waking each other up; however, I have never observed this in practice, as the conditions checked by such blocking loops are usually read-only. Future work could heuristically address this by deprioritizing such threads, as measured by the number of times they've yield-blocked, so that a third thread which could actually make progress may run if it exists.

False-positive deadlock avoidance

In cases where the yield-loop heuristic described above produces false positives, i.e., blocking a thread which could make progress on its own after all, Landslide must avoid reporting a deadlock bug if no other thread ends up waking it up through memory conflicts. After all other threads quiesce, which ordinarily would trigger deadlock detection, Landslide checks the system for any yield-looping threads that were blocked heuristically.[36] If any exist, it forces them awake and allows them to proceed; if they are truly blocked in

[33] Landslide must recognize yield-blocking loops that contain mutex operations to allow for open-coded reimplementations of `cond_wait()`, which for example the `paraguay` test uses intentionally, because it is testing the correctness of the student's `cond_wait()`.

[34] If preemption points exist in between, instead only 20 such (`TOO_MANY_XCHGS_WITH_PPS`), to avoid stressing DPOR's $O(n)$ independence computation.

[35] A yield-blocking loop may depend on another thread's execution with no memory conflicts at all simply by checking the return value to observe whether the yield was successful. In principle, Landslide should check for such loops whenever the yielded-to thread's runnability changes; at present, this is not implemented, and instead leans on the false positive deadlock detector to wake the blocking thread up.

[36] This includes threads blocked on specific mutexes using the `blocked_on_addr` field, which is set when `yield` is invoked within `mutex_lock()` without waiting for 10 loops.

principle, they will merely trip the yield-loop limit and go back to sleep again, whereupon Landslide will issue a deadlock bug report after all.

Two other forms of heuristic blocking exist in Landslide which are also subject to this retry procedure. Firstly, when an interleaving has already exhausted the number of preemptions allowed under ICB's current bound, other threads which are runnable but which require preemptions to switch to are considered "ICB-blocked". When yield-looping mixes with ICB-blocking, the yielding thread will require a preemption for the other thread to fulfill its blocking condition. In such a case, simply spending all 128 deadlock-avoidance retries on the yield-looping thread will not solve anything, so the ICB-blocked thread must be forced awake with higher priority, disregarding the preemption bound, to allow the system to progress. Secondly, in programs which use HTM, threads blocked by retry sets (§6.2.5) must be forced awake with priority before yield-blocked threads, along similar reasoning.

Landslide will retry this process up to a heuristic limit of 128 times (`DEADLOCK_FP_MAX_ATTEMPTS` before issuing a deadlock bug report. The overhead of this check is quadratic time in the number of retries permitted (as DPOR is quadratic in the overall branch depth), although this is negligible compared to the exponential size of state spaces overall. If the heuristic limit is too small, very "loopy" programs (which invoke xchg or yield therein) could falsely exhaust this limit while not being truly deadlocked, so a good limit should be well higher than the total number of concurrency events expected for any Landslide-friendly test. The cost of a high limit manifests in the length of preemption traces when deadlock is declared, which will display a proportional number of meaningless preemption points, although future work could easily truncate them retrospectively. This check is implemented in `try_avoid_fp_deadlock()` in `arbiter.c`.

3.5 Summary

This chapter has presented Landslide, a stateless model checker for Pebbles and Pintos kernels and Pebbles userspace thread libraries. For the most part, Landslide's contribution to the world of concurrency testing is as a feat of engineering which combines many existing techniques from prior work in a single implementation. These include:

- Uses Dynamic Partial Order Reduction (DPOR) [FG05] to prune equivalent interleavings while exploring the state space (§3.4.2).
- Uses its simulator's memory tracing integration [MCE+02, Law96] to power data race analysis [PS03, FF09] (§3.4.4), as well as DPOR.
- Supports Preemption Sealing [BBC+10] to configure the set of active preemption points and hence keep state space sizes manageable (§3.4.1).
- Extends DPOR with Recursive and Weighted Backtrack Estimation [SBG12] to predict overall ultimate state space size and completion time (§3.4.3).
- Extends DPOR (at the user's option) with Iterative Context Bounding [MQ07] to heuristically uncover bugs faster should they exist, at the expense of overall completion time (§3.4.5).

However, over the years Landslide has also grown some convenience, accessibility, and or optimization features that could be considered contributions in their own right.

- Extends Preemption Sealing for finer-grained control over whether individual functions and/or global variables should be considered for, or excluded from, DPOR and/or data race analysis (§3.2.1; `thrlib_function` and `ignore_dr_function`, respectively). This allows a test case author to precisely target the exact subset of code she is interested in testing, avoiding unnecessary state space inflation from trusted code.
- Adds a mutex testing mode to configure whether the data race analysis should consider locks' internal accesses (§3.1.3; `TESTING_MUTEXES`). This allows the user to verify that her lock implementation correctly provides mutual exclusion (upcoming in §5.1.3; `mutex_test`), before relying on them as trusted code in subsequent tests.
- Implements the sleep set extension to DPOR [God96, FG05, AAJS14], which identifies and skips further equivalences based on which memory accesses DPOR intended to reorder, in a way that is slightly less optimal compared to prior work, but much simpler to implement and requires less state at runtime (§3.4.2).
- Heuristically identifies infinite loops and livelocks by leveraging knowledge of how long the test program should be expected to run obtained during previous executions, uniquely suited to stateless model checking (§3.4.6).
- Provides automatic code instrumentation facilities, beyond just the simple task of identifying important function boundaries, that attempt to understand a wide variety of even the most unusual student implementations (§3.3.7, §3.3.8). This includes an advanced heuristic for detecting ad-hoc synchronization loops that Landslide should treat as blocking, even if it does not trigger the automatically-inserted blocking annotations (§3.4.6), and a further heuristic to ensure that any inaccuracy on the part of the first heuristic does not result in false-positive bug reports (§3.4.6).

The following three chapters, Chapters 4, 5 and 6, will each build upon Landslide to present the thesis's main research contributions.

Chapter 4

Quicksand

There are awful, sad things in this world. But there are a lot of things worth protecting, too.

—Kaname Madoka, Puella Magi Madoka★Magica

There is a fundamental disconnect between existing stateless model checkers and human users when it comes to testing concurrent code meaningfully within a fixed CPU budget. Existing tools test systems according to a fixed preemption strategy, leading to runtime dependent entirely on the complexity of the test program, which may range from minutes to tens of thousands of years. Meanwhile, users approach testing with a finite amount of patience, usually not varying from one test cycle to another as their code changes and evolves: students frantically testing last-minute changes facing a project deadline will likely wait no longer than an hour for test results, while a company preparing its product for production deployment may spend upwards of weeks on rigorous stability testing. Regardless of the use case, a stateless model checker committing in advance to test whichever single state space arises from its fixed strategy is certain to either under- or over-shoot its user's needs. A model checker which preempts the system too often will fail to complete the test in time, and one which preempts infrequently enough to complete with time to spare will leave the user wondering if it overlooked any bugs.

This chapter presents Quicksand, an execution framework for model checking to manage this trade-off at run-time. Given a fixed CPU budget, representing the user's patience for testing, Quicksand dynamically alters its preemption strategy based on data race analysis ([SI09, FF09], §3.4.4) and optimizes the size of state spaces on the fly, guided by state space estimation ([SBG12], §3.4.3), to best match that budget. I will discuss the trade-off inherent in number of preemption points used (§4.1), introduce *Iterative Deepening*, the algorithm that Quicksand uses to automatically navigate that trade-off (§4.2), prove its soundness relative to the more expensive full verification approach (§4.3), and present a large evaluation of Quicksand against several state-of-the-art approaches in which Quicksand performs best on both bug-finding and verification (§4.5).

The main contributions of this chapter were published in the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '16) as *Stateless Model Checking with Data-Race Preemption Points* [BG16].

4.1 Motivation

When configuring a model checker's preemption strategy, or indeed, choosing a model checker to begin with, the resulting state space is *parameterized* by the set of preemption points. In the first example of Figure 2.4 I constructed the state space by expanding both threads' `x++` operations into three pseudo-assembly instructions, then designating every instruction as a possible preemption point, yielding $\binom{3+3}{3} = 20$ total interleavings. Later, §3.4.2 showed that DPOR prunes 16 of those as equivalent, although the reduced state space size is still combinatorial in the number of *conflicting* events, rather than the total number. For larger tests, committing in advance to test all possible interleavings quickly becomes impractical. Accordingly, many existing model checkers opt for preempting on only a subset of execution events, such as synchronization API boundaries.

4.1.1 Preemption points

Consider the new example program in Figure 4.1(a), in which one thread protects its accesses to `count` with a mutex, while the other protects its accesses with atomic increment instructions. Assuming `count` is only ever incremented, never decremented, the assertion in Thread 2 expects both of its preceding increments to be visible, no matter how many other threads come incrementing `count` simultaneously. However, this assumes any other accesses to `count` use the same protection mechanism, i.e., `atomic_xadd()`, but since Thread 1 uses a mutex (which Thread 2 never touches), the threads can interleave to cause the assert to fail, as shown in Figure 4.1(b). Figure 4.2 shows the resulting state space supposing the model checker preempts only on synchronization APIs.[1] However, Figure 4.1(b)'s interleaving involves preempting Thread 1 between its load and store of `count`, which is not a known synchronization call, but rather a data race. Hence, none of the interleavings in Figure 4.2's state space will expose the failure; a *data-race preemption point* is required to find this bug.

How should a model checker know to instrument this particular data race for preemption in order to find the assertion failure lurking underneath? Committing in advance to preempt on every instruction is certain to include this data race, but invites massive state space explosion. Even as DPOR helps to skip equivalent interleavings of non-conflicting transitions, DPOR itself is $O(n^2)$ in the number of preemption points in a single execution, which is not compatible with such an approach. Accordingly, stateless model checkers must find more efficient ways to be able to uncover bugs such as these. In related work, Portend [KZC12] proposed to combine data race analysis with preemption-driven artificial scheduling, although it obtains its data race candidates from a stand-alone, single-pass analysis. In order to identify every data race that could possibly arise under the given test case, a model checker must check many different interleavings to begin with, perform the

[1] Most modern C/C++ programs invoke atomic memory instructions by using compiler intrinsics, which could themselves be instrumented as a known synchronization API. However, not all programs are guaranteed to use well-understood interfaces; in fact, in the 15-410 class projects to be tested in the upcoming evaluation (§4.5.1), students are encouraged to roll their own atomics to get more experience writing x86 assembly. For the sake of this example, I leave `atomic_xadd()` uninstrumented.

Initially `int count = 0; mutex_t m;`

	Thread 1		**Thread 2**
11	`mutex_lock(&m);`	21	`atomic_xadd(&count, 1);`
12	`count++;`	22	`yield();`
13	`mutex_unlock(&m);`	23	`atomic_xadd(&count, 1);`
14	`assert(count >= 1);`	24	`assert(count >= 2);`

(a) Example program with synchronization API calls highlighted.

	Thread 1		**Thread 2**	**Value of** `count`
11	`mutex_lock(&m);`			0
12a	`int tmp = count;`			0
		21	`atomic_xadd(&count, 1);`	1
		22	`yield();`	1
		23	`atomic_xadd(&count, 1);`	2
12b	`count = tmp + 1;`			1
13	`mutex_unlock(&m);`			1
14	`assert(count >= 1);`			1
		24	`assert(count >= 2);`	1

(b) Buggy interleaving of (a) in which the single increment of Thread 1 overwrites both those of Thread 2. Note the data-race preemption at 12a-12b.

Figure 4.1: Example bug requiring data-race preemption points to expose. Because the two threads use different modes of synchronization to protect their respective accesses to `count`, preempting on synchronization calls alone is insufficient to expose the bug. Rather, Thread 1 must be preempted between its non-atomic load and store of `count`.

Portend approach for every data race it finds, which may in turn uncover more data races (hidden in flow control paths reachable only through interleavings of the first race, perhaps), and then continue model checking those multiple races together in a bidirectional feedback loop between the two algorithms. Upcoming, §4.2 will show how I achieve this in Quicksand, and §4.3 will justify the technique's formal verification power. In the evaluation, §4.5.2 and §4.5.3 will show that Quicksand strikes a healthy balance between fast bug-finding and full verification, and §4.5.4 will justify the need for such a feedback loop by showing that many data races require model checking to reliably detect.

4.1.2 Terminology

For the rest of this chapter, I will use the following terminology as shorthand for the concepts explained above.

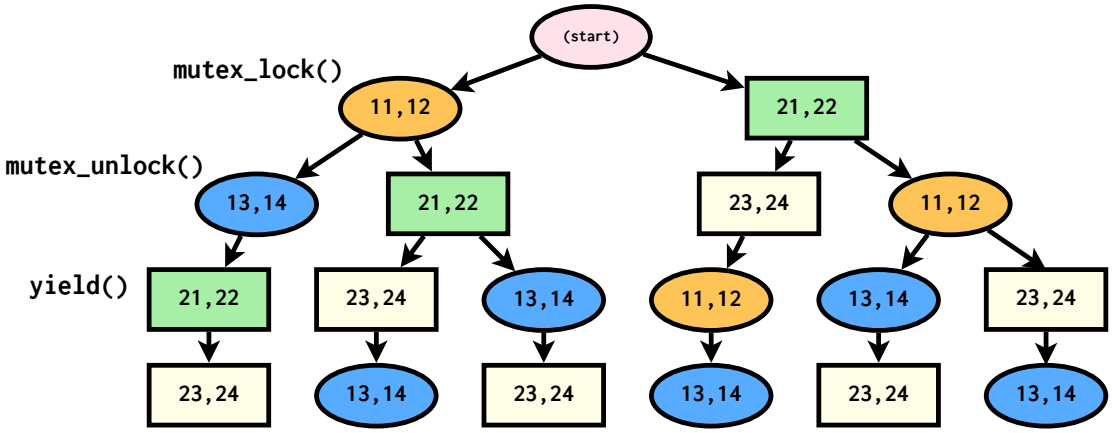


Figure 4.2: State space of Figure 4.1 with synchronization preemption points only. Note that none of these interleavings preempt Thread 1 at the necessary place (between 12a and 12b) to trigger Thread 2's assertion failure.

- **Single-state-space model checking** refers to the state-of-the-art model checking strategy, i.e., approaching each test with preemption points fixed in advance.
- **Maximal state space** refers to the set of thread interleavings possible by preempting on all currently-known preemption points, whether synchronization or data-race, i.e., the singular state space tested by *single-state-space model checking*.
- **Minimal state space** indicates the opposite: those thread interleavings requiring no more than Landslide's mandatory preemptions on voluntary context switches.
- **Data-race bug** will refer to a concrete failure, such as Figure 4.1's assertion failure above, whereas
- **Data race** refers to the racing access pair itself (§3.4.4).
- **Data race candidate** shall refer specifically to potentially-racing accesses identified by Limited Happens-Before, when disambiguation with *data races* is necessary.
- **Data-race preemption point** denotes a custom model checker configuration, issued after finding a *data race*, requesting it to preempt each involved thread just before its racing memory access (§4.2.3).
- **Benign data races** are those that, when reordered in an alternate interleaving, do not lead to a *data-race bug*, while
- **False positives** are *data race candidates* that, upon trying to reorder them, turn out not to exist in the alternate interleaving at all, such as in Figure 2.6(b).
- **Nondeterministic data race** will refer to data races that cannot be exposed on the first thread interleaving, but require model checking to expose to begin with.

4.2 Iterative Deepening

To address the problem of choosing meaningful preemption points, I have developed an algorithm called *Iterative Deepening*, implemented in a wrapper program specific to Landslide called *Quicksand*. Named after the analogous technique in chess artificial intelligence [Kor85], Iterative Deepening is a search strategy for exponentially-sized state spaces, in general, which makes progressively deeper searches of the state space until the CPU budget is exhausted. In this context, the depth roughly corresponds to the subset of preemption points used. Hence, Quicksand schedules multiple Landslide instances in parallel to test many different subsets of the available preemption points,

For the remainder of the chapter, I will use Iterative Deepening to refer to the algorithm in the abstract, which could in principle apply to any stateless model checking domain, and Quicksand to refer to the specific implementation, which relies on data race analysis and specific heuristics to optimize its testing approach for kernels and thread libraries. I will also henceforth refer to each unique set of preemption points as a *job*.

Iterative Deepening is a wrapper algorithm around stateless model checking. A model checker is still used to test each state space, and other reduction techniques such as DPOR (§3.4.2) are still applicable in each. Moreover, because Iterative Deepening treats the set of preemption points as mutable, it can add new preemption points reactively based on any runtime analysis. This chapter will focus on run-time data race analysis [SI09, FF09] as the mechanism for finding new preemption candidates. The next section (§4.3) will prove that in fact, in addition to statically-known synchronization preemption points, this suffices to provide at least as strong verification guarantees as any other possible preemption point set.

4.2.1 Changing state spaces

To introduce the Iterative Deepening algorithm, I will first show a simple approach for handling new preemption points in the absence of any CPU budget restriction.

Given unlimited testing time, switching to the new maximal state space whenever adding a new preemption point would be the quickest way to reach full verification. The maximal state space is guaranteed to subsume all execution sequences reachable in any subset state space, so considering any incomplete subset of the known preemption points would be redundant work. Algorithm 3 demonstrates this naïve approach. It is seeded with the set of all statically-known synchronization API preemption points, and invoked whenever a new data race candidate is found. The upcoming proofs in §4.3, being concerned with the verification guarantee provided when the search may complete within the CPU budget, are based on this simple version of Iterative Deepening. The user may also wish to configure her testing tool to prefer this approach, at her discretion, such as when she believes all bugs have been fixed and wants a verification as fast as possible; §4.4.4 discusses this execution mode further.

However, in many tests of even modestly-sized programs, full verification is not feasible, and focusing on the maximal state space alone is likely to be fruitless. Hence, Iterative Deepening also allows for prioritizing subset jobs based on number of preemption points,

Algorithm 3: Naïve Iterative Deepening method

Input: j, the currently-running job
Input: $\mathcal{A}$, the set of all known preemption points
if $\exists p \in \mathcal{A}.p \notin PPSet(j)$ **then**
  return NewJob($\mathcal{A}$) // New maximal state space; switch to it
else
  return j // j is still maximal
end

ETA, and whether data race candidates are included among their preemption points. It relies on state-space estimation [SBG12] to predict which jobs are likely to complete within a reasonable time, before actually testing a large fraction of interleavings for each. The overall goal is to decide automatically when to defer testing a state space, so an inexpert user can provide only her total CPU budget as a test parameter, and to enable completing appropriately-sized jobs within that budget. Quicksand seeks to maximize completed state spaces, as each one serves as a guarantee that all possible interleavings therein were tested; §4.6.4 discusses some limitations of this approach. The next three subsections will show how to schedule these smaller jobs based on their preemption points and ETAs.

4.2.2 Initial preemption points

Iterative Deepening must be seeded with a set of initial state spaces, which can be any number of subsets of the statically-available preemption points that prior work model checkers would use. The upcoming soundness proof relies on the maximal state space being included among these for verification's sake, but to optimize for finding bugs faster, implementations may wish to simultaneously to try testing subsets thereof.

For testing user-space code, Quicksand begins with the four possible combinations of preemption points from Figure 4.2: $\{yield\}$, $\{yield, lock\}$, $\{yield, unlock\}$, and $\{yield, lock, unlock\}$, By extension, these also induce preemptions on any other primitives which use internal locks, such as condition variables or semaphores. Preempting on voluntary switches such as `yield` is always necessary to maintain Landslide's invariant that only one thread runs between consecutive preemption points, so the `yield` preemption point is always implicitly enabled. For kernel-level testing, interrupt-disabling is analogous to locking, so preemptions must also be introduced just before a disable-interrupt opcode (on x86, `cli`) and just after interrupts are re-enabled (on x86, `sti`). During data race analysis, `cli` and `sti` are treated as a single global lock (note that `cli`'d memory accesses can still race with others that have interrupts on).[2] Quicksand is configured to begin with $\{yield\}$, $\{yield, lock\}$, $\{yield, unlock\}$, $\{yield, cli\}$, $\{yield, sti\}$, and $\{yield, lock, unlock, cli, sti\}$. As a heuristic, it doesn't test every intermediate subset such as $\{lock, sti\}$, which would result

[2]Some kernels disable preemption without disabling interrupts, which can be communicated to the model checker using manual annotations, and must be treated similarly. This also assumes uni-processor scheduling; for SMP kernels, replace `cli/sti` with spinlocks.

in 2^p jobs, although this could potentially be improved in future work (§4.6.3).

4.2.3 Data-race preemption points

As discussed in §4.1.1, data races may beget new interleavings not reachable by preempting on synchronization API boundaries alone. Because each data race indicates an access pair that can interleave at instruction granularity, different program behaviour may arise if the threads are preempted just before the racing instructions, some of which the programmer may not have even expected, i.e., be bugs, and it is logical to apply model checking to find or verify the absence of such bugs.

With Iterative Deepening, this is a simple matter of creating a new state space with an additional preemption point enabled on the racing instructions by each thread, as shown in Algorithm 4. These *data-race preemption points* form the foundation of Quicksand's contribution. Note that even though a data race may involve two different instructions, α and β, Quicksand's strategy is to add new state spaces with only one new racing instruction at a time. Rather than adding a single large state space, configured to preempt on both involved instructions, i.e., $AB = \text{PPSet}(j_0) \cup \alpha \cup \beta$, it prefers to add multiple smaller jobs which have a higher chance of completing in time, i.e., $A = \text{PPSet}(j_0) \cup \alpha$ and $B = \text{PPSet}(j_0) \cup \beta$. If A and B are bug-free, they will in turn add AB later during their own execution. The condition on line 1 ensures that we avoid duplicating any state spaces with multiple data-race preemption points; for example, AB is reachable by multiple paths through its different subsets A and B, but should of course be tested only once.

Algorithm 4: Adding new jobs with data-race preemption points.

Input: j_0, the currently-running job
Input: $\mathcal{J}$, the set of all existing (or completed) jobs
Input: α, an instruction reported by the model checker as part of a data race
1 **if** $\forall j \in \mathcal{J}$, *PPSet(*$j_0$*)* $\cup \alpha \not\subseteq$ *PPSet(j)* **then**
2 | AddNewJob(PPSet(j_0) $\cup \alpha$, HeuristicPriority(α))
3 **end**
4 **if** $\forall j \in \mathcal{J}$, *PPSet(j)* $\neq \{yield, \alpha\}$ **then**
5 | AddNewJob($\{yield, \alpha\}$, HeuristicPriority(α))
6 **end**

Furthermore, Iterative Deepening allows not always strictly increasing the number of preemption points whenever a new data race is identified. For each instruction involved in a data race, Quicksand adds two new jobs: a "small" job to preempt on that instruction only (line 5), and a "big" job to preempt on that instruction as well as each preemption point used by the reporting job (line 2). Hence, each *pair* of racing accesses will spawn four new jobs. Figure 4.3 depicts the resulting overall workflow in Quicksand, includ-

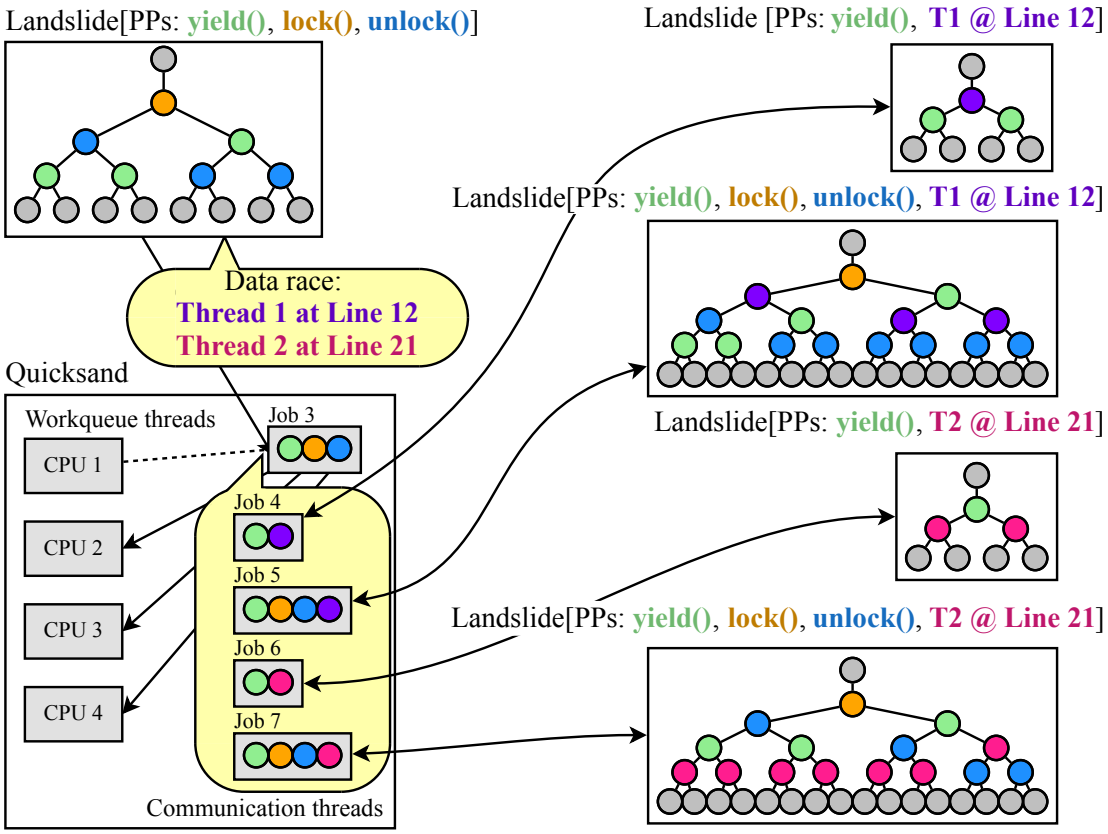


Figure 4.3: Quicksand incorporates data races as new preemption points at run-time by managing the exploration of multiple state spaces, communicating with each Landslide instance to receive ETAs, data races, and bug reports. When a data race is reported, a new preemption point is added for each involved memory access, and new jobs are added for later testing, corresponding to different combinations thereof with the existing preemption points.

ing the four such jobs resulting from one data race report.[3][4] The rationale of spawning multiple jobs is that one cannot know in advance which will be most fruitful: while the big job risks not completing in time, the small job risks missing the data race entirely if the original preemption points were required to expose it. In practice, I have observed many bugs found quickly by these small jobs, and many other bugs missed by the small jobs found eventually by the big jobs. This phenomenon motivates Iterative Deepening to prioritize jobs at run-time.

The new state spaces may expose a failure, in which case Iterative Deepening must stop and report a data-race bug, or complete successfully, indicating a *benign* (i.e., false-

[3]As an optimization, though the big jobs should be expected to uncover more data races and in turn produce even bigger jobs still, small jobs should be forbidden from "reproducing", as their purpose is only fast heuristic bug-finding rather than exhaustive coverage; see `handle_data_race()` in `messaging.c`.

[4]For visual simplicity, fake state spaces are shown here to convey only relative size differences, but not the internal asymmetric structure inherent to interleavings of multiple threads (compare to Figure 4.2).

positive) data race. They may also uncover a new data race candidate entirely in some alternate interleaving, in which case we may iteratively advance to a superset state space which will preempt at both racing access pairs. Being constrained by a CPU budget, Iterative Deepening may time out before completing a data race's associated state space, in which case the data race remains neither confirmed nor refuted. In such cases, Quicksand elects to impose some burden on the user by reporting it as a potential false positive and recommend that she investigate it by hand to judge for herself whether it be a bug. §4.6.4 will discuss future opportunities for improving debugging output in cases of such *partial verification*. However, experience shows that this interactivity pays off: in the next chapter's educational user study (§5.3), one student reported during the survey that they used this recommendation, combined with their own intuition, to find a bug that Quicksand was not able to find alone (§5.3.3).

4.2.4 Choosing the best job

With a limited CPU budget, many larger tests are likely to be fail to complete in time, and smaller tests are likely to be more fruitful at finding bugs quickly. A model checker's state space estimation (§3.4.3) can provide a hint to select between these jobs. How to handle jobs whose ETAs are too high for the given CPU budget is the heart of Iterative Deepening, and is listed formally in Algorithm 5.[5]

Its main feature is understanding that if $\text{PPSet}(j_1) \subset \text{PPSet}(j_2)$, and j_1 is suspended, then j_2's state space is guaranteed to be strictly larger, so j_2 will take at least as long. Hence, as long as j_1 is suspended on account of being too big, j_2 should not be tested either, unless j_1 is later resumed and its ETA improves over time after further execution. Similarly, whenever a job finds a bug, all pending superset jobs may safely be cancelled, as they are guaranteed to contain the same program behaviour, and likely to simply find the same bug again. Implementation-wise, Quicksand receives an updated estimate from each Landslide instance whenever it finishes executing a new interleaving, and separates them accordingly into a set of *suspended* jobs, i.e., partially-explored state spaces with high ETAs, and a set of *pending* jobs, i.e., untested ones with unknown ETAs. When Landslide reports an ETA too high for some job, it is compared with other pending and suspended jobs to find another one more likely to complete in time.[6]

Iterative Deepening also accounts for the inherent inaccuracy of ETA estimates. Line 1 heuristically scales up the time remaining to avoid suspending jobs too aggressively in case their ETAs are actually overestimated. Lines 12-15 account for the possibility that among two suspended jobs, $\text{PPSet}(j_1) \subset \text{PPSet}(j_2)$ but $\text{ETA}(j_1) > \text{ETA}(j_2)$. This may seem surprising, but can often arise because estimates tend to get more accurate over time, and j_1 perhaps ran much longer, on account of being overall smaller, before becoming

[5] Though its worst-case performance is $O(|\mathcal{P}| \times |\mathcal{S}|)$, in practice the non-constant portion beyond line 4 runs very infrequently and is negligible compared to the exponentially-sized state spaces themselves.

[6] Note that when Quicksand is configured to use multiple CPUs, simultaneously-running jobs are not considered among the set of possible jbos to switch to, so if there are fewer total jobs with ETA lower than the time budget than the allowed parallelism factor, some CPUs may end up speculatively running large jobs in hopes that the ETA turns out to be an overestimate.

Algorithm 5: Suspending exploration of a job in favor of a potentially smaller one.

Input: j_0, the currently-running job
Input: $\mathcal{P}$, the list of pending jobs, sorted by decreasing heuristic priority
Input: $\mathcal{S}$, the list of already-suspended jobs, sorted by increasing ETA
Input: T, the remaining time in the CPU budget

```
if ETA(j_0) < HeuristicETAFactor × T then
    return j_0 // Common case: job is expected to finish.
end
foreach job j_P ∈ P do
    // Don't run a pending job if a subset of it is already suspended; its ETA would
      be at least as bad.
    if ∀j_S ∈ S, PPSet(j_S) ⊄ PPSet(j_P) then
        return j_P
    end
end
foreach job j_S ∈ S do
    if PPSet(j_0) ⊄ PPSet(j_S) ∧ ETA(j_0) > ETA(j_S) then
        // If a subset of j_S is also suspended, don't run the larger one first.
        if ∀j_S2 ∈ S, PPSet(j_S2) ⊄ PPSet(j_S) then
            return j_S
        end
    end
end
return j_0 // ETA(j_0) was bad, but no other j was better.
```

suspended. In such scenarios, the algorithm heuristically assumes the smaller job's ETA is more accurate, in order to avoid repeatedly resuming larger jobs briefly only to find that their ETAs keep getting worse and worse.[7]

4.2.5 Heuristics

As predicting the ETAs of state spaces of unknown size and using that plus size of a set of preemption points as a proxy for how likely a job is to find bugs or complete is a fundamentally messy process, it is appropriate to equip the algorithm with some heuristics informed by experience. Algorithm 5 allows the option to heuristically scale a job's ETA when comparing it to the overall time budget, which can compensate for any inaccuracy by the estimator. Quicksand uses a scaling factor defaulting to 2, chosen based on experiments from prior work [SBG12]. It also includes a heuristic to never suspend jobs before they pass a certain threshold of interleavings tested, with a default of 32, informed by my personal experience that ETAs require around that much progress into the state space

[7] In order to avoid thrashing in Quicksand.

before they stabilize (at least relative to each other on similar state spaces, not necessarily relative to the ultimate true size).[8]

Landslide classifies data race candidates as *both-order* or *single-order*, as defined in prior work [KZC12], based on whether it observed the racing instructions ordered in both possible sequences or only one in the original state space, respectively. Single-order candidates are more likely to be false positives (§2.3), although preempting during the access itself is necessary to say for sure. Hence, Quicskand add preemption points for both types of candidates, and heuristically prioritizes jobs with both-order data races over those with only single-order data races. The HeuristicPriority(α) call in Algorithm 4 corresponds to this strategy. For single-order races, Quicksand does not initially add a preemption point for the later access at all: if preempting on the first access is capable of reordering the race, it will be updated to both-order in the new state space, and the second preemption point will be added then. §8.3 will discuss opportunities for future work to expand these heuristics with more nuanced search strategies still.

4.2.6 Reallocation false positives

Finally, I identified a particular class of false positive data race candidates under the Limited Happens-Before analysis (§2.3) in which the associated memory was recycled by re-allocation between the two accesses, and claim that it is safe to completely disregard them when considering where to add new preemption points. Figure 4.4 shows a common code pattern and interleaving which can expose such behaviour. If the `malloc()` on line 4 returns the same address passed to `free()` on line 2, then lines 1 and 7 will be flagged as a potential data race. I term this a *reallocation false-positive data race candidiate*. To the human eye, this is obviously a false positive: reordering lines 4-7 before lines 1-2 will cause `malloc()` to return a different region of allocated memory, in turn causing `x` and `y` to no longer collide. In studying a similar pattern, the Eraser tool from prior work [SBN+97] found that Thread 2's logic usually corresponds to an initialization pattern, but for generality I have added an arbitrary `publish` action to the example on line 6.

As long as the allocation heap is properly synchronized, a Pure Happens-Before analysis should identify a happens-before edge between line 2's `free()` and line 4's `malloc()`, and report no race. However, the upcoming evaluation will show that Limited Happens-Before retains some advantages over Pure (§4.5.2), so it is useful to automatically suppress data race candidates that are certain to end up being false positives when reordered. Such collisions could instead be avoided with a hacked allocator which never recycles memory, but that could unacceptably impact performance in `malloc()`-heavy tests.

The ability to disregard reallocation false positives is unique to Iterative Deepening. When limited to a single test execution, suppressing any data race candidate matching this pattern is unsound. Consider the more unusual program in Figure 4.5, in which the memory is recycled the same way, but the racing access's address is not tied to `malloc()`'s return value. Here, reordering lines 6-7 before line 3 will allow `x` and `x2` to race. Discard-

[8]These two heuristics are configurable with the `-e` and `-E` command-line options, respectively, as discussed in §3.1.2.

```
struct x { int foo; int baz; } *x;
struct y { int bar; } *y;
```

	Thread 1	Thread 2
1	`x->foo = ...;`	
2	`free(x);`	
3		`// x's memory reallocated`
4		`y = malloc(sizeof *y);`
5		`// ...initialize...`
6		`publish(y);`
7		`y->bar = ...;`

Figure 4.4: Reallocation false positive pattern arising from `malloc()` returning the same memory that was just `free()`d.

	Thread 1	Thread 2
1	`publish(x);`	
2	`x->foo = ...;`	
3	`free(x);`	
4		`x2 = get_published_x();`
5		`// x's memory recycled`
6		`y = malloc(sizeof *y);`
7		`x2->foo = ...;`

Figure 4.5: Reallocation-pattern data race that hides a true bug. If a single-pass Limited Happens-Before analysis discarded all data race candidates with intervening reallocations, it would miss the bug in this adversarial program.

ing the data race report as a false positive after checking just this one execution would overlook such a bug, but Iterative Deepening is guaranteed to explore the alternate interleaving, in which the true data race will show up without `free()` and `malloc()` interposing, so it is safe to suppress at first, as I will prove in §4.3.2. Moreover, in the context of Iterative Deepening, being able to discard certain data race candidates allows Quicksand to skip exploring some entire state spaces, and hence run fewer Landslides overall; this is analogous to DPOR's ability to skip equivalent interleavings within a single Landslide instance. Upcoming in the evaluation, §4.5.3's Table 4.4 will show how many redundant state spaces Quicksand is able to prune with this technique.

4.3 Soundness

Adding new data-race preemption points in a feedback loop can uncover bugs not previously reachable by preempting on synchronization APIs alone, as some prior model checkers do [SBG10], but how does it compare to the other extreme end of the trade-

off, that is, committing in advance to preempt on every single shared memory access [Hol97, YCGK08]? It turns out, assuming sufficient CPU budget, Iterative Deepening can in principle expose every possible program behaviour that even that latter approach can find, providing an equally strong verification guarantee. This section presents a proof of this claim (§4.3.1), as well as a supplementary proof (§4.3.2) of the soundness of pruning reallocation false positives discussed previously (§4.2.6).

I present these proofs as they appeared in the OOPSLA paper [BG16]: written as sketches in informal prose, to optimize for rapidly conveying an intuition for why it works rather than to justify every internal step within the proof structure.[9] A more rigorous treatment is available in the tech report which accompanied the conference paper [Blum16].

Assumptions. The proofs are built on a DPOR definition which assumes sequentially-consistent memory hardware. All algorithms involved are assumed to operate on a machine model of a single globally-consistent execution trace, which fundamentally cannot account for memory reordering nondeterminism. For existing work on combining DPOR with relaxed memory, I refer the reader to [ZKW15]. They also assume the Limited Happens-Before definition for the data race analysis. I leave the case for Pure Happens-Before to future work, although if I may appeal to intuition, it requires only to show that for any data race candidate Limited Happens-Before reports in a given execution, that Pure Happens-Before does not, either it will be a false positive, or the latter will find it in an alternate execution within the same state space, or the latter will find a different data race that ultimately leads to a bigger state space in which the first one may be found, much like a generalization of §4.3.2.

4.3.1 Convergence to total verification

The proof of Iterative Deepening's soundness is in two parts. In the first part, I prove that for any possible interleaving one could execute with preemptions anywhere, an equivalent interleaving must exist using only data-race and synchronization preemption points. In the second, I prove that starting from synchronization preemption points only, Iterative Deepening must eventually reach a state space containing such an interleaving, no matter how many data races are involved.

Equivalence

Given a preemption point p, let $\mathsf{next}(p)$ denote the next transition after p executed by the thread which ran immediately before p, let $\mathsf{instr}(p)$ denote the first instruction of $\mathsf{next}(p)$, and let $\mathsf{others}(p)$ denote the transitions by other threads between p and $\mathsf{next}(p)$.

Lemma 1 (Equivalence of non-data-race preemption points)**.** *For any thread interleaving possible by preempting on any instruction, there exists an equivalent interleaving which uses only data-race and synchronization API preemption points.*

[9] Also because this thesis is long enough already.

Proof. Let p be the first preemption point in the given interleaving such that $\mathsf{instr}(p)$ is not a data race with $\mathsf{others}(p)$ nor is a synchronization API boundary. Because $\mathsf{instr}(p)$ is not a synchronization boundary, no lock can be held during $\mathsf{others}(p)$ that was also held by the first thread across p. Hence, because $\mathsf{instr}(p)$ is not a data race, it cannot be a shared memory conflict with $\mathsf{others}(p)$ at all. Let i be the first instruction among $\mathsf{next}(p)$ which is such a conflict, or a synchronization boundary. If i is a shared memory conflict, it must be a data race, for the same reasoning as above. Modify the input interleaving by reordering $\mathsf{instr}(p)$ until i, not including i, to before $\mathsf{others}(p)$. By the soundness of DPOR (§3.4.2; [FG05]), this is equivalent to the input interleaving. In other words, p has been transformed into p' such that $\mathsf{next}(p') = i$, which is a data race or synchronization boundary. All other preemption points in the input trace can be inductively converted in the same manner. □

Saturation

For Iterative Deepening to "eventually" reach a certain state space, all data-race preemption points involved must be *reachable* during the test.

Definition 1 (Reachability)**.** *A data race candidate, and its associated preemption point(s), are reachable if it will be identified by a model checker configured to preempt only on already-reachable preemption points.*

Initially, the statically-available synchronization API preemption points (§4.2.2) are reachable. Reachability of data-race preemption points is transitive.

Lemma 2 (Saturation of data races)**.** *Given any interleaving comprising only data-race and synchronization API preemption points, all involved preemption points are reachable.*

Proof. Induct on the preemption points according to the order of their preemptions during an execution sequence. Given that the interleaving prefix preceding some point p is reachable, the proof goal is that either p be reachable, or a new data race among $\mathsf{others}(p)$, not previously reachable, be newly reachable. The latter condition suffices because in a finitely-sized codebase, there must be finitely many unique racing instruction pairs.

First, p must be "coalesced" away, as well as any other not-yet-reachable points in $\mathsf{others}(p)$. Consider the alternate interleaving in which the first thread executes past p until the first already-reachable point, then the other threads among $\mathsf{others}(p)$ execute the same way. This interleaving's preemption points are all reachable, so a state space $\mathcal{S}$ containing it will be tested.

If p is a not-yet-reachable data-race preemption point, it must be possible for some other thread to execute a data-racing instruction with $\mathsf{instr}(p)$. If this conflict was observed in the state space containing our coalesced interleaving, p is reached. Otherwise, appeal to the soundness property of DPOR: If a program behaviour is possible by interleaving threads at the boundaries of the given transitions, it will be tested in the containing state space. By contrapositive, to expose this behaviour, one or more preemptions must occur in the middle of some transition, rather than at the boundaries.

Let us now see by contradiction that there cannot be *multiple* data-race preemption points which must all be enabled before either data race can be identified; i.e., a circular dependency. Assume there does not exist a single transition $t_1 \in \mathcal{S}$ which alone can be split into $\{t_1', t_1''\}$ by a point q, such that another thread's concurrent transition t_2 conflicts with t_1''. By the soundness of DPOR, because all t_2s are independent with t_1'', $\mathcal{S} \equiv \mathcal{S} \cup q$. Replacing $\mathcal{S}$ with $\mathcal{S} \cup q$ in the above assumption shows that no *pair* of new qs would expose new program behaviour, and inductively, no set of qs of any size, which contradicts the previous paragraph. Hence, a single new not-yet-reachable data race is reachable in $\mathcal{S}$. Hence p will be reached. □

Convergence

I name the overall soundness property "convergence" in reference to the way it must eventually arrive, after potentially many iterated state spaces, at full verification strength.

Theorem 1 (Convergence)**.** *If a bug can be exposed by any thread interleaving possible by preempting on all instructions during a specific test, Iterative Deepening will eventually test an equivalent interleaving which exposes the same bug.*

Proof. For any possible interleaving, Lemma 1 provides an equivalent one with only data-race and synchronization preemption points, and Lemma 2 proves all involved preemption points are reachable. Hence, Iterative Deepening will eventually test a state space containing the equivalent buggy interleaving. □

And thus Iterative Deepening is sound.

4.3.2 Suppressing reallocation false positives

Next I prove that §4.2.6's optimization of discarding reallocation false positives is sound under Iterative Deepening.

Theorem 2 (Soundness of eliminating reallocation data race candidates)**.** *If a reallocation candidate is not a false positive, DPOR will reorder threads such that either the accesses can race without fitting the reallocation pattern, or a use-after-free bug will be reported immediately.*

Proof. Any such program must contain an access a_1 by one thread T1, followed by a `free()` and a `malloc()` possibly by either thread, followed by an access a_2 by the other thread T2, not depending on the result of the middle malloc. Without loss of generality, fix T1 to perform the `free()` and T2 the subsequent `malloc()`, as shown in Figure 4.5. The other cases are similar, although note that if T2 performs the `free()`, and the `malloc()` is reordered before it, T2's final access will be a use-after-free immediately. Let us also assume the only way for the program to get pointers to heap memory is through `malloc()`; hence, there must also be some "publish" action p by T1 which communicates the address to T2. Because this is a true potential data race, p must occur before a_1, as a_2 cannot be reordered before p.

The proof goal is that a preemption point be identified during T1 between p and a_1. The publish action must involve some thread communication, whether through a shared data structure or message-passing API. If locking or message-passing is used, the set of static synchronization preemption points (§4.2.2) suffices to provide one. Otherwise, p (and the corresponding read by T2) will be a potential data race, although that may itself be another reallocation candidate. In this case, apply induction on the chain of pointers, if any, leading to the shared address containing p: in the base case, p is communicated via global data or message-passing, and in the inductive step, DPOR will reorder threads sufficiently to identify a preemption point on p. Note that this induction may result in several possible intermediate preemption points, each requiring a new state space to be tested, of course, Theorem 1 guarantees this under Iterative Deepening. Hence there will be a preemption point between p and a_1 no matter the mode of communication.

With this preemption point, DPOR will reorder a_2 before a_1, while not changing a_2's location. As T2's `malloc()` now occurs before T1's `free()`, it will allocate different memory. Hence a_1 and a_2 can race without appearing to fall under the reallocation pattern. □

This spells QED so we are done [Var07]. Note that this proof does not rely on the existence of preemption points on the internal lock of `malloc()` or `free()`, which is an ideal candidate to ignore via `without_function` (§3.4.1) to reduce state space size. Future work may generalize this proof structure to not rely on specific knowledge of `malloc()`'s and `free()`'s behaviour, but instead to require only any intervening synchronization event, thereby extending the overall soundness proof to accomodate Pure Happens-Before as well as Limited Happens-Before. The experiments in future chapters of this thesis will assume that this holds.

4.4 Implementation

Quicksand is an independent program that wraps the execution of several stateless model checker instances. It is expected that Landslide be this checker, but any other tool which implements the same messaging interface would be compatible as well. The implementation is roughly 3000 lines of C. All source files mentioned in this section live in the `id/` subdirectory of the Landslide repository, with the exception of the Landslide extensions (listed last). As Chapter 3 was in some sense a developer's guide to Landslide, this section will serve similarly for Quicksand.

4.4.1 User interface

The available command-line options for configuring its CPU-time budget, exploration modes, and so on are listed in §3.1.2. Additionally, Quicksand periodically issues a progress report at fixed intervals to inform the user on the completion, bug-finding, and/or estimated progress of each job. Figure 4.6 shows an example. I highlight a few notable features of the jobs therein to serve as a concrete example that may cement the reader's intuition of §4.2's more abstract algorithm descriptions:

```
================================== PROGRESS REPORT ==================================
total time elapsed: 2m 52s
[JOB 0] COMPLETE (4 interleavings tested; 7s elapsed)
    PPs: { }
[JOB 1] Running (21.932870%; ETA 8m 14s)
     PPs: { 'mutex_lock' }
[JOB 2] BUG FOUND: landslide-trace-1544661430.29.html (51 interleavings tested)
     PPs: { 'mutex_unlock' }
[JOB 3] Running (9.852431%; ETA 24m 32s)
     PPs: { 'mutex_lock' 'mutex_unlock' }
[JOB 4] COMPLETE (6 interleavings tested; 9s elapsed)
    PPs: { 'data race @ 0x102917' }
[JOB 5] Running (3.710938%; ETA 1h 2m 14s)
     PPs: { 'mutex_lock' 'mutex_unlock' 'data race @ 0x102917' }
[JOB 6] COMPLETE (4 interleavings tested; 8s elapsed)
    PPs: { 'data race @ 0x1000ecf' }
[JOB 7] Running (6.119792%; ETA 33m 14s)
     PPs: { 'mutex_lock' 'mutex_unlock' 'data race @ 0x1000ecf' }
[JOB 11] Running (3.670247%; ETA 50m 16s)
     PPs: { 'mutex_lock' 'mutex_unlock' 'data race @ 0x102917' 'data race @ 0x1000ecf' }
[JOB 8] Deferred... (33.340567%; ETA 2h 6m 3s)
     PPs: { 'mutex_unlock' 'data race @ 0x102917' }
[JOB 9] Deferred... (34.466226%; ETA 2h 35m 37s)
     PPs: { 'mutex_unlock' 'data race @ 0x1000ecf' }
[JOB 10] Deferred... (11.113790%; ETA 4h 20m 31s)
     PPs: { 'mutex_lock' 'data race @ 0x102917' }
=====================================================================================
```

Figure 4.6: Example Quicksand progress report showing the various possible job states.

- Jobs 0, 1, 2, and 3 are the initially-seeded state spaces (§4.2.2).
- Job 2 reports a bug found, and shows the filename of the HTML preemption trace (§3.1.5, Figure 3.3) which the user should examine to diagnose it.[10]
- Jobs 4 and 6 are the "small" jobs added to test the two data races in isolation; 5 and 7 are the corresponding "large" jobs (§4.2.3).
- Job 11 is the maximal state space, containing all synchronization preemption points and both currently-known data races.
- The intermediate jobs 8, 9, and 10 have been suspended for having ETAs at least twice as large as the provided CPU budget (1 hour), according to the ETA scaling factor heuristic (§4.2.5).
- Note that job 11's ETA is currently lower than 8's, 9's, and 10's, despite being a strict superset of each. This corresponds to the ETA inversion situation discussed in §4.2.4: Quicksand simply hasn't made as much progress into job 11 (compare their percentage estimates rather than ETAs) for its ETA to be accurate enough yet.

[10] I actually cheated by copy/pasting this job alone from a different run of Quicksand; the other jobs come from a test with the bug already fixed, in order that exploration progress far enough to defer some jobs for the sake of example. In a real execution, the superset jobs 3, 5, 7, 8, 9, and 11 would be cancelled.

Future work could extend these progress reports to be more interactive, allowing the user to reprioritize state spaces at her whim or to disable certain data-race preemption points after checking them by hand, as discussed in §8.1.

Besides the progress reports, Quicksand also prints a notice for each new data race that Landslide detects, like so, corresponding to the above progress report, for example:

```
Found a racy access at 0x00102917 in deschedule <unknown>
Found a racy access at 0x01000ecf in cond_signal (cond.c:101)
```

If using Limited Happens-Before instead of Pure, it prints "potentially-racy access" instead. This is implemented in `pp_new()` in `pp.c`. If the CPU budget runs out and Quicksand must stop exploring prematurely (or the user's patience runs out and she interrupts it with ctrl-C), it prints a final report of all data races it was not able to finish classifying as either buggy or benign, and urges the user to finish checking them with visual inspection (`print_live_data_race_pps()` in `pp.c`). It is this feature which one respondent in the student user survey (§5.3.3) credited for finding an extra bug.

If the verbose option (`-v`) is supplied, Quicksand will also print one line per interleaving tested by all its Landslide instances, showing the number of branches tested so far, the estimated percent progress, and the ETA, as shown earlier in §3.4.3. This produces a lot more output, and can make progress reports hard to read as they scroll off the screen quickly, but the author personally finds this mode less disorienting than ten seconds of pure silence between each progress report. Of course, future work could improve this with a GUI, or at least a split-screen console view. It will also cause the progress reports to report more detailed information, such as which preemption points are nondeterministic data races (§4.1.1) and number of reallocation false positives suppressed (§4.2.6).

4.4.2 Model checker interface

The interface with the model checker has two parts. First, when starting each job, Quicksand creates a configuration file declaring which preemption points to use, plus other test-case-specific options such as which preemption points to suppress (§3.4.1), especially those arising from the `malloc()` lock (§4.3.2), which functions DPOR and data race analysis should treat as trusted code (§3.2.1), whether to enable mutex-testing mode (§3.1.3), transactional memory (§6.2), and so on. This is done by `run_job()` in `job.c`.

Then, a dedicated Quicksand thread (`start_job()` in `job.c`) communicates with its corresponding model checker process via message-passing over a FIFO pipe (`talk_to_child()` in `messaging.c`). Landslide messages after testing each interleaving to report updated progress and ETAs as well as whenever a new data race candidate or bug is found. Quicksand in turn replies whether to resume/suspend (due to too high ETA) or quit (due to timeout) (`handle_should_continue()` in `messaging.c`). It suspends jobs simply by making Landslide wait on a message-passing reply. Should Quicksand later re-schedule a suspended job, it sends a message to continue, resuming Landslide right where it left off; otherwise, it wakes it up only after time runs out, causing it to exit immediately. The message-passing format is defined at the top of `messaging.c`, and a matching definition appears in Landslide's source file of the same name.

4.4.3 Architecture

Quicksand's overall architecture is a thread pool of workers, one for each CPU it was configured to use with `-c` (§3.1.2). These threads do not correspond directly to each active instance of Landslide, as some may be deferred; rather, each worker thread links up temporarily to a job thread, whose duties the previous paragraph describes, and process them as the overall work-queue of state spaces to be tested demands. Following is a brief description of each of Quicksand's major modules.

- **Job management** (`job.c`): Contains the lifecycle code for job threads, generation of Landslide configuration files, and Linux process management code to launch child Landslide instances (`run_job()`).
- **Messaging** (`messaging.c`): Manages communication with child Landslides (`talk_to_child()`, implementing certain aspects of Iterative Deepening, creating new jobs in response to data race reports (`handle_data_race()`) and deferring too-big jobs in reponse to ETA updates (`handle_estimate()`).
- **Preemption point registry** (`pp.c`): Tracks the set of known synchronization primitives (initialized by main) and data races (`pp_new()`), including set comparison routines (`pp_subset()`) and computing a job's priority based on the types of included preemption points (`unexplored_priority()`).
- **Workqueue** (`work.c`): Implements the per-CPU worker threads, including the check for whether to switch priority from one job to another (Algorithm 5, `should_work_block()` and `get_job()`), as well as managing the shared workqueue of jobs overall (`workqueue_thread()`). Also implements the fixed-interval progress reporting (`progress_report_thread()`).
- **Bugs** (`bug.c`): Tracks a list of found bugs for main to repeat at program exit, and implements the check for superset state spaces to be cancelled if a subset already found a bug (`bug_already_found()`).
- **Options** (`option.c`): Processes command-line options, checking for legality of various combinations of exploration modes. New options may be added with the convenient macros `DEF_CMDLINE_FLAG()` and `DEF_CMDLINE_OPTION()`.
- **Main** (`main.c`): Initializes the default state spaces, waits for worker threads to terminate after either completion or time-out, and issues a final list of bug reports, data race reports, or congratulations as appropriate.

Finally, because in very large tests, the number of suspended Landslide instances may grow without abatement, Quicksand checks every progress report interval whether the memory footprint of these inactive Landslides pose a threat to the system's total memory. Implemented in `cant_swap()` [ED09] in `work.c`, it checks if the system's memory usage exceeds a fixed percentage of its total (`RAM_USAGE_DANGERZONE`, default 90), and if so, abandons a fixed percentage of suspended Landslides (`KILL_DEFERRED_JOBS`, default 50). Generally, the currently-running Landslide instances should never threaten to hit swap, as there can only be as many of those as CPUs, but this also accounts for memory used by other processes, so this is not guaranteed to avoid swapping on a heavily-stressed system

(such as running multiple Quicksands at once). Naturally, if Quicksand ever needs to invoke this protocol, any hope at a total verification is compromised.

4.4.4 Exploration modes

In addition to Iterative Deepening, which Quicksand defaults to if no options are given to specify otherwise, Quicksand also supports several alternate exploration strategies, as follows.

- **Single state space, basic DPOR** (`-C`): Runs a single instance of Landslide configured to preempt on all statically-known synchronization preemption points. Corresponds to dBug's approach [SBG10]. Never adds any new preemption points based on data race reports.
- **Single state space, ICB** (`-I`): Runs a single instance of Landslide with preemption points as above, but running Iterative Context Bounding with BPOR (§3.4.5) instead of plain DPOR. Corresponds to CHESS's approach [MQB+08]. Requires either `-C` or `-M` (see below).
- **Single state space, preempt-everywhere** (`-0`): Runs a single instance of Landslide as above, but preempting on every shared memory access, not just synchronization. Corresponds to the approach of SPIN [Hol97] and Inspect [YCGK08]; CHESS supports this mode as well with optional compiler instrumentation. Requires `-C`; may optionally be combined with `-I`.
- **Maximal state space mode** (`-M`): Runs the naïve version of Iterative Deepening shown in Algorithm 3, i.e., immediately abandons any state space whenever a superset of it exists. This results in always testing the maximal state space only, with no inherent parallelism, and optimizes for the fastest verification when the user has reason to believe no bugs will exist. No prior work implements this approach. Note that this mode was implemented after [BG16]'s publication, and I will feature it in the evaluation of transactional memory (§6.3) rather than in this chapter.

Quicksand restricts ICB to be usable only in modes when it runs only one Landslide at a time. ICB is itself a heuristic search ordering strategy to uncover bugs faster, so while technically easy to run Iterative Deepening with all jobs thereunder running ICB, that would suffer both approaches' repeated work compounded. §8.3 discusses integrating the two approaches to hopefully reap the benefits of both. However, maximal state space mode does support ICB, as it focuses on verification only, but if the result is a time-out, the user may find an ICB-style preemption-bounded partial verification useful.

4.4.5 Landslide extensions

I have added several features to Landslide specifically for use under Quicksand. Source files mentioned in this subsection live under the usual Landslide source directory.

The other end of the messaging protocol (§4.4.2) is implemented in `messaging.c`. When Quicksand suspends Landslide, it detects how much time it spent asleep, and cor-

rects for that amount during its next ETA computation (`fudge_time()` in `estimate.c`). Landslide's data race analysis also includes a heuristic to avoid reporting "too suspicious" data race candidates which it believes arise from the initialization pattern [SBN+97]: if a conflicting access pair is single-order (§4.2.5) and also arose during a known synchronization API's `init()` or `destroy()` function, Landslide will not message it to Quicksand, at least not until it is reclassified as both-order.

To recognize the reallocation pattern discussed in §4.2.6 during data race analysis, Landslide includes a generation counter in its heap allocation tracking (§3.3.3). Each heap allocation is given a unique ID, and when evaluating whether two heap accesses can race, the IDs of their containing blocks must match (`was_freed_remalloced()` in `memory.c`), in addition to the other requirements of Happens-Before. If the generations do not match, Landslide sets the `free_re_malloc` flag in the messaging protocol to Quicksand. If the race is later observed in a reordering which avoids the reallocation pattern (such as in Figure 4.5), Landslide will report it as normal, and Quicksand will promote it to a normal preemption point in the registry (`pp_new()` in `pp.c`, "`for realsies`" case). Also included in this message is a flag to indicate whether a data race was found nondeterministically (i.e., not on the first interleaving), such as described in §4.1.1.

Preempt-everywhere mode (§4.4.4) imposes a heavy burden on Landslide on account of the sheer number of preemption points involved. First of all, because there are separate tracing entrypoints for memory accesses and instructions (`instrument.c`), it cannot simply invoke the checkpointing routine (§3.3.5) immediately. Also, we must still exclude thread-local and kernel (if testing userspace) or user (if testing kernelspace) accesses. Rather, the memory analysis (§3.3.3) invokes `maybe_preempt_here()` in `pp.c` for every access it would ordinary record for DPOR. If the access is outside of the current stack frame, and not part of the mutexes (unless `TESTING_MUTEXES`), this sets a scheduler action flag `preempt_for_shm_here` which makes preemption point identification treat it the same as a data race (§3.4.1). `check_withins()` is also modified to never switch to allowlist mode. Finally, Landslide increases its heuristic constant for infinite loop detection (§3.4.6) on the first interleaving from 4000 to 2^{20}, to account for the increased orders of magnitude in preemptible events.

4.5 Evaluation

In Quicksand, Iterative Deepening and data race analysis are intimately connected: the former relies on the later to supply it with new preemption points, thereby refining its search for new concurrent behaviours, while the latter relies on the former to thoroughly check all possible interleavings around its reported memory accesses and classify them as buggy or benign. Despite this synergy, which is necessary for total verification soundness (§4.3), each of these two techniques is a contribution in its own right when it comes to bug-finding performance. Hence, this evaluation will measure not only the combined approach's full verification power, but also the bug-finding performance of each technique separately, as compared to state-of-the-art single-state-space ICB and DPOR. I pose the following evaluation questions.

1. Does integrating data-race preemption points improve the accuracy of model checking?
   (a) Do data-race preemption points expose new bugs that couldn't be found with synchronization ones alone?
   (b) Do data-race preemption points expose the same bugs the preempt-everywhere approach could find, faster?
   (c) Does Quicksand provide more full verifications more quickly than the preempt-everywhere approach?
   (d) How does the choice between Pure and Limited Happens-Before affect bug-finding and verification performance?
2. Does testing alternate interleavings with model checking improve the accuracy of data race analysis?
   (a) Does Quicksand avoid false positives compared to single-execution Limited Happens-Before?
   (b) Does Quicksand find data-race bugs that single-execution Pure Happens-Before or Limited Happens-Before alone would miss?

4.5.1 Experimental setup

The test suite consists of 79 P2 student thread libraries (§2.4.1), submitted in 15-410 during the Spring 2014, Fall 2014, and Spring 2015 semesters,[11] and 78 Pintos student kernels (§2.4.2), submitted in Berkeley's CS162 and U. Chicago's CMSC 23000 during Spring 2015. The P2s in this dataset average 1807 lines of C and x86 assembly code, and the Pintoses average 718 lines (by `diff` to the provided basecode), for a total of 198,772 lines of code tested for this evaluation.

I chose P2s and Pintoses for this test suite because of the relative ease of generating hundreds of unique state spaces, varied in size and correctness, and with a diverse set of bug types.[12] While many prior work stateless model checking papers [MQ07, AAJS14, Hua15, KLSV17] publish studies of single-digit or low-double-digit numbers of bugs found in "real-world" programs, sometimes reported to and confirmed by the upstream developers, to motivate stateless model checking to be used in production settings, I believe this approach to be too anecdotal for comparing several model checking strategies *against each other*, and opt for this approach instead for better statistical significance.[13]

[11] The 2014 semesters were before Chapter 5's user study experiments, and for Spring 2015 (the first semester thereof), students who used Landslide during the project were excluded from this dataset.

[12] In addition to concurrency bugs, many of the codebases exhibited *deterministic* bugs (i.e., encountered on the first interleaving tested), which I fixed by hand before running these tests to ensure that every bug in this study required meaningful work by the model checker.

[13] Not to mention – as I couldn't say in a conference paper, but can say now – that extending Landslide to support native Linux programs, complete with filesystem and network nondeterminism, would have been an engineering burden beyond my ability to do alone and still graduate on time.

Test cases

I tested P2s with six multithreaded programs: `mutex_test`, for locking algorithm correctness, `thr_exit_join`, a test of thread lifecycle, `broadcast_test` and `paraguay` for condition variables, `paradise_lost` for semaphores, and `rwlock_downgrade_read_test` for R/W locks. These are the same tests I distributed Landslide with in Chapter 5; §5.1.3 describes them in further detail. For `mutex_test`, `paradise_lost`, and `paraguay`, I used the `without_function` command to exclude `thr_create()`, `thr_exit()`, and `thr_join()` preemption points, and for `mutex_test` I enabled `TESTING_MUTEXES` (§3.1.3). I tested Pintoses with three programs from the class's provided test suite: `priority-sema`, a test of the kernel scheduling algorithm, `alarm-simultaneous`, for the timer sleep routine, and `wait-test`, for process lifecycle system calls. These are a subset of those used in Chapter 5; see §5.2.3. Some of the Pintoses were partially implemented, so each test could only be run on a subset of the 78 submissions; see the "Number tested" column in Table 4.1. For all tests, I also excluded preemption points on `malloc()`'s internal lock using `without_function`. In total, the evaluation comprises 629 unique tests (i.e., pairs of a test program and a Pintos or P2), at least 181 of which will be seen to expose bugs.

Model checker configuration

To evaluate the benefits of data-race preemption points and Iterative Deepening separately, I ran the test suite under Quicksand in three different experimental configurations, each of which was given a 1-hour budget and 10 CPUs for each test.

- **QS-Limited-HB**: Quicksand with Landslide configured to use Limited Happens-Before for its data race analysis (`-H`),
- **QS-Pure-HB**: Quicksand using Pure Happens-Before instead (`-V`), and
- **QS-Sync-Only**: Quicksand with initial preemption points only, as described in §4.2.2, but never adding new ones from reported data races.

I represented the MC State of the Art[14] with three configurations of stand-alone Landslide on the same test suite, corresponding to the search strategies discussed in §4.4.4 and §4.4.5.

- **SSS-MC-DPOR**: Single state space mode (`-C`) using the maximal preemption point set from §4.2.2, explored with DPOR (§3.4.2),
- **SSS-MC-ICB**: With preemption points as above, but instead using ICB [MQ07] with BPOR [CMM13] to find bugs faster (`-I`, §3.4.5), and
- **SSS-MC-Shared-Mem**: Using ICB+BPOR, configured to preempt on any shared memory access (`-0`) (decided at runtime, excluding threads' accesses to their own stacks), which in principle includes all possible data races.

Prior work has shown how to parallelize DPOR of a single state space across multiple processors [SBGH12], but it remains an open research problem how to extend the algorithm to ICB. Hence, I optimistically gave all control experiments a linear speedup of 10

[14]The author's DJ name.

hours per test with 1 CPU; i.e., assuming it could parallelize with 100% efficiency. To match this, Quicksand reports both the CPU time and wall-clock time spent during its execution. Comparing CPU time leads to a more fair comparison, although Quicksand's inherent parallelism, which only a wall-clock time comparison would show, is also a convenient benefit unto itself. All tests ran on 12-core 3.2 GHz Xeon W3670 machines with 12GB of RAM.

4.5.2 Bug-finding

Figure 4.7 plots the bug-finding performance of Quicksand's three experimental trials against the three control approaches in a cumulative distribution of total bugs found against elapsed CPU time. The farthest-right point on each series indicates in how many total test cases that trial found a bug after the 10 CPU-hour timeout. Figure 4.8 shows the same experiments, measured by wall-clock time until each bug was found instead.

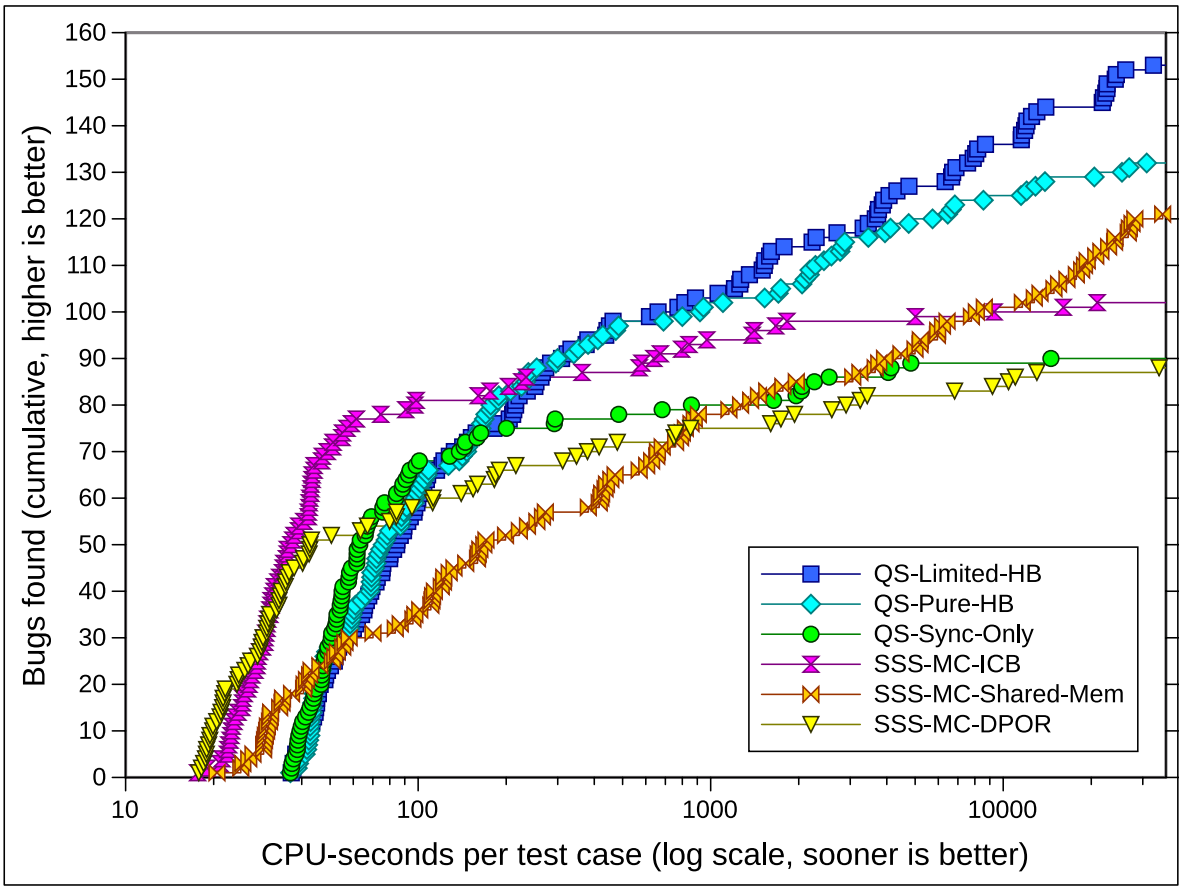


Figure 4.7: Quicksand's bug-finding performance measured in CPU time. Quicksand finds 125% as many bugs with data-race preemption points at the 10-hour mark, compared to the best prior work approach. Quicksand's startup overhead is exaggerated, as the control experiments are not parallelized.

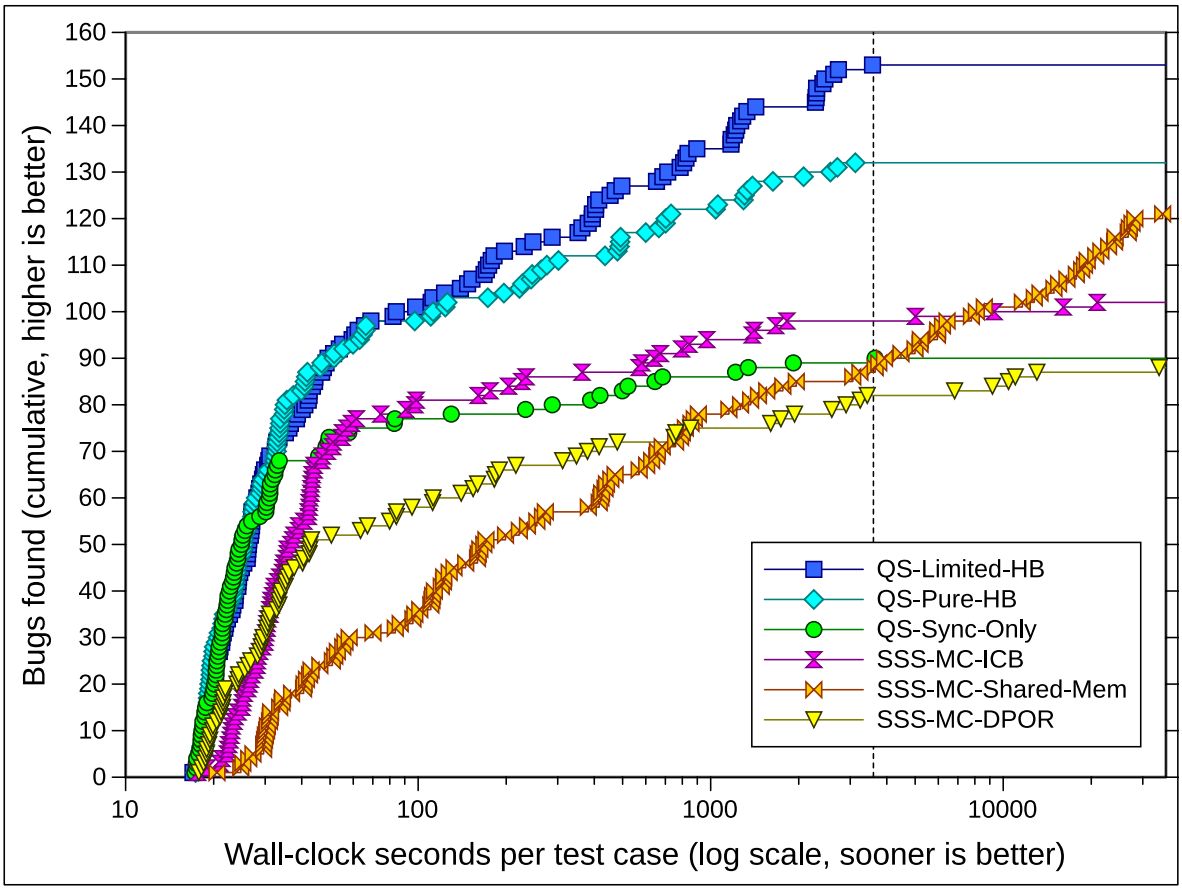


Figure 4.8: Quicksand's bug-finding performance measured in wall-clock time. Quicksand is parallelized tenfold; the vertical line indicates its 1 hour limit.

Performance comparison

Compared to SSS-MC-ICB (the fastest among the control experiments), Quicksand finds more bugs within any fixed CPU budget greater than 200 seconds. In other words, draw a vertical line at $x = N$ for any $N > 200$ to represent timing out each test after y seconds elapsed, and Quicksand's bug total will exceed that of ICB. SSS-MC-Shared-Mem initially suffers a substantial performance penalty from the sheer number of preemption points it must analyze, but ultimately outstrips SSS-MC-ICB, which fundamentally cannot find data-race bugs, after 135 CPU-minutes with its 100th bug found, ultimately finishing the 10 CPU-hours as the best prior work approach in the long term. Compared to SSS-MC-Shared-Mem, Quicksand's Limited HB version finishes with 125% as many bugs in total.

Regarding Quicksand's tenfold parallelism, before the break-even point at 200 seconds, it lags behind SSS-MC-ICB due to the additional start-up overhead of testing many state spaces at once even though the easy bugs may be found extremely quickly in any of them. However, converting ICB's early CPU-time advantage into faster wall-clock performance remains an open research problem [SBGH12]. Figure 4.8 gives Quicksand full credit for its inherent parallelism, which ICB cannot yet practically match: with a proces-

sor allocation of 10 CPUs, it outperforms all prior work approaches for any fixed budget of wall-clock time (i.e., comparing across a vertical line at $x = N$ for any N).

The QS-Sync-Only experiment tests whether Iterative Deepening would be effective even for model checking domains without data races, such as distributed systems [KAJV07, YCW+09, LHJ+14, SBGH11, CGS13] and programming languages whose type systems statically reject concurrent mutable shared state [VWW96, HPJW+92, KN18]. When Quicksand ignores all data race candidates, its results are competitive with SSS-MC-DPOR, although SSS-MC-ICB outperforms it slightly. This is unsurprising: the seed subsets of preemption points that QS-Sync-Only is limited to (§4.2.2) are much less flexible than ICB's preemption strategy. This result suggests that in future work, Quicksand should consider using ICB in parallel with its default configuration when it finds no data race candidates to test. I discuss this possibility further in §8.3.

On the other hand, comparing QS-Limited-HB to SSS-MC-Shared-Mem shows that Iterative Deepening thoroughly outperforms ICB when shared-memory preemptions come into play. Statically configuring a preemption point for every shared memory access in advance produces orders of magnitude more points than waiting for an access to be identified as part of a data race at runtime. In principle, DPOR and ICB+BPOR should suffice to identify and prune any equivalent thread interleavings arising from extraneous preemption points on non-conflicting accesses. However, in practice, the sheer number of accesses during each new execution (often thousands) added significant performance overhead to DPOR's $O(n^2)$ memory independence computation (§3.4.2), as well as the $O(n)$ overhead of checkpointing the execution state at each preemption point (§3.3.5). Iterative Deepening avoids this overhead by waiting until runtime to identify fewer, more relevant preemption points dynamically, and is hence more suitable for model checking when data races are involved.

Types of bugs

Table 4.1 provides more detail on each of the bugs shown in Figure 4.7, broken down by test case. The left half shows the number found by each experimental approach, with the totals of each column corresponding to the values at $x = 10$ hours in Figure 4.7. In `mutex_test`, which checks the lock implementation for correctly providing mutual exclusion (rather than trusting its correctness, as all other tests do), SSS-MC-ICB and SSS-MC-DPOR found dramatically fewer bugs (just 1). Prior work has proposed *abstraction reduction* [Sim13], in which verifying correctness properties of synchronization primitives allows subsequently trusting them in other tests which use them to mitigate state space explosion; §6.3.3 will explore this technique further. By contrast, QS-Limited-HB found 10 mutex bugs, and SSS-MC-Shared-Mem found 12. In the scope of this chapter, this serves as strong evidence that new low-level synchronization code must be verified with data-race preemption points, whether combined with Iterative Deepening or ICB.

To ensure that the corpus of P2 and Pintos bugs gives an unbiased comparison between Quicksand and ICB, I also counted the preemption bounds at which ICB found each of its bugs, i.e., the minimum number of involuntary thread switches each bug required to expose. Table 4.2 shows the distribution of these bounds, which is consistent with

Test	Num. tested	Quicksand LHB	Quicksand PHB	Single-state-space MC ICB	Single-state-space MC DPOR	Single-state-space MC ShMem
`broadcast_test`	79	8	8	5	6	7
`thr_exit_join`	79	23	20	13	13	14
`mutex_test`	79	10	9	1	1	12
`paradise_lost`	79	17	16	12	11	12
`paraguay`	79	10	8	5	5	11
`rwlock_downgrade`	79	27	26	25	23	28
`priority-sema`	59	7	7	1	1	8
`alarm-simultaneous`	44	21	12	16	5	29
`wait-simple`	52	30	26	24	23	1
Total	629	153	132	102	88	122

Table 4.1: Summary of bugs found by each test program. QS-LHB and QS-PHB are Quicksand; ICB/DPOR/ShMem are the controls (§4.5.1).

Bound	SSS-MC-ICB	SSS-MC-Shared-Mem	Prior work ICB [MQ07]
0	2	1	3
1	82	86	7
2	16	32	5
3	2	3	1
4+	0	0	0
Total	102	122	16

Table 4.2: Distribution of preemption bounds among bugs found by ICB control experiments. Bound 0 means the bug was found by switching threads only on `yield` calls.

the results of [MQ07, Table 2], reproduced in the rightmost column (obtained under a different test suite, of course, of only 5 programs). This shows no bias towards bugs that would be harder for ICB to find. In fact, this evaluation's preemption bound distribution is *more* heavily biased towards fewer preemptions, suggesting that if anything, my test suite is even friendlier still to ICB than that of prior work.

4.5.3 Verification

The previous section showed that Quicksand's suite of bug-finding heuristics, built around Iterative Deepening, outperform the best single-state-space approaches, even after correcting for its inherent parallelism. This section will hold Quicksand to its promise to uphold the other side of the trade-off as well: that it reach full verification on correct tests reasonably quickly.

Full verification

Figure 4.9 plots the cumulative distribution of total verifications provided by each approach, in the same style of graph as Figure 4.7 and Figure 4.8. For 167 of the 629

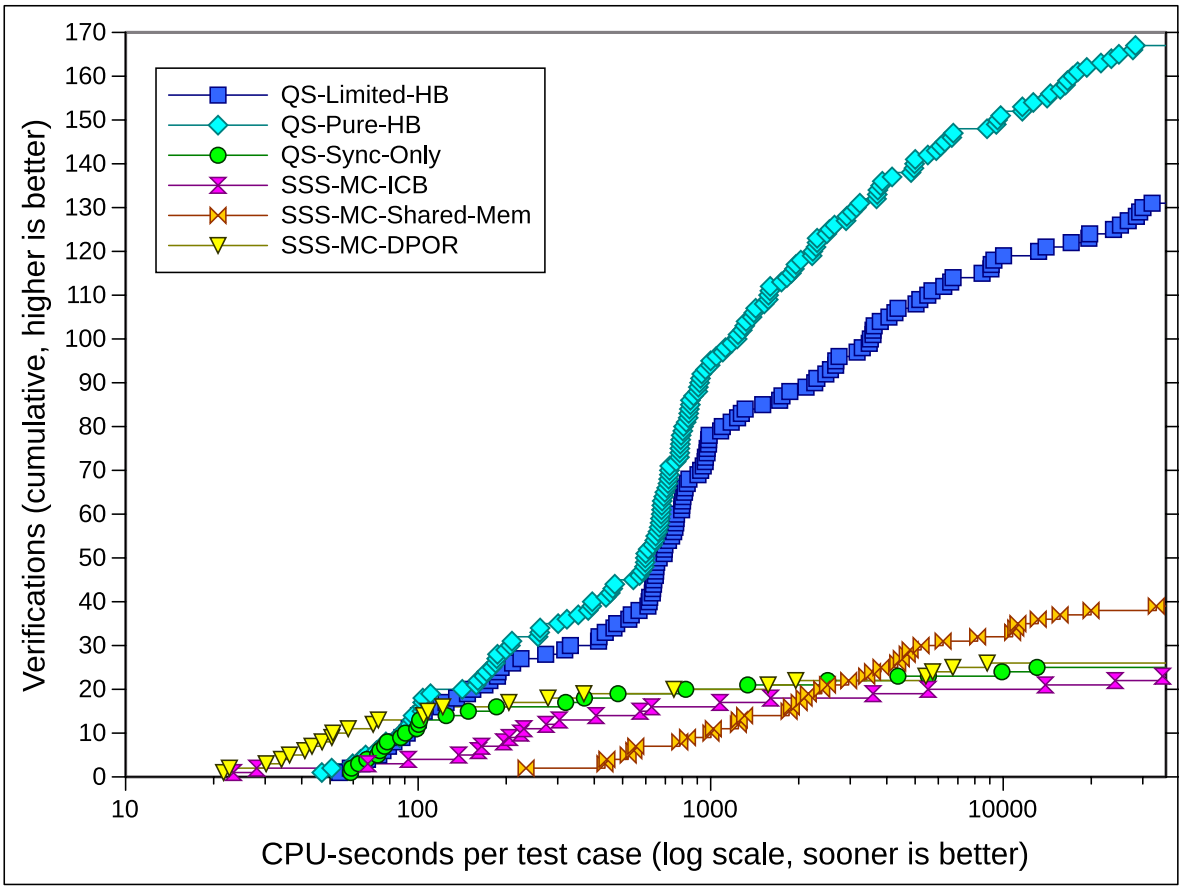


Figure 4.9: Verification performance comparison between Quicksand and single-state-space approaches. Among the latter, only SSS-MC-Shared-Mem is theoretically capable of verifying any test with data races; the others' series include only tests with no data races whatsoever, in which case synchronization preemption points alone suffice for a full verification.

tests, QS-Pure-HB was able to reach and complete the maximal state space with no bugs found, hence providing the total verification guarantee justified by the proofs in §4.3. QS-Limited-HB completed a verification for 153 of 629 tests, slightly slower on account of Limited Happens-Before's higher false positive rate. The next best approach for verifications was SSS-MC-Shared-Mem, which completed its search in only 39 cases.

Ultimately, using Limited Happens-Before for finding data race candidates allowed Quicksand to find more bugs, while Pure Happens-Before allowed for reaching full verification faster. I attribute this trade-off to the fact that Limited Happens-Before need not wait to test many alternate thread interleavings before finding a data race candidate to begin with; rather, it can add new jobs to start testing potential races immediately.[15] On the other hand, Limited Happens-Before can get overwhelmed by too many false positives,

[15]Upcoming, Table 4.4 will corroborate this conclusion: the difference is most dramatic in `alarm-simultaneous`, the test where Quicksand struggled most to finish even small subset jobs.

needing to refute such candidates by testing new state spaces for each one, while Pure Happens-Before can refute false positives *en passant* by testing alternate interleavings in its original state spaces. This suggests that model checkers which incorporate data race analysis should implement both modes and offer the user to choose based on their desired and/or expected testing outcome.

The only testing modes which are theoretically capable of verifying any test with data races were QS-Pure-HB, QS-Limited-HB, and SSS-MC-Shared-Mem (i.e., preempt-everywhere mode). When QS-Sync-Only, SSS-MC-DPOR, and SSS-MC-ICB complete their respective maximal state spaces (i.e., all synchronization preemption points), that constitutes a full verification only in the case where no data races were identified at all, meaning Iterative Deepening would search no deeper than that anyway. Therefore, in Figure 4.9, the data series for these latter three configurations represent only completed tests with no data races. Even though SSS-MC-Shared-Mem tends to hang out in the same neighbourhood as them, note that SSS-MC-Shared-Mem is still steadily increasing in verifications provided at the 10-hour cutoff (let alone the Quicksand ones), while the other three seem to reach a plateau of around 20-30 tests relatively soon.

A single-state-space model checker could rely on the user to properly synchronize all reported data races, in accordance with the philosophy that even non-failing races should count as bugs [Boe11, Boe12], ultimately improving the number of tests it can verify with no data races. However, RacerX [EA03] showed that overwhelming the user with warnings about non-failing behaviours jeopardizes their patience for the tool, which motivates Quicksand to follow in the footsteps of Portend [KZC12] instead.

Overall, including data-race preemption points increases verification capacity by 4.25x. Assuming sequentially-consistent hardware, QS-Pure-HB classified many true data races as benign, while the SSS-MC-ICB approach could at best report such races to the user. This graph's results show that code written in a natural environment by inexpert users (students) generally does not obey the sort of strict coding discipline necessary for a model checker to make simplifying assumptions such as "no data races", justifying this chapter's claim that data-race preemption points are essential to model checking.

Partial verification

When a model checking job times out, the user would more likely prefer a summary of what parts of the test were verified rather than to write off all the CPU time spent as wasted. To this end, Quicksand reports which subsets of preemption points resulted in state spaces that did complete in time, in hopes that the user can supplement such a result with her own intuition by inspecting the code corresponding to the preemption points not tested (especially data races). From prior work, Preemption Sealing [BBC+10] has argued the value of similar *compositional testing* when full verification is intractable, deferring to the user's expertise to judge the value of each subset of preemption points verified. Table 4.3 shows Quicksand's partial verification results on timed-out tests.

On 229 tests, SSS-MC-ICB timed out after 10 hours with no bugs found. Among these tests, QS-Limited-HB found bugs in 37. The other 192 represent cases where neither

Test	Num. tested	Mutual timeouts	Avg. tested subset SSes
broadcast_test	79	7	112.3
thr_exit_join	79	12	69.7
mutex_test	79	0	-
paradise_lost	79	50	77.4
paraguay	79	45	59.6
rwlock_downgrade	79	44	86.3
priority-sema	59	2	13.0
alarm-simultaneous	44	17	7.8
wait-simple	52	15	33.8
Total	629	192	65.8

Table 4.3: Summary of partial verification results on timed-out tests. "Mutual timeouts" counts how often both QS-Limited-HB and SSS-MC-ICB (the best bug-finding approach from each group) timed out. Among those, "Average tested subset SSes" counts how many partial verifications QS-Limited-HB provided on average for each test.

Quicksand nor ICB were able to provide a conclusive result either way.[16] For these 192, I show the number of state spaces Quicksand was able to complete in the "Average tested subset SSes" column. These completions guarantee that, if the test program could expose a bug, it would depend on a data race not discovered yet, or be reachable only under a superset combination of preemption points not yet reached.

4.5.4 Data race analysis

Beyond finding new bugs and completing full verifications with data-race preemption points, I evaluated Quicksand's performance for classifying data race candidates in two ways: its ability to check nondeterministic data races not reachable under a single-pass analysis (§4.1.1) and its ability to suppress reallocation false positives (§4.2.6). Table 4.4 presents the results for this section.

Nondeterministic data races

Some memory accesses may be hidden in a control flow path that requires a nondeterministic preemption to be executed (§4.1.1). In such cases, a single-pass dynamic data race detector might not achieve the coverage necessary to identify a racing access pair as a candidate at all, let alone check the resulting behaviour with such as Landslide. I instrumented Landslide to report these to Quicksand and counted how many such led to Quicksand finding new bugs when used as preemption points. Such bugs could be considered *false negatives* of the single-pass approach. The left half of Table 4.4 breaks down the

[16]Thesis note: SSS-MC-Shared-Mem was added subsequently to this analysis's publication in [BG16], at which time SSS-MC-ICB was the best-performing approach among control experiments. Nevertheless, mutual timeouts among all six testing approaches constituted roughly one third of the test suite.

Test	Num. tested	Data-race bugs: Limited HB All	Data-race bugs: Limited HB N.D.	Data-race bugs: Pure HB All	Data-race bugs: Pure HB N.D.	Verifications: Pure HB DR PPs	Verifications: Pure HB Benign	Verifications: Pure HB Untested	Realloc. FPs
broadcast_test	79	2	1	2	1	655	97	150	52
thr_exit_join	79	11	4	7	3	566	68	249	338
mutex_test	79	9	1	8	1	911	127	44	7
paradise_lost	79	7	3	6	2	783	2	414	166
paraguay	79	6	1	3	2	936	9	510	180
rwlock_downgrade	79	4	1	3	0	543	1	310	156
priority-sema	59	6	4	6	6	65	51	3	0
alarm-simultaneous	44	17	1	7	6	35	0	29	35
wait-simple	52	7	2	2	0	71	1	28	31
Total	629	69	15	44	21	4565	356	1737	965

Table 4.4: Data race statistics among Quicksand experiments. "Data-race bugs" counts, among Quicksand's bugs, how many required data-race preemption points to expose; among those, the "N.D." ("nondeterministic") columns show how many candidates required model checking to identify in the first place (§4.5.4). "Total DR PPs" counts how many unique data-racing instructions QS-Pure-HB identified among tests where it found no bugs. Among those, "Benign" counts how many were refuted as non-failing, while "Untested" counts how many could not be checked in the time limit. Finally, "Realloc. FPs" counts how many reallocation false positives QS-Limited-HB suppressed.

types of bugs found in each test case, showing both the total number of data-race bugs and the number among those that required such nondeterministic data races to expose. To ensure a fair comparison, I disabled Quicksand's reallocation false positive suppression (§4.2.6, itself evaluated in the next section) for this experiment. This prevents Landslide from suppressing an observed reallocation data race candidate on the first interleaving, which would falsely classify it as nondeterministic, even though a single-pass would not (indeed, should not) suppress such candidates.

Figure 4.10 visualizes the difference between single-pass and model-checking-enabled data race analysis. The first and third series represent the bugs found using preemption points from single-pass data race candidates only, i.e., the state-of-the-art approach used by RaceFuzzer [Sen08] and Portend [KZC12]. The second and fourth series show all data-race bugs Quicksand found, which includes the former type as well as new bugs involving nondeterministic races. QS-Limited-HB found a nice 69 data-race bugs in total, 15 of which required nondeterministic data-race preemption points to expose. QS-Pure-HB is even more dependent thereupon, requiring them in 21 cases among its 44 total data-race bugs. Moreover, although the frequency of these nondeterministic races varies across the different test cases (for example, almost all in `broadcast_test` were nondeterministic; almost none in `mutex_test`), they are still at least present in all tests, meaning it is not just an issue of writing "better" test cases to avoid them.

Note that I do not compare how much testing time is required before identifying the data races involved in each bug. While single-pass data races are all found after a single test execution, Quicksand may potentially take up to all 10 CPU-hours before identifying

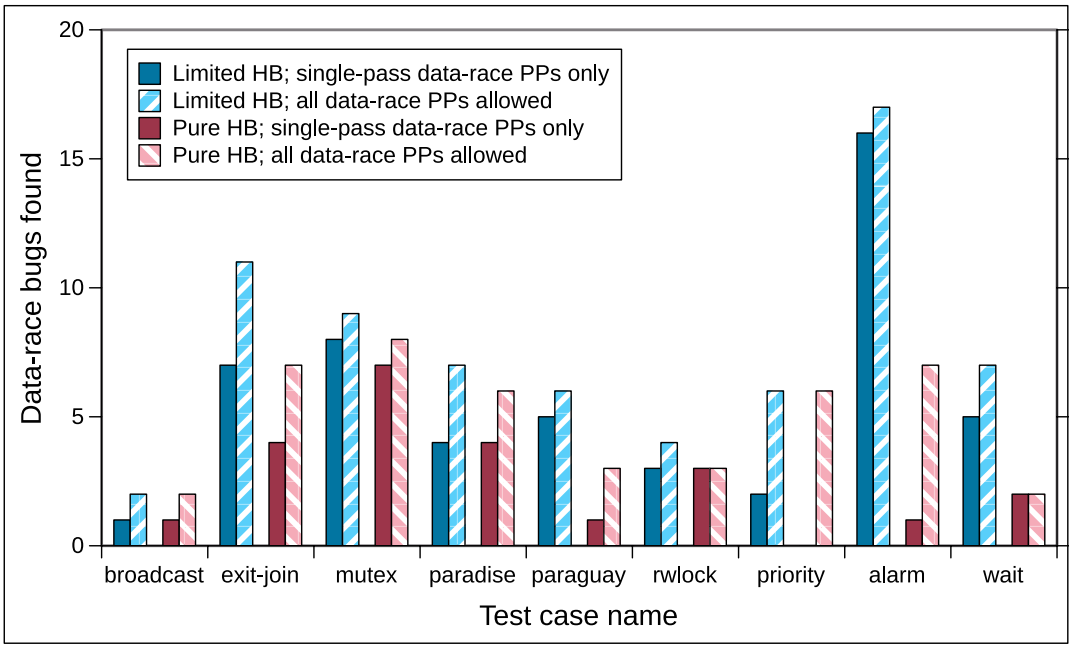


Figure 4.10: Some bugs require nondeterministic data-race preemption points to expose. Incorporating these data races, which are not observed under single-pass analysis, as new preemption points allowed Quicksand to find 128% (Limited Happens-Before) to 191% (Pure Happens-Before) as many data-race bugs compared to using single-pass candidates alone.

a nondeterministic data race. However, prior work data race tools [SI09, FF09], being not integrated with a model checker, are not intended to discover new candidates under subsequent runs. Running a single-pass data race tool repeatedly for 10 CPU-hours could potentially uncover some nondeterministic candidates, but stress testing's comparative problem with achieving reliable coverage is already well-understood [MQ07, CBM10], so I hope the reader will consider this experiment enough evidence for model checking already. Likewise, replay-based tools such as RaceFuzzer [Sen08] and Portend [KZC12] depend upon the data race detector to provide an execution trace leading to each candidate. This result suggests that such tools could benefit from a similar feedback loop as is used in Iterative Deepening, for example, to discover new transitively-reachable data races while testing initial ones, even if full verification not necessarily be their goal.

Reallocation false positive suppression

In §4.3.2 I showed the soundness of suppressing data race reports between two heap accesses when the surrounding memory was re-allocated in between. Table 4.4's "Realloc FPs" column shows the total number of such data race candidates for each test program, totaling 965 across all tests. Among these, only 64 were observed to avoid the reallocation in an alternate interleaving, thereupon being promoted to real data-race preemption points. §4.3.2's proof guarantees the safety of pruning all state spaces resulting from

the 901 others. Among the 64 true data races, none exposed a new bug when used as a preemption point. This suggests that for other data race tools, suppressing reallocation candidates may be a productive heuristic, even if unsound without Iterative Deepening. However, Quicksand was able to correctly identify the 64 violations of that heuristic, among 26 distinct tests, and fall back to classifying them with DPOR.

4.6 Discussion

This section will list the current limitations of Quicksand and Iterative Deepening and discuss opportunities for future improvement.

4.6.1 Experimental bias

The evaluation design (§4.5.1) contains two major shortcomings that, to my surprise, no conference reviewer, audience member, or colleague ever called me out on. Firstly, the single-state-space preempt-everywhere strategy was conducted only with ICB enabled (SSS-MC-ShMem). This resulted in reasonable bug-finding performance, ultimately reaching 80% as many bugs found as Quicksand's best (QS-Limited-HB) in Figure 4.7. However, ICB tends to repeat work across multiple preemption bound iterations, as evidenced between SSS-MC-ICB and SSS-MC-DPOR in Figure 4.9. Correspondingly, a version of SSS-MC-ShMem configured to use traditional DPOR without ICB would effectively begin with a preemption bound of infinity and perhaps be more competitive with Quicksand on verifications. Of course, this would trade off against its bug-finding performance, but an expert user could compensate by deciding between the two modes depending whether she thinks the test is more likely to be buggy or correct. Granting such user involvement, Quicksand's maximal state space mode (§4.4.4) would correspondingly verify more tests than QS-Pure-HB, especially if it were given 10 wall-clock hours on 1 CPU rather than 1 on 10. Future work could also improve both Quicksand's and ICB's ability to identify and skip redundant work across their respective iterations, as discussed below.

Secondly, and more fundamentally, representing state-of-the-art approaches by reimplementing them in one's own tool is fraught. The comparison between Iterative Deepening and ICB in §4.5.2 and §4.5.3 could possibly have been conflated by Landslide's implementation of ICB and/or DPOR being slower on account of the simulated execution environment (§3.3). A more rigorously scientific comparison would extend a prior work model checker, such as CHESS [MQB+08], to support dynamically-configured data-race preemption points, and evaluate Quicksand with it versus its own ICB implementation as well, to isolate any such conflating factors. Concurrently with these results' publication in OOPSLA [BG16], more advanced state space reduction algorithms were proposed, such as MCR [Hua15] and RCMC [KLSV17] (§7.1). It is not immediately obvious that these techniques' benefits would be orthogonal with Iterative Deepening; in other words, the benefit of Quicksand with a MCR- or RCMC-enabled model checker might be reduced compared to the benefit shown in §4.5.2 and §4.5.3. Future model checkers' evaluations should strive to fairly represent these and other latest algorithms in their comparisons.

4.6.2 Avoiding redundant work

When Quicksand extends a small state space with more preemption points, the new state space is guaranteed to test a superset of interleavings compared to the old one. Because Quicksand prioritizes completing small state spaces before their descendants, the superset state spaces we run later will repeat each branch of their already-completed subsets, and any interleaving which does not preempt threads on any of the new preemption points will be repeated work. This may make Quicksand slower than the single-state-space approach to find certain bugs, for example, if both `mutex_lock()` and `mutex_unlock()` preemption points together expose a bug, but not either alone. Predicting whether an upcoming interleaving has already been tested is not straightforward, but future implementations of Iterative Deepening and/or ICB could incorporate cross-job memoization to prune some or all such repeated work.

Similarly, when pursuing total verification, if the state space resulting from preempting on every instruction (or equivalently, the maximal state space, thanks to §4.3) could be completed in time, a model checker which immediately jumped to that state space, abandoning all smaller subsets would certainly achieve verification faster. Quicksand's maximal state space mode (`-M`, see §4.4.4) can strike a middle ground between Iterative Deepening and single-state-space preempt-everywhere,[17] but future implementations of Iterative Deepening could prioritize the maximal state space more flexibly still. For example, pinning its job to one of the available processors regardless of the status of any smaller jobs would avoid getting too flooded with smaller jobs to even begin the maximal job before time runs out. When full verification is infeasible, completing even an intermediate-sized job would allow immediately pruning all subset jobs thereof, perhaps using a form of binary search (on the preemption point set size) to find an appropriately-sized intermediate job.

4.6.3 Preemption point subsets

Quicksand was able to partially guarantee safety for some preemption points in 93% of tests with too-large maximal state spaces (§4.5.3). However, in 6 cases, no more than the minimal state space could be verified, and in 18 others, no state spaces were completed at all. Larger state spaces often result from finer-grained locking, which can indicate a more intricate concurrent algorithm or an unnecessarily complicated design (or both). Such programs may require even more rigorous verification than a program with a single global lock, making them important to consider for future work. While Quicksand uses `within_function` (§4.4.2) *statically* to restrict where preemption points could arise in advance of the test, future Iterative Deepening implementations could use this mechanism to *dynamically* subset preemption points further, making partial verification of larger tests possible, potentially even involving the user with interface options to enable and disable preemption points of her choosing at run-time. §8.3 discusses this possibility further.

[17]Thesis note: `-M` mode was implemented after the conference paper's publication [BG16], and will be evaluated alongside transactional memory later in §6.3.

Static data race analysis

In §4.5.2, I evaluated the state-of-the-art approach's ability to find data-race-induced failures by configuring a static predicate to preempt on any non-stack memory access (`-0`, see §4.4.4). This introduced hundreds of new preemption points on each new test execution, with a prohibitive performance impact. While this performance could be improved by relaxing the preemption strategy, instead using a static or single-pass analysis to find data race candidates in advance [KZC12], that would sacrifice soundness of the verification guarantee, as I showed in §4.5.4. However, Quicksand itself could employ static data race analysis such as RacerX [EA03], or single-pass dynamic analysis such as ThreadSanitizer [SI09] in future work. Any data race candidates identified in advance could heuristically be included in Quicksand's initial seed preemption point sets (§4.2.2), enabling it to focus on the most suspicious races immediately, rather than waiting for them to be identified after potentially many iterations of model checking.

4.6.4 Partial verification

When full verification is not computationally feasible, some jobs with data-race preemption points will inevitably time out, and Quicksand cannot guarantee those races are false positives or benign, even though no bug was found. In the "Untested DR PPs" column of Table 4.4, I show how many such candidates Quicksand (with Pure Happens-Before) could not verify in each test, ultimately totaling 38% of all data races in tests which timed out. In prior work, Portend introduced the *k-witness harmless* metric [KZC12] for heuristically classifying the likelihood that each data race lead to a failure or be benign. Quicksand could incorporate this metric to guide the user's attention to the unverified data races most likely to be worth her time. In §4.5.3 I presented partial verification results measured in tens or hundreds of subset state spaces completed on average per test case. However, attempting to maximize the raw number of completed state spaces is not necessarily the most user-friendly way to present partial verifications. For starters, those numbers included small state spaces which were subsets of other state spaces also completed; the user need not examine both subset and superset separately to understand what was tested. Future work should at least perform basic set comparisons to present only the non-redundant state spaces completed when time runs out. For a further research challenge, user studies could help to determine the most effective interface for presenting these partial results from a software development perspective, which I discuss further in §8.1.

Quicksand is not the first concurrency tester to provide a partial verification guarantee when it times out on too-large tests. Probabilistic Concurrency Testing (PCT) [BKMN10] proposes to use random exploration of the state space and quantify the probability that a bug may remain in some untested interleaving after a time-out, eschewing DPOR's depth-first search model to instead sample broad cross-sections of large state spaces. However, it proposes no alternate reduction strategy, making full verification impractical, and furthermore is opaque to the user about which parts of her code were actually tested. Meanwhile, ICB proposes to inform the user of the maximum number of preemptions used to

test any individual interleaving, under the assurance that most bugs are likely to be found with fewer preemptions (Table 4.2). Iterative Deepening allows the expert user to restrict a test's scope via the `within_function` command to only the modules of a codebase she wishes to test. These guarantees could each be useful to developers in different scenarios, and future work could combine the three approaches to provide all benefits at once, for example, using ETAs to heuristically decide when to switch between DPOR, ICB, and/or PCT in large state spaces, as discussed further in §8.3.

4.7 Summary

In order to supplant conventional stress testing, which, despite its inability to reliably expose, reproduce, or verify absent bugs in any finite amount of testing time, remains a popular choice for concurrency programmers of all skill levels, stateless model checking must meet users' needs regarding realistic testing budgets. This chapter has presented Quicksand, which automatically navigates the trade-off between fast bug-finding and formal verification depending on the size of the test. My contributions have been as follows.

- Iterative Deepening (§4.2), an algorithm for model checkers to simultaneously test multiple state spaces, incorporating new preemption points identified with dynamic analysis on the fly.
- A proof of convergence (§4.3.1), showing that for any verification obtained under even the most extreme preemption strategy, Iterative Deepening with data race analysis provides an equivalently strong one, with far fewer preemption points necessary.
- A technique for suppressing certain false positive data race candidates under Limited Happens-Before (§3.4.4) by identifying intervening `malloc()` and `free()` calls (§4.2.6), and a corresponding soundness proof when this technique is used under Iterative Deepening (§4.3.2).
- Quicksand (§4.4), an Iterative Deepening implementation which incorporates several heuristics for prioritizing which state spaces are most likely to uncover bugs or, should no bugs exist therein, which ones are most likely to complete within a user-specified fixed CPU budget, as informed by state space estimation (§3.4.3).
- A 629-test evaluation of Quicksand against several prior state-of-the-art model checking approaches implemented in Landslide (§4.5), which showed that Quicksand provides both faster bug-finding (§4.5.2) and more full verifications (§4.5.3), delivering "the best of both worlds" as promised, and also demonstrated the need for a bidirectional feedback loop between model checking and data race analysis (§4.5.4).

The next chapter will tell of my experience and results deploying Landslide in an educational setting, equipped with Quicksand to allow even inexperienced student users to benefit from stateless model checking with little to no manual configuration burden.

Chapter 5

Education

Knowing the students might one day fix their concurrency bugs... it fills you with determination.

—Undertale (paraphrased)

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 I showed in my M.S. thesis [Blum12a] 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 The Pennsylvania 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 they 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 A. 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

From the Spring 2015 semester through Spring 2018, 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 [Eck18d], 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 `https://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 was given in Spring 2018, and is available at `https://www.cs.cmu.edu/~410/landslide/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. 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 (of course, this is an anecdotal opinion, not a scientific conclusion).

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 online at `https://www.cs.cmu.edu/~410/landslide/landslide-guide-p2.pdf` (CMU version) and at `https://www.cs.cmu.edu/~410/landslide/landslide-guide-psu.pdf` (PSU version) and at `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 check, 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`) have been updated, 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`, designed to test student semaphore code, 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 [Eck18c].
- The `landslide` wrapper script logs the time and command-line options of invocation

[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.

and captures a snapshot of the student code and results of the test and saves them to AFS (CMU's network file system) [HKM+88] after each run.

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 second 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 J. Sullivan.
- `rwlock_downgrade_read_test`: Copied directly from the P2 test suite; tests R/W locks for mutually-exclusive and deadlock-free `rwlock_downgrade()`, which should atomically convert a lock held in exclusive write mode to shared read mode. Written by me (during my time 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/landslide-friendly-tests.

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 replaced the default definition with a different name and/or data structure. If so, emits macros to configure `is_runqueue()` to handle certain common alternate approaches (part of the patch described below). Either way, defines a function `get_rq_addr()` to return the address of the 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, also 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 in the Landslide repository at `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.

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

The data set comprises 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 (inspecting their

[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).

code if necessary, as multiple traces may refer to the same bug), how many were deterministic (i.e., reported 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 filter out false positives 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 subsequently verify any or all thereof. Among the concurrency bugs, a wide variety of types

	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.

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 because 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 concurrency bugs it found to get fixed before project handin. Although lacking a control

[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.

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

1. `priority-donate-multiple` found two deterministic bugs and one 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 one deterministic bug and three 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 three separate locations. Lastly, the heap checking alone found a deterministic use-after-free in a third group's `process_execute()`.

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 having found 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 [EA03], 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 might have 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.

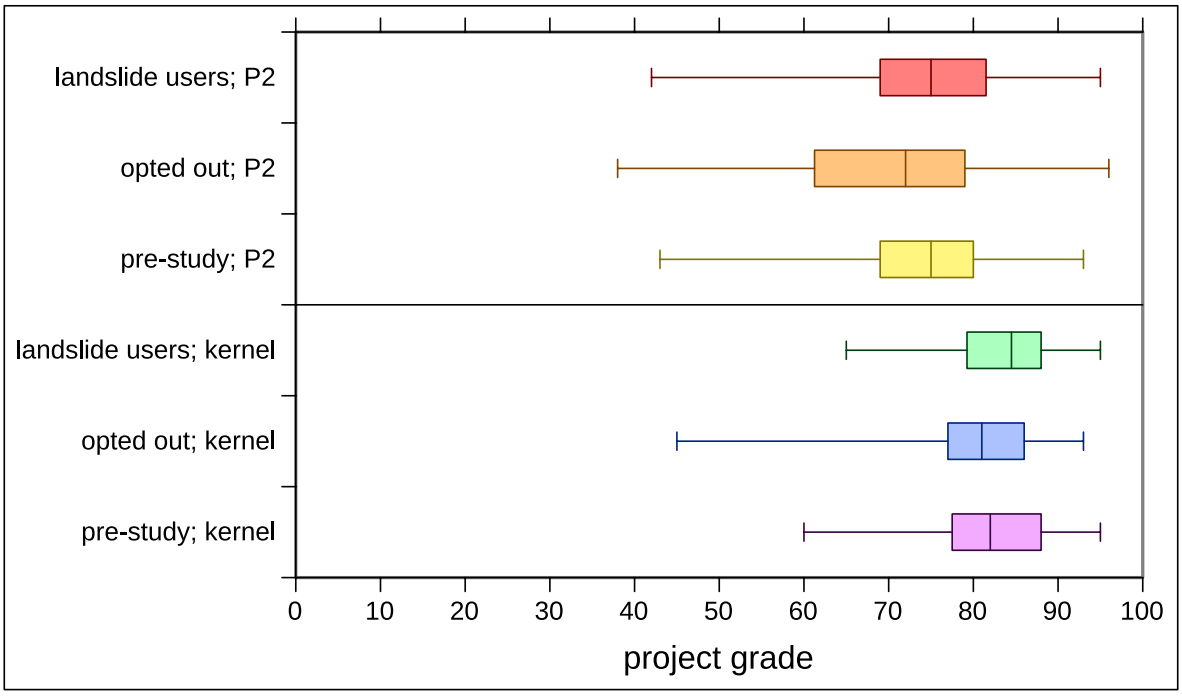
Impact on grades

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 ("declined") 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.

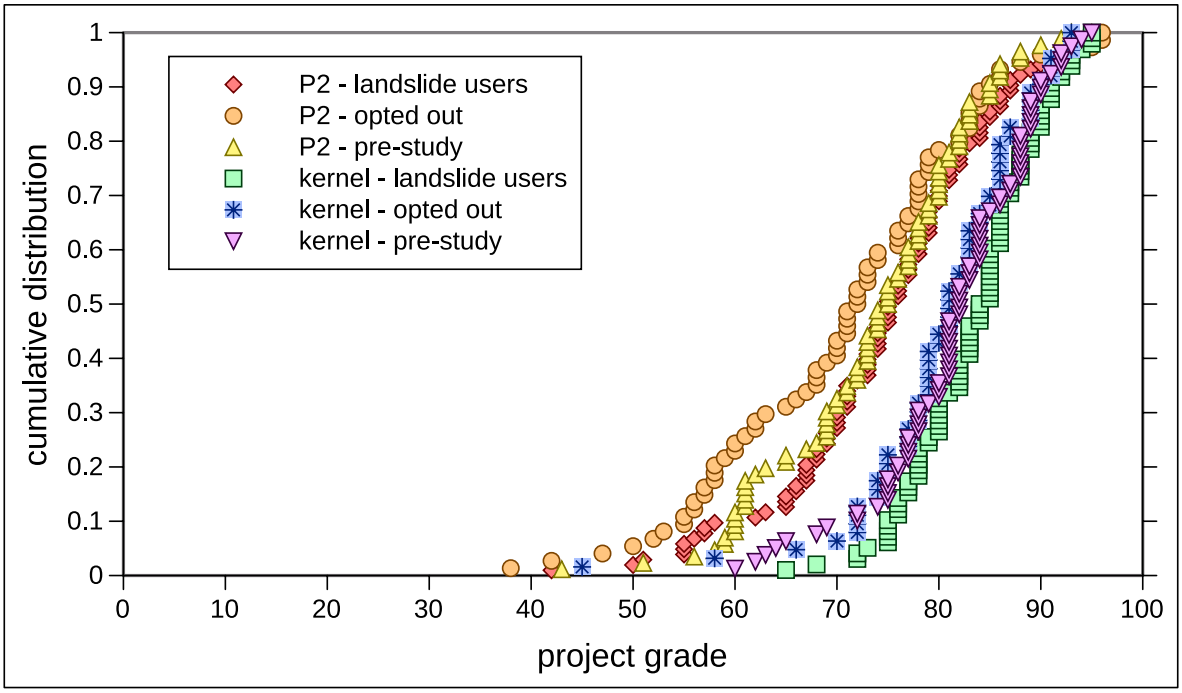
Results. Overall, Landslide users did indeed receive slightly better grades overall on their P2 submissions than non-users from the same semesters, although comparing to the second control group lessens that difference somewhat. The difference is also less pronounced in both comparisons among P3 grades (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.

Statistical significance. Figure 5.2 presents more detailed statistics from these six distributions. To assess whether the differences are significant, I use the *k*-sample discrete

[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.

Anderson-Darling test [AD52, SS87] provided by the R statistical programming environment [R18, SZ18], 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, which do not address that alternative hypothesis, were significant, while the cross-semester comparisons, which attempted to do so, failed to confidently reject even the null hypothesis. However, the difference being smaller in P3 than in P2 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. Table 5.3 presents the results.

[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 [FB13].

[7] I choose not to correct for multiplicity among these four comparsons as 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.

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 (i.e., one contender calls `critical_section()` twice) 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 test probes 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 cor-

rect 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 [R18] using Fisher's exact test [Fis22], 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 [Mun11] (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 §4.5.4) 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 another, 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 favourably to stress testing, and found the experience worthwhile and worth

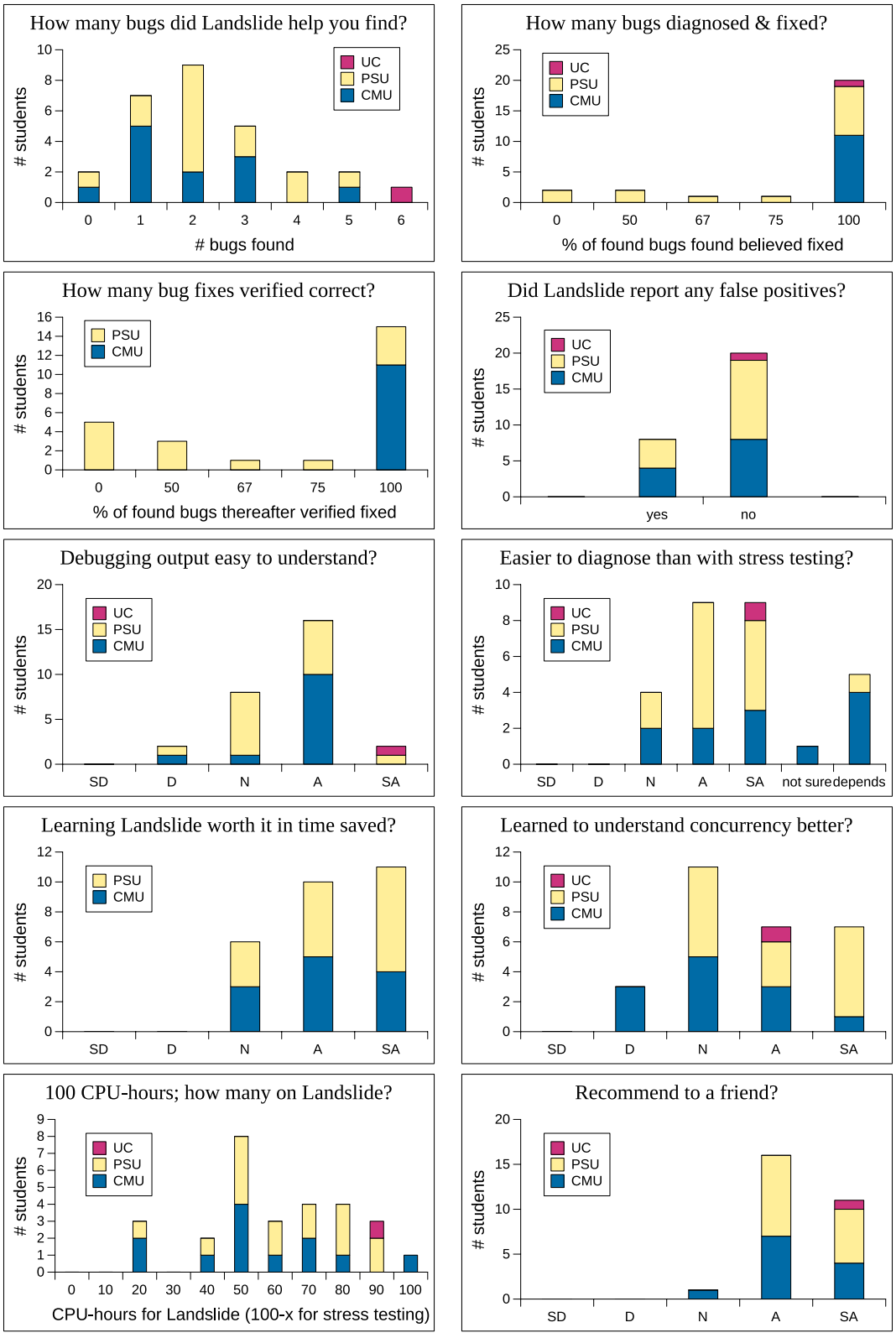


Figure 5.3: Student survey responses. SD/D/N/A/SA stands for strongly disagree/ disagree/neutral/agree/strongly agree.

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 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; see §8.2). 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` (a 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

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

filenames and line numbers in stack traces, and (quite naturally) suggested that accurate stack traces would make debugging easier. Checking against their usage 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. (As 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 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. Three students declined to answer this

question; I reproduce the rest here, paraphrasing for clarity and brevity, denoting my own clarifications and commentary with [square brackets].[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. [This respondent 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 which may or may not find bugs.
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

[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.

overnight.

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 [reported by three students].
8. [Seven non-respondents.]

Though I did not ask this question on the CMU survey, my experience handling student questions in person suggests CMU students (as well as Professor Eckhardt) 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 (CMU students comparatively preferred the more nuanced answer that it depends on the type of bug), and reported more often that it 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

This section will discuss the study's limitations and offer 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 2012 [Blum12a], 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

Aside from 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" question, 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 and 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.
5. *In addition to finding bugs, did you manage to fully verify your code under any tests by letting Landslide complete all state spaces before reaching the specified time out?* This would measure how thoroughly students understood the underlying research technique, and serve as a follow-up to the "how many bug fixes verified" question (where students often stopped Landslide midway through after a little while).

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 not 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 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 giving a lecture remotely via videoconference, 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 §4.5'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 [SBG10] or CHESS [MQB+08], could easily be used on other thread-library-like programming projects at any university.

5.5 Summary

This chapter has introduced Landslide as an educational tool in the undergraduate operating systems classes at CMU, U. Chicago, and PSU. I have made the following contributions:

- A lecture, polished over seven semesters, for introducing stateless model checking as a concurrency testing approach in high-level undergraduate classes (with many slides admittedly specific to Landslide) (§5.1.1).
- A system for automatically instrumenting student thread library (CMU, PSU) and kernel (U. Chicago) project implementations to be compatible with stateless model checking (§5.1.2, §5.2.2).
- An experimental procedure for evaluating debugging and verification tools as student-accessible teaching aids, including a survey for measuring the more intangible human experience (§5.1.4, §5.2.4).
- A large evaluation spanning nine semesters of operating systems classes (§5.3.1), two of which were held at other universities besides CMU, in which 145 students used Landslide, a majority of which found bugs (§5.3.1), and a vast majority of which expressed via the survey that they had a good experience (§5.3.3).
- A thorough statistical analysis of Landslide's effect on student submission quality (§5.3.2), evaluating both impact on ultimate project grades and student tendency to submit any of four well-known case study bugs. Ultimately, while positive trends were observed, this study concluded that a statistically-significant conclusion could not be drawn while simultaneously compensating for several obvious sources of possible bias. Nevertheless, this level of statistical rigor is not yet widespread in computer science research, so I would hope this study may set a precedent that help mature the field.
- Concrete action taken to establish Landslide as a permanent fixture of 15-410's grading process at CMU, student-oriented user guides should the course staff wish it to remain available as a debugging tool, as well as general recommendations for any Pintos-based OS class at other universities to use it for grading as well (§5.4.3).

Human subjects research is inherently messy. Each individual 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, and relatively very little negative feedback such as false positives. 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, and students provided intelligent feedback suggesting they truly understood the debugging process.

Chapter 6

Transactions

One can only build a sand castle where the sand is wet.
But where the sand is wet, the tide comes.
Yet we still build sand castles.

—Yuri, Doki Doki Literature Club

Transactional memory (TM) [HM93] is a concurrency primitive by which programmers may attempt an arbitrary sequence of shared memory accesses, which will either be entirely committed (i.e., 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, entirely discarded with an error code returned to allow the programmer to invoke a synchronized backup path. Transactional memory specifications typically have three API functions, abstractly speaking:

- begin 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. When the transaction is started, begin returns a success code. 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.
- end 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 point with an error code as described above.
- abort 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 as 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 [ATLM+06, SATH+06, DGK09]. 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 [Intel18], 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 [Hac14, Intel17], attesting to the feature's complexity and the need for formal verification on both sides of its API.

Terminology

The world of transactional memory is home to several more confusing acronyms similar to "HTM". Transactional Synchronization Extensions (TSX) refers to Intel's implementation of HTM on Haswell and more recent microarchitectures [HKO+14]. 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()` [GNU16]. 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 [RG01, Intel13]. 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.

Transactional atomicity comes in two flavours, *strong* and *weak atomicity* [MBL06]. HTM implementations use the processor's cache coherence protocol to detect conflicts, and hence any conflicting memory access from another CPU will cause an abort, regardless of whether that access was itself transactional. Hence, HTM transactions are *strongly atomic*, i.e., appear indivisible to all other code. STM implementations rely on the programmer to use its interface to access any shared memory she intends to protect with its transactions, which cannot detect conflicts with non-transactional accesses that bypass it. Hence, STM transactions are *weakly atomic*, i.e., its transactions will abort only from conflicts with other transactions, while other code may interleave with them freely. The manner in which non-transactional code may interleave with STM transactions further depends on whether the STM uses *eager versioning*, wherein transactional updates are immediately visible to non-transactional code, or *lazy versioning*, wherein such updates are visible only after commit [SMAT+07, §2]. Unless explicitly noted as weak atomicity, this chapter assumes strong atomicity, and "STM semantics" refers to a strongly-atomic

HTM program emulating STM with retry loops (see §6.2.2). Treatment of weak atomicity in §6.1.4 and §6.2.4 assumes eager versioning.

Motivation

For the most part, existing stateless model checkers model interrupt-driven thread switching as the only source of nondeterminism in a program's execution. In other words, assuming a fixed test input,[1] the only way multiple behaviours can arise is by scheduling different threads at each preemption point. Only recently have checkers emerged that extend this model by incorporating other nondeterminism sources: store-buffer reordering arising from relaxed memory models [ND13, ZKW15, KLSV17] and event-driven application programming models [JMR+15, BRV15], both of which are especially relevant in mobile programming. Under TM, `begin`'s return value is also nondeterministic, and moreover may depend on execution events from the future in a parallel universe (i.e., subsequent memory conflicts detected and reverted), and no model checker yet exists which can model this. In this chapter I will develop a concurrency model to incorporate abort nondeterminism (§6.1), including two equivalence proofs to help cope with the consequent state space explosion (§6.1.2 and §6.1.3), extend the proofs to account for weak atomicity (§6.1.4), extend Landslide to support testing TM programs under a variety of execution semantics (§6.2), and demonstrate its practicality both by finding new bugs (§6.3.2) and by verifying correctness (§6.3.3) in a variety of microbenchmarks and real-world HTM programs.

The following sections will make heavy reference to the example TSX program from Figure 2.3 (§2.1.4) while defining the new concurrency model, so I reproduce it here in Figure 6.1 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 nondetermin-

[1] Symbolic execution [King76], a neighbouring research area, addresses input nondeterminism; see §7.5.

ism, 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 §6.4.4.

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, as shown below in Figure 6.2.

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

Figure 6.2: Figure 6.1 with non-atomic operations shown as pseudo-assembly.

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, a hypothetical interleaving such as shown in Figure 6.3, or equivalently written sequentially as:

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

or, henceforth further simplified for clarity:

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

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

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

Thread 1	Thread 2
`status = _xbegin()`	
`temp <- x;`	
`temp <- temp + 1;`	
	`status = _xbegin()`
	`x++;`
	`_xend();`
`x <- temp;`	
`_xend();`	

Figure 6.3: Impossible hypothetical interleaving within transactions of Figure 6.2. The effects of **T1**@2*a*−2*b* are never globally visible, as its transaction must abort, and moreover **T1**@2*c* − 3 will not execute at all.

However, the `x++` operation from the failure path (correspondingly 6*a*, 6*b*, 6*c*) *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:[2]

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

Prior work [DAS16] proposed the idiom shown in Figure 6.4 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 **T1**@1−2*c*, **T2**@1−3, **T1**@4 − 7[3] glossed over how such a sequence of operations would be carried out under HTM. For example, **T1**'s write during 2*c* is not actually visible to **T2**, although it would be under a thread-scheduling-only concurrency model.

Intel's official TSX documentation [Intel18] summarizes its interface and behaviour in prose. Recent work has used proof assistants to formalize some of the execution semantics of x86 in general [SSN+09] and of transactions in particular (at both hardware- and language-level) [CSW18]. However, state-of-the-art advances in model checking algorithms still state their theorems and proofs in prose [CMM13, AAJS14, ZKW15, DL15, Hua15, BG16, KLSV17], 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:

[2]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.

[3]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 6.4: More correct version 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.

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 $\mathbf{T1}@1, \mathbf{T2}@1-3, \mathbf{T1}@4-7$ is an equivalent verification to MCing the one above; in fact, this interleaving suffices to check all observable behaviours of all interleavings of all subsets of $\mathbf{T2}@1-3$ with all subsets of $\mathbf{T1}@2a-2c$, whether they share a memory conflict or not. Stated formally, let:

- $\mathbf{Ti}@\alpha$ be an HTM begin operation,
- $\mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_n$ be the transaction body (with β_n the HTM end call),
- $\mathbf{Ti}@\phi_1 \ldots \mathbf{Ti}@\phi_m$ be the failure path, and
- $\mathbf{Ti}@\omega_1 \ldots \mathbf{Ti}@\omega_l$ 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**, WLOG pre-

sented here as two threads, and for any number of thread switches away from Ti during the transaction, WLOG presented here as one switch):

Lemma 3 (Equivalence of Aborts)**.** *For any interleaving prefix*

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

with $b < n$, $j \neq i$, $k \neq i$, et cetera, either:

1. $\mathbf{Ti}@\alpha, \mathbf{Tj}@\gamma_1 \ldots \mathbf{Tj}@\gamma_j, \mathbf{Tk}@\kappa_1 \ldots \mathbf{Tk}@\kappa_k, \mathbf{Ti}@\phi_1 \ldots$ *(conflicting case), or*
2. $\mathbf{Ti}@\alpha, \mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_b \ldots \mathbf{Ti}@\beta_n, \mathbf{Tj}@\gamma_1 \ldots \mathbf{Tj}@\gamma_j, \mathbf{Tk}@\kappa_1 \ldots \mathbf{Tk}@\kappa_k$ *(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 $\mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_n$. If so, then the hardware will abort Ti's transaction, discarding the effects of $\mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_n$ and jumping to $\mathbf{Ti}@\phi_1$, satisfying case 1. Otherwise, by DPOR's definition of transition dependence ([FG05]; see also §3.4.2), $\mathbf{Ti}@\beta_{b+1} \ldots \mathbf{Ti}@\beta_n$ 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 3 (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 $\mathbf{Ti}@\alpha, \mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_b, \mathbf{Tj}@\ldots, \mathbf{Tk}@\ldots, \mathbf{Ti}@\beta_{b+1} \in I$, apply Lemma 3 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 3 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 3. 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 m. Lemma 3 guarantees mutual exclusion of m.

Theorem 4 (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 of the two variant definitons of data races that Landslide supports (§3.4.4). 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 [SI09, OC03].
  - $I \Rightarrow I'$: 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 $\mathbf{Tj}@\nu$ [PS03, FF09].
  - $I \Rightarrow I'$: If $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ race in I, then they cannot both be in successful transactions, or else Lemma 3 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 $\mathbf{Ti}@\mu, \mathbf{Tj}@\nu$ race in I', then they cannot both hold m, or else $\mathsf{lockify}_m$ would provide the corresponding unlock and lock for $\mathbf{Ti}@\epsilon$ and $\mathbf{Tj}@\chi$ 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. □

6.1.4 Weak atomicity

Prior work has highlighted the difference between *strong* and *weak atomicity* of transactions [MBL06, SMAT+07]. Thus far we have assumed transactions will abort from any

intervening memory conflict, whether or not that access is itself transactional. This corresponds to strong atomicity, the execution semantics provided by TSX. However, weakly-atomic STM systems do not abort transactions when a non-transactional access conflicts, and condition #1 of the execution semantics from §6.1.2 does not hold. To model weak atomicity, Lemma 3 must be extended with a third possibility:

Lemma 4 (Weak Equivalence of Aborts)**.** *For any interleaving prefix such as defined in Lemma 3, either one of the original two conditions hold, or:*

3. *All events in* $\mathbf{Tj}@\gamma_1 \ldots \mathbf{Tj}@\gamma_j, \mathbf{Tk}@\kappa_1 \ldots \mathbf{Tk}@\kappa_k$ *are non-transactional, and the original interleaving itself is a legal execution.*

Proof. As before, except first case on whether any intervening operations by **Tj**/**Tk** are transactional.[4] If so, either the original case #1 or #2 applies. If not, **Ti** may ultimately proceed with $\mathbf{Ti}@\beta_{b+1}$ (its preempted next transactional instruction) without aborting, and the new case #3 applies instead. □

Theorem 3 must be weakened correspondingly, yielding Theorem 5.

Theorem 5 (Weak 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 with respect to other transactions only***, or aborted immediately.*

Proof. As before. □

This proof assumes that all operations $\mathbf{Ti}@\beta$ are in fact executed transactionally. In STM systems, the user must annotate all shared memory accesses she wishes to be protected, so an incorrectly-annotated program could mix transactional and non-transactional operations in **Ti**. To emulate the interactions thereof with other threads would require fully simulating failure rollback after all, which I must leave to future work. I believe this assumption to be justified for the sake of any future weakly-atomic HTM systems, in which the user would not have the mixed blessing of such fine-grained annotations.

Conveniently, Theorem 5 still guarantees transaction-transaction atomicity, so mutual exclusion of m is preserved in §6.1.3's formulation of $I' = \exists m.\mathsf{lockify}_m(I)$, and Theorem 4 applies unmodified. §6.2.4 discusses the implementation consequences of this weakening.

6.2 Implementation

Support for TSX programs in Landslide is implemented in five 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.

[4] If **Tj** or **Tk** already have a transaction active, their `begin` must have occurred before $\mathbf{Ti}@\alpha$, and the theorem can apply starting from the earlier `begin` with the thread IDs swapped.

6.2.1 User interface

Landslide provides its own "implementation" of the TSX interface, which matches GCC's interface exactly, located in `landslide-friendly-tests/inc/htm.h` under the subdirectory `pebsim/p2-basecode/`. The interface is implemented in `landslide-friendly-tests/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.[5] 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 `landslide-friendly-tests/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) (§6.1.2, §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 (§6.2.5). Requires `-X` and *not* `-A`.
- `-W` (for "Weak atomicity") enables weak atomicity, allowing non-transactional code to interleave within transactions, corresponding to §6.1.4. Requires `-S`.

6.2.2 Failure injection

Each preemption point (`struct nobe` 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.

[5] 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.

Adding possible abort codes

Up to four abort codes are considered depending on §6.2.1's testing options. 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 [Blum18a].
- `_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 `landslide-friendly-tests/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 [GNU16], 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. So called because doing so would trigger a mode switch, which the specification allows to abort for any reason.[6] See below for discussion of other cache capacity concerns.

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

[6] As far as I know, the reason for syscall aborts is both hardware- and kernel-dependent, and in fact nondeterministic; on my Haswell Core i7 running Linux 4.19, I observed a majority of `_XABORT_CONFLICT`s, a scant few `_XABORT_CAPACITY`s, and surprisingly many with no reason specified whatever; however, not once did a transaction succeed with even a single `gettid()` call [Blum18c]. For simplicity of both implementation and client programs, Landslide always distinguishes system call aborts with `_XABORT_CAPACITY`.

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 [GNU16], depends on currently-unsupported `xbegin` nesting, which I discuss further in §6.4.3.

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.[7] 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.

6.2.4 Weak atomicity

If the user supplies the `-W` option (§6.2.1), Landslide will emulate STM's weak atomicity semantics (§6.1.4). Concretely, it achieves this by reporting both halves of data races that occur between transactional and non-transactional code (§3.4.4) (previously, only the latter half would be reported), allowing preemption points to be identified during transactions (§3.4.1), and allowing system calls during transactions to proceed rather

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

than forcing a `CAPACITY` abort (as programmers may now need to nest other synchronization within transactions to protect against non-transactional code). Landslide requires the other STM options (`-A -S`) also to be enabled to emulate weak atomicity, partly because I know of no existing HTM system which implements it, but mostly to keep the implementation simple. As one might expect, it inflates state spaces considerably, and moreover, every single HTM benchmark in the upcoming evaluation, implemented to rely on strong atomicity, exhibits bugs under weak atomicity. Commit 68401fc implements this mode.

6.2.5 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 6.5 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 independent with each other.

Initially `int foo = 0, 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 6.5: 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 6.5'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 (§6.3.3) 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* ([God96, FG05, AAJS14], §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 [AAJS14]), and refuses to run it until the conflicting thread runs its transaction first, unless such would result in deadlock.[8] Upcoming in the evaluation, Figure 6.8 will visualize the reduction achieved in two test cases.

6.3 Evaluation

While prior work has focused on verifying transactional memory implementations themselves [DGLM09, GK08, GHS08, OST08], 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 its performance. Likewise, since Landslide's HTM 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?[9]
3. How does Iterative Deepening's (§4.2) performance compare to Maximal State Space mode (§3.1.2)?[10]

[8] 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. Commit 02b70c8 implements this fix.

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

[10] 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.

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?

6.3.1 Experimental setup

The evaluation suite comprises several unit tests hand-written by yours truly, microbenchmarks and transactional data structures from [DAS16], a transactional spinlock from [Sei13], and various combinations thereof, as follows.

1. Unit tests and microbenchmarks
   - `htm1`: The bug from Figures 2.3 and 6.1.
   - `htm2`: The fixed version as in Figure 6.4.
   - `counter`: Microbenchmark version of `htm2` which replaces the complex locking failure path with an atomic `xadd`, from [DAS16].
   - `swap`: Microbenchmark that swaps values in an array, from [DAS16].
   - `swapbug`: `swap` modified to introduce circular locking in the failure path.
   - `fig63`: Generalized version of Figure 6.5, contrived to induce as much reduction as possible from the retry sets optimization.
2. Data structure tests
   - `avl_insert`: AVL tree concurrent insertion test [DAS16].
   - `avl_fixed`: `avl_insert` with the AVL bug fixed (spoilers!!).
   - `map_basic`: Separate-chaining hashmap concurrent insertion test [DAS16].
   - `map_basicer`: `map_basic` configured with a larger initial map 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.
   - `map_basicer_mutex`: `map_basicer` simplified likewise.
3. 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 the implementations `spinlock` (from [Sei13]), `spin_fixed` (spoilers!!), and `mutex` (replaces the spinlock with a Landslide-annotated P2 mutex to reduce state space size).

The notation `testname`(K, N) denotes 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 [Blum18a]. The number of interleavings in each state space is, of course, deterministic and do not vary across runs.

I investigated the popular transactional benchmark suite STAMP [MCKO08] 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 [BMT+07], 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 [Blum18a]. 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.

6.3.2 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

buggy test	K,N	Quicksand mode cpu (s)	Quicksand mode wall (s)	Quicksand mode int's	Maximal state space mode (-M) cpu (s)	Maximal state space mode (-M) wall (s)	Maximal state space mode (-M) int's	Maximal state space mode (-M) SS size (est.)
`htm1` (assertion)	2,1	32.73	8.02	5	*9.24	*6.49	5	*12*
	2,2	50.72	9.22	9	*9.93	*7.03	9	*102*
	2,3	88.35	14.63	17	*10.03	*7.12	17	*819*
	2,4	108.31	17.44	33	*11.24	*8.29	33	*6553*
	3,1	60.63	11.63	5	*9.63	*6.70	5	*76*
	3,2	78.69	13.17	9	*9.43	*6.37	9	*3686*
	3,3	70.50	13.05	17	*10.43	*6.96	17	*176947*
	3,4	70.69	12.96	33	*11.74	*8.71	33	*8493465*
	4,1	51.83	9.85	5	*9.56	*6.63	5	*460*
	4,2	44.83	8.94	9	*9.83	*6.84	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.

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.

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 6.6 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) continue` makes the necessary preemption window extremely small (between it and `_xbegin()`), making

```
while (_retry) continue;
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 6.6: Unmodified code from `htmavl.hpp` showing the previously-unknown segfault bug Landslide found in `avl_insert`. The transaction path fails to check `_retry`, leading to data races and corruption just as in `htm1`.

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 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.[11] 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). The test case detects this with `_xtest()`. Figure 6.7 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 `spin_fixed` 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 [Chi14], 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

[11] `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 6.7: 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).

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 §6.3.3) 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 6.4, yet still got it wrong once, having to write it out by hand 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 [Intel13] or a simple spinlock [Sei13], which can be verified in isolation with smaller state spaces then trusted in turn when checking their client programs [Sim13]. §6.3.3 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 (§4.5), 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 §4.5, 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 [MQ07] to soften the trade-off both for smaller tests and for verification.

6.3.3 Verification

For the test cases with no bugs found, I sought to provide Landslide's verification guarantee (§4.3) 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 6.4'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 [ZKW15], 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 [AAJS14] 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).

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*	†*294d 5h*	†*8185793*	†*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*	†4.62×10^7	†*51191y*	†3.33×10^7	†*36393y*	†3.51×10^7
	3,1	†*5y 292d*	†2.87×10^9	†*735y 212d*	†3.35×10^8	†*97y 100d*	†1.70×10^9
map_basic	2,1	1950.48	30719	867.68	13237	367.91	5446
	2,2	†*1y 176d*	†7.55×10^7	†*11y 94d*	†9.57×10^7	†*13d 11h*	†*565334*
	3,1	†*6y 133d*	†1.85×10^9	†*8y 282d*	†7.90×10^8	†*4y 156d*	†8.88×10^8
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 (§6.2.5); "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	40

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 within).

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). 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 §6.3.2 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 their soundness. 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 (§6.2.5). 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

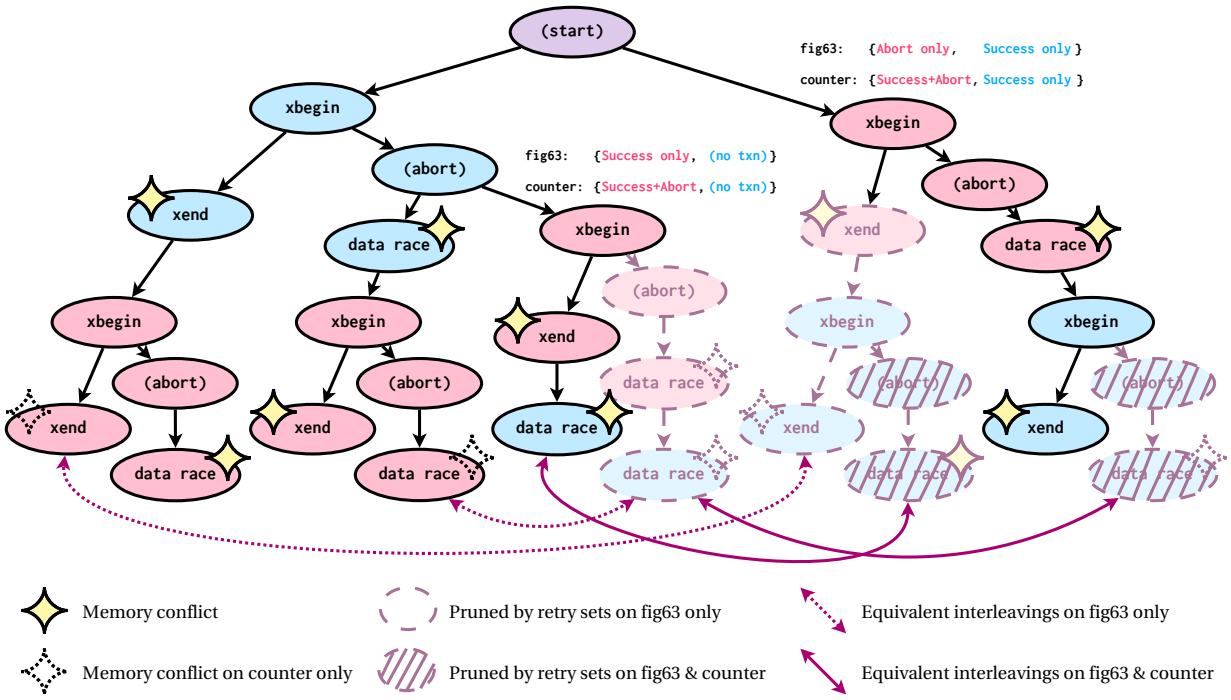


Figure 6.8: Visualization of retry set state space reduction. On both `counter(2,1)` and `fig63(2,1)`, baseline DPOR tests all 10 interleavings pictured, with the middle 2 arising from the data-race preemption point within the abort path. With the reduction enabled, after the 4th and 6th branches (i.e., 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`.

that conflict only with aborts and not with each other, and vice versa. In `fig63(2,1)`, corresponding to Figure 6.5, 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 6.8 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 synchronization 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.[12] Retry sets on the other hand identify and skip that equivalence (tech-

[12] Not pictured in Figure 6.8 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.

nically 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).

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 [DAS16], after correcting the former's atomicity bug (Figure 6.6), 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 [Sim13] 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 6.9 shows their core logic. Finally, I parameterized them over how the lock was implemented: a real-world `spinlock` implementation from [Sei13], `spin_fixed`, the same with the performance bug from §6.3.2 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 6.6'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

```
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 6.9: Abstraction reduction test cases.

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
	2,5	3417.86	76787
	3,1	8.26	129
	3,2	1403.46	30653
	3,3	*†11d 22h*	*†23136172*
	4,1	199.96	4488
	4,2	*†41d 0h*	*†66520074*
map_mutex	2,1	39.81	83
	2,2	*†26h 21m*	*†1085126*
	3,1	*†173d 17h*	*†12572187*
map_basicer_mutex	2,1	14.34	9
	2,2	147.27	2953
	2,3	12946.21	244691
	2,4	*†2d 17h*	*†1130184*
	3,1	637.81	12707
	3,2	*†102d 13h*	*†13220616*
	4,1	*†33h 33m*	*†1778661*

(b) Verifying the lock's client code.

Table 6.4: Continuation of Table 6.2, demonstrating abstraction reduction 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.

`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.

Note two curiosities: firstly, the impact that fixing Figure 6.7'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 6 interleavings per second for the former, 20 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

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

6.4.1 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_pp_abort_set()` and `update_pp_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, suggest that the latter does not necessarily subsume the former, 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[13]), so it remains an open problem to soundly prune either success paths or retry aborts while other abort codes are in play, and preserve the reduction achieved with the baseline configuration.

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

[13] See `landslide-friendly-tests/htm_causality.c`; that was a fun realization to have already halfway into implementing retry sets the wrong way at first.

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 6.5 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.

6.4.2 STM reduction soundness

In §6.3.3 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).

6.4.3 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 [CHM16, DJR17] 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 (§6.3.3), then check the rest of the program with (for example) traditional mutexes that can nest safely.

6.4.4 Relaxed memory orderings

§6.1's formalization of thread interleavings does not account for read/write reorderings possible on relaxed consistency architectures [AG96]. In fact, even after [DAS16]'s proposed fix to the atomicity protocol in Figure 6.4, 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 [Sul17b]. 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

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

is possible (reordered accesses underlined for emphasis). An acquire barrier is needed between lines 8 and 9 to solve this problem on TSO [Blum18b] (on x86, either `mfence` or `xchg/xadd`). Recent work [CSW18] 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 6.4, 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 [ZKW15], as well as proposed formal execution semantics for HTM on these architectures [DJR17, CSW18]; 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 the transactional AVL tree and hashmap [DAS16] 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)` (`landslide-friendly-tests/htm_mutex.c` in the repository) provides an example of how to use compiler intrinsics to emit the necessary barriers.

6.5 Summary

This chapter has extended the concurrency model of stateless model checking to support transactional memory, and proposed several new reduction strategies for coping with the concomitant state space explosion. I have made the following contributions:

- An approach for model checkers to emulate strongly-atomic HTM by treating transactions as globally-uninterruptible atomic sections and injecting failure without simulating rollback (§6.1.2) and applying conventional data race analysis with all transactions treated as a global lock (§6.1.3), plus an extension to also emulate weakly-atomic STM systems by relaxing the above atomicity to apply to other transactions only (§6.1.4), each section with a corresponding soundness proof of equivalence to existing systems' semantics.
- An implementation of both forms of transactional memory in Landslide, with further options to configure the variety of abort codes injected (§6.2).
- A suite of transactional benchmark programs, designed to produce interesting state spaces of varying size, bugginess, and reducibility (§6.3.1). This test suite is more appropriate for evaluating model checkers than prior work, which to date has focused on testing the raw performance of TM backend implementations, with little focus on the interactions of abort paths.
- Two previously-unknown bugs found in transactional programs from prior work (§6.3.2).
- Retry set reduction, a new state space reduction algorithm (§6.2.5) which the evaluation shows identifies and skips equivalent interleavings compared to standard DPOR, even when transactional and abort paths are fully mutually conflicting (§6.3.3).
- Demonstration of two further heuristic reduction strategies on HTM programs, abort code reduction (§6.3.3) and abstraction reduction (§6.3.3) which rely on human intuition to ensure their soundness to tackle even larger state spaces still.
- Verifications of Figure 6.4's atomicity protocol and the `lock(mutex)` transactional lock up to 2 threads, 4 iterations and 4 threads, 1 iteration for each (§6.3.3).

As concurrent systems grow in complexity to meet modern performance demands, they grow ever more fraught with opportunity for subtle bugs, and the programmer likewise faces ever more difficulty in verifying her programs. Following in the footsteps of related work for relaxed memory concurrency, this chapter has enabled stateless model checking to address yet another new technique, and concludes this thesis's contributions to the field. The upcoming chapters will offer a tour of related research problems, solved and unsolved, practical and philosophical, past and future.

Chapter 7

Related Work

It is important to draw wisdom from many different places.
If you take it from only one place, it becomes rigid and stale.
—Iroh, Avatar: The Last Airbender

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: tool implementations that target specific problem domains and algorithmic advances to make ever-larger state spaces computationally feasible. The following two subsections discuss the most closely related prior work accordingly.

7.1.1 Tools

Stateless model checking dates back to Verisoft [God97], the 1997 tool which first attempted to exhaustively explore the possible 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 MCs is Microsoft Research's CHESS [MQB+08], 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 [KAJV07], MoDist [YCW+09], SAMC [LHJ+14], ETA [SBGH11], and Concuerror [CGS13], focus on distributed systems, where concurrent events are limited to message-passing and may span

across multiple machines. R4 [JMR+15] and EventRacer [BRV15] check event-driven concurrent programs typical in mobile applications. Like Landslide, SimTester [YSR12] is a Simics [MCE+02]-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 [SBG10], 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 [YCGK08] 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 [LVT+11] 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 [Hol97] 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 [MW07]. TLC [YML99] checks formal models of concurrent program behaviour written in the specification language TLA+ [Lam02], 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 (see §2.5). Déjà Fu [WR15] 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 tools in the past few years. Relacy [Vyu11], a header-only C++ model checking library for synchronization primitives, was the first to broach this field, although it requires custom annotations for non-atomic memory accesses and does not fully model all memory reorderingss. CDSChecker [ND13] extends DPOR with a *reads-from* relation to capture most of the C++11 memory model's new behaviours. Nidhugg [AAA+15] is a checker for TSO and PSO which instruments LLVM abstract assembly, although does not yet support the C++11 memory model. rInspect [ZKW15] offers further heuristic state space reduction using buffer bounding (described below). RCMC [KLSV17] models a "repaired" version of the C++11 memory model known as RC11 [LVK+17], 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 [CMM13] 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 MC has yet proposed a satisfactory model for the "thin-air" problem [Sul17b], 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 §4.3's; see §2.3.2) 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 [Blum18d], 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.

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.

7.1.2 Algorithms

To date a number of techniques have been proposed to mitigate exponential explosion, the Sisyphean rock of stateless MC. The notion that some interleavings could lead to indistinguishable program states and be therefore redundant, known as *partial order reduction* (POR), was first proposed in [GW94] and explored in detail in [God96]. *Dynamic POR* (DPOR) was later developed in [FG05], 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 stateless MC. 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. Distributed DPOR [SBGH12] allows the exploration to be parallelized, with a minimum of overhead from redundant interleavings that would ordinarily be pruned in sequential DPOR. Optimal DPOR [AAJS14] 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 [DL15] and Maximal Causality Reduction (MCR) [Hua15] use an SMT solver [DMB08] to identify additional pruning opportunities. Implementing parallelization, 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 [TKL+12] provides additional 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 [JMR+15], used by the R4 checker mentioned above, extends DPOR to event-driven programs by separating the notion of enabled events from that of multiple threads. TaxDC [LLLG16], 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 [ZKW15] 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 §6.4.4. It also introduced a heuristic *buffer bound-*

ing technique, analogous to ICB, to mitigate the corresponding increase in state space size. The same year, Nidhugg [AAA+15] proposed a DPOR extension to account for TSO and PSO using *chronological traces*. MCR was recently extended to support relaxed memory models likewise [HH16]. Just this year, RCMC [KLSV17] 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 powerful 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 [SBG12] or by a human's wild guess), various heuristic strategies have been proposed. Preemption Sealing [BBC+10] 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) [MQ07] (§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 [CMM13] 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. Maple [YNPP12] proposed a heuristic metric for measuring the amount of "concurrency coverage", analogous to line-by-line coverage for sequential programs, and prioritizes testing interleavings which increase the coverage metric, although it makes several limiting assumptions such as preemption locality and independence of values read/written. DeMeter [GWZ+11] adapted abstraction reduction to distributed systems verification under the name Dynamic Interface Reduction, while dBug [Sim13] applied abstraction reduction to synchronization primitives, and I showed how it could be applied similarly to transactional memory in §6.3.3. 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 [SBN+97], 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. Data race detectors are largely distinguished by their particular flavour of the Happens-Before (HB) relation, as discussed in §2.3.2. Djit+ [PS03] and FastTrack [FF09] are among those which soundly avoid false positives using "Pure" HB, tracking Lamport-style vector clocks [Lam78] 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 [OC03], used in tools such as ThreadSanitizer [SI09], 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 [SES+12] is a refinement of Limited HB which avoids the most common cases of false positives, including §4.3.2's reallocation false positives. It could strike a middle ground in the bug-finding/verification tradeoff between Pure and Limited HB (§4.5) that would be a welcome enhancement in Quicksand. IFRit [EDLC+12] 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 [Law96] or Simics [MCE+02]. DroidRacer [MKM14] and CAFA [HYN+14] find data races in Android applications, using domain-specific heuristics (orthogonal to Quicksand's method) to reduce false positives. DataCollider [EMBO10] finds data races in kernel code by using hardware breakpoints and random sampling to achieve high performance.

Although many MCs listed in the previous section are content to report any data races as outright bugs, RacerX [EA03] 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 [NWT+07], 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 [Sen08] avoids false positives by requiring an actual failure be exhibited, as Quicksand does, although it uses random schedule fuzzing rather than model checking for its concurrency coverage. Portend [KZC12] 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 §4.5.4 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 (§4.3). 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 [Eck18a] and kernel [Eck18b] 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 [PRB09] has recently emerged as the most popular educational kernel (by count of top CS schools in the United States who use 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 [CPA93], 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 [CFY+06], 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 [AN12], which perhaps highlights the importance of related work on model-checking event-driven applications [JMR+15].

To my knowledge, this is the first study of model checking in an educational setting, although teaching concurrency is not itself an unstudied problem. [LBM09] 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 interleavings automatically (model checking) 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 [AM17] 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). The Deadlock Empire [HP16] is an educational introduction to concurrent programming and debugging, presented as an online puzzle-hunt of sorts, backed up by a fantasy-themed narrative. Each puzzle requires the user to manually step through the execution of multiple threads to find buggy interleavings. Though itself lacking in scientific analysis of its

educational power, it demonstrates the kind of user-friendly features and overall approach that many students expressed desire for in their survey responses (§5.3.3). Future work could extend Landslide's preemption traces to include similar interactivity.

Willgrind [Nac17] is a tool recently developed at Virginia Tech that targets a fork-join parallelism project and checks for memory errors (using the Valgrind [NS07] 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 [HM93], has received renewed attention in recent years since the launch of Intel's Haswell architecture [HKO+14], 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 [DLMN09, YHLR13, DAS16]. 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 [LCF+14] 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 [CHM16] enhances STM transactions to nest with HTM ones, while another [GHS09] adds support for relaxed memory models. Meanwhile, two recent papers [CSW18, DJR17] 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 MCs 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 [GK08, GHS08, DGLM09] 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 [SATH+06], an STM implementation built by Intel, was checked using SPIN [Hol97], verifying up to 2 threads running 1 transaction each with up to 3 memory accesses [OST08]. 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 [MCKO08] is a benchmark suite transactional programs, implemented using the OpenTM interface [BMT+07], used by many papers in the field to evaluate the performance of both STM and HTM implementations alike. As discussed

in §6.3.1, it focuses more on performance than on interesting concurrency properties. Even so, the more recent Stampede suite [NP17] 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 must minimize incidental conflicts and 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 [ZLJ16] 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 arguably 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).

The paper which originally defined weak and strong atomicity [MBL06] also 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* [LPSZ08], 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 [VTSL12] found a majority of bugs 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 [CWLS18] 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. Language-level concurrency abstractions were first introduced by the Communicating Sequential Processes (CSP) model [Hoa78], which defines a concurrent program as a set of isolated processes allowed to communicate only via blocking

input/output commands. Erlang [VWW96], an early concurrent functional language, introduced the actor model for concurrency based on the CSP model, in which threads likewise 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 [CGS13], discussed above, is a MC tool for Erlang programs. Haskell [HPJW+92] 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 [WR15] checks concurrent Haskell programs. Rust's type system supports 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 [KN18, §4.2,§10.3], and a trait system to ensure no shared state between threads by default [KN18, §10.2,§16.4]. 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 [KN18, §16.2].[1] I know of no existing model checker for Rust as of yet. The Relaxed Memory Calculus [Sul17b] proposes to extend C++ with annotations for weak memory atomics, which allows for static 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 [KTTHN14] 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 model checking, 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 [OAA09] and CoreDet [BAD+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 [LCB11] then extended the scope of determinization to include data races, while Peregrine [CWG+11] improved performance by using record-and-replay to compute a set of possible safe schedules. Parrot [CSL+13] later integrated with the aforementioned dBug [SBG10] to offer a partially-determinizing runtime scheduler that offered near-baseline performance by

[1] The author themself contributed the original design for this latter feature [Blum12b].

allowing the programmer to manually annotate speed-critical nondeterministic sections and then check the resulting state spaces using dBug's model checking as normal. Most recently, Sofritas [DELD18] 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 MC: 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 MC, which tests many possible thread execution paths under schedule nondeterminism, another popular testing approach is symbolic execution [King76], which tests many possible flow control paths under input nondeterminism. Symbolic executors abstract a program's variables and use constraint solvers such as Z3 [DMB08] to work backwards and synthesize combinations of test inputs which can lead to a failure. KLEE [CDE08], 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 [KGW10] 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 [MLR15] 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.

Formal verification

seL4 [KEH+09] 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. Verve [YH10] is another fully-verified kernel which uses type safety, rather than virtual

memory isolation, to provide strong stability and isolation guarantees. It uses a small core written in typed assembly language (TAL) [MWCG99] to provide runtime services such as stack management, interrupt handling, and garbage collection. The remaining higher-level kernel services are written in C# and compiled to TAL. Unlike seL4, whose proofs relied on human-driven interactive theorem proving, Verve's verification comes from automated type- and invariant-checking, relying on pre/postcondition and loop invariant annotations for its TAL core only, thereby imposing much less burden on its programmers. seL4's microkernel nature allows it to run untrusted code, such as drivers, in the safety of virtual memory isolation, while Verve's type safety restricts what programs can be run at all. Both approaches accept the limitation of non-parallel, uniprocessor execution. CertiKOS [GSC+16] extends seL4's approach to include full concurrency and fine-grained locking in the scope of verification, using a proof in the Coq proof assistant [inria89] 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 [NSZ+17] extended the xv6 educational kernel [CFY+06] 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. CSPEC [CK18] is another recent system for formally specifying and verifying concurrent systems using Coq, which is then extracted into an executable version in Haskell. Its type system incorporates the notion of reordering independence, much akin to DPOR, although the complexity overhead is very high: a locked counter on TSO weak memory requires 10 abstraction layers to formally specify, requiring among other things reasoning about the lock's previous owner as well as the current one. 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. Stateless model checking offers a middle ground for users to achieve slightly less formal, but still theoretically-founded, verification guarantees even for unsafe programming languages. Lastly, CompCertTSO [SVZN+13] extends the CompCert verified compiler [Ler09] to capture x86's relaxed memory semantics, guaranteeing that a program written in the ClightTSO subset of C is translated accurately to assembly with the same behaviour. Like CertiKOS, its implementation is verified in Coq. Although Landslide checks programs directly at the executable level, blind to the source code the programmer personally wrote, extending that pipeline with a certifying compiler would improve the overall verification, ensuring that Landslide was actually checking the behaviour the programmer intended.

Chapter 8

Future Work

"Journey before destination."
Some may call it a simple platitude, but it is far more. A journey will have pain and failure. It is not only the steps forward that we must accept. It is the stumbles. The trials. The knowledge that we will fail.
If we stop, if we accept the person we are when we fall, the journey ends. To love the journey is to accept no such end.

—Dalinar Kholin, Oathbringer

Each of the previous three chapters concluded by discussing specific limitations, listing concrete and immediate ways to address them with existing techniques (§4.6, §5.4, §6.4). Meanwhile, this chapter takes a broader interpretation of "future work", namely, how might future Landslides solve research problems I didn't even pose to begin with.

8.1 User-friendliness

In the user study survey (§5.3.3), students most often complained that interpreting Landslide's preemption traces to diagnose and understand their bugs was too difficult. While the understanding step of course requires human intuition, there is certainly room to improve the diagnosis step beyond just showing the user one static HTML table. Related works such as Symbiosis [MLR15] can find a minimal difference between the failing trace and a non-failing one, which would allow the user to effortlessly pinpoint which preemptions are closest to the true root cause. Further, using ICB [MQ07] to show the user a minimum set of preemptions necessary to expose the bug could help her narrow down possible diagnoses more quickly. Finally, the preemption trace itself could be interactive, allowing the user to click and drag to reorder thread transitions and see how the interleaving would change, or to click and drag preemption points across the source code to figure out how much need to be atomic.

State space size management remains an issue, as ever. While Quicksand's professed selling point is that the user need input only a CPU budget, at the same time, pruning uninteresting branches out of the overall state space would allow Quicksand to achieve

more meaningful results in that same budget. Theoretical advances in state space reduction come out every year (§7.1.2), but the user's human intuition can also contribute if properly encouraged. For the P2 tests (§5.1.3), I configured Quicksand by hand to issue appropriate `without_function` commands to Landslide (§3.4.1), and even more still for the HTM tests (§6.3.1), but a user writing her own tests would have to configure DPOR by hand. A more mature version would coach the user to decide which functions focus the test on, using state space estimation to give an idea of expected testing time, and listing the variables/locations of DPOR's memory conflicts to help her identify more candidate functions to potentially ignore. Finally, Landslide could integrate with a version control system to do incremental testing, automatically analyzing the functions touched by each patch, and heuristically prioritizing preemption points therein to quickly check small updates on top of an already-verified codebase.

8.2 Verification

In its current form, Landslide is limited to testing only those behaviours that the test case it's hooked up to can generate. The most obvious limitation of this is resource exhaustion scenarios: stateless model checking simply cannot handle tests long-running enough to exhaust system memory (succumbing to exponential explosion long before), so cannot exercise any flow control that involves `malloc()` failing, for example. This specific issue could be solved using by-hand annotations to block all preemption points until just before exhaustion, or by extending Landslide to inject allocation failures at `malloc()` callsites (akin to `_xbegin()`). However, these require the user to realize in advance that she should worry about allocation-failure bugs, and to configure Landslide specifically to target them. In general, a mature testing tool should not require the user to know in advance where her bugs might be before being able to conduct an effective test. In future work, a stateless model checking framework could collaborate with its user to semi-automatically design better tests. Concurrency coverage metrics such as proposed by Maple [YNPP12] could be extended to capture possible behaviours under any test input, not just within the fixed state space under one given test, and symbolic execution frameworks such as Contessa [KGW10] could search for possible inputs to suggest changes which might expose them. In cases where not-yet-covered conditional branches require certain function return values, such as `malloc()` failing, the tool could offer to add failure injection with the user's approval, or at least prompt her to write a new test with that as a subgoal.

In the case study of submitted P2 bugs (§5.3.2), I noticed several submissions of a common pattern in which Landslide overlooked the `thr_exit()` use-after-free bug: they all manually recycled the exited thread's stack using an internal free-list, rather than calling `free()` or `remove_pages`, so Landslide's heap-checker failed to see anything out of the ordinary about the subsequent writes to it. Explicit annotations about the custom free-list's invariants could make Landslide treat it like `free()` and catch the errant write, but that begs the question of knowing about the bug in advance. Catching this bug would require extending the test case to `thr_fork()` a new thread to reuse the old stack and suffer data corruption from the stray write. In this case, the user might happen to find this

just by increasing the number of threads/iterations, which is just good testing practice, but in general, automatically inferring such data structure invariants to suggest better assertions remains an open problem. Future work might combine Landslide's ability to prove correctness for finite K, N parameters with a formal induction proof that generalizes the result, finding a minimum number of threads necessary to expose all behaviours (at least three in this example), although I know of no related work which addresses the second half of this problem. Likewise, §6.3.3's abstraction reduction requires the user to believe that `lock(spinlock)` and friends correctly express the mutual exclusion property in informal code[1]. Future work could further check such tests against external formal specifications to make the abstraction reduction more trustworthy.

As important as Quicksand's convergence theorem (§4.3), the HTM equivalence proofs (§6.1), and other soundness results from prior work are, they fall short of fully end-to-end formal verification in terms of trusting the output of stateless model checkers in practice. In addition to the issue of test cases, there is also currently no guarantee that Landslide's concrete implementation matches the theoretical properties. My personal faith in Landslide's implementation comes from years of empirical observation: inspecting small state spaces (such as Figure 6.8) by hand to check that preemption points show up in the right places and that DPOR prunes the expected equivalences; as well as checking that the relationships between larger state spaces satisfy expected invariants, such as when testing two sets of preemption points, one a subset of the other, or testing the same but one with an additional reduction applied, the one's resulting state space must be smaller than the other's. Nevertheless, discrepancies may still lurk undiscovered: the failure injection bug described in §6.3.1 took until implementing retry set reduction to discover, well after publishing the first experimental results in [Blum18a]. As the concurrency model becomes more complex, more opportunities arise for the implementation to deviate from what the soundness theorem actually proves. Kernels like seL4 [KEH+09] and CertiKOS [GSC+16] and concurrency-aware compilers like CompCertTSO [SVZN+13], all formally verified against external specifications, provide important pioneering steps in this direction.[2]

8.3 Heuristics

In my experience, the WBE algorithm (used in Landslide for estimating number of interleavings) consistently underestimates, being often seen to count (nearly) monotonically up toward the true value as exploration progresses, while the RE algorithm (used in Landslide for estimating total execution time) can be unstable and erratic, bouncing wildly among orders of magnitude even in the space of ten or fewer adjacent interleavings. These flaws are especially visible in the `avl_fixed` transactional verification results (Table 6.2). Both estimation algorithms use the known structure of the tree as a model for

[1] In Figure 6.9(a), the `if (!_xtest()) thr_yield(-1);` part required expert knowledge of HTM semantics to write correctly: an unconditional yield there would cause the transaction to abort every time, verifying mutual exclusion in the failure path only.

[2] Of course, being verified themselves, these projects have no need for a verified Landslide, but checking unverified (e.g., student) code with such would still be an improvement.

the unknown (see §3.4.3), but make no use of domain-specific knowledge such as which threads were chosen to run at each preemption point. For example, consider the root of the left subtree in Figure 2.4, thread 1's `tmp1++`. At that moment thread 1 has 1 more instruction left to execute, while thread 2 has 3. If thread 1's last instruction is chosen to run next, there can only be one way to run thread 2's 3 steps thereafter ($\binom{3+0}{0} = 1$); if thread 2's first instruction is chosen to run next, there will be 3 ways to interleave thread 1's last among thread 2's remaining 2 ($\binom{2+1}{1} = 3$). In general, the largest child subtree at any preemption point will be the one resulting from running the thread with the most transitions left, or more formally, fixing $N + M = C$, $\binom{N+M}{N}$ is maximized when $N \approx M$. WBE and RE could both be relatively easily extended to incorporate this observation, using knowledge from branches already tested to guess (still heuristically) appropriate values for N and M. Predicting more advanced aspects of state space structure, such as subtrees pruned by sleep set reduction (§3.4.2; ★4 in Figure 3.6), would likely require analyzing DPOR's memory conflicts as well.

There is much room to expand Landslide's use of heuristics to balance verification with fast bug-finding. Landslide's maximal state space mode (§6.3.3) currently requires the user to decide in advance whether she thinks a full verification is possible within the time limit, and supply `-M` if so, sacrificing some bug-finding power up front (or just running the test twice). Quicksand could make this decision automatically, always dedicating one of its available CPUs to the maximal state space while the rest run Iterative Deepening as normal. Currently, Landslide explores each state space sequentially, but if extended with Distributed DPOR [SBGH12], Quicksand could also dynamically allocate more or fewer CPUs to the maximal state space according to its ETA. Likewise, Quicksand could incorporate ICB [MQ07] to get the best of both worlds: when testing smaller data-race jobs, start them in ICB mode to begin with, and when the ETAs of larger jobs (including the maximal one) grow too large, downgrade them to ICB to try to at least get a partial verification for that preemption point set rather than discarding it entirely. ICB could also be extended to include HTM failure injections as part of its model, counting them either as part of the overall preemption bound or under a separate abort bound. State space estimation under ICB currently counts only DPOR's tagged branches that don't get skipped under the preemption bound, estimating that bound's state space size and resetting whenever the bound increases; it could also be extended to be ICB-aware, counting branches skipped due to the bound separately and computing two estimates at once, which would in turn allow Quicksand to make more informed decisions about when to use ICB.

Machine learning has become popular recently as a magic cure-all for many programming problems, but is also notorious for amplifying any biases in its training set (or in the minds of its developers) in unpredictable ways that have been shown to harm people of color [Spe17, BG18], women [Lea18, Das18], and transgender people [Key18]. In the domain of concurrency testing, such bias would be unlikely to result in interpersonal discrimination, but it would translate to finding only types of bugs already seen in the training data, abandoning any novel bug patterns to further obscurity. Granted, Iterative Deepening and ICB also deprioritize "deeper" bugs, as measured by number of unique or total preemptions, respectively, but when a bug is not found these strategies also enable the user to understand the nature of the partial verification. Machine learning

would be more appropriate for improving state space estimation, whose output is already a black-box oracle from the user's perspective, and where the worst case of being wrong is that Quicksand prioritize its jobs suboptimally. In-progress estimates fluctuate depending on many factors, such as interleaving-dependent flow control, and I suspect deeper patterns may exist among multiple different state spaces, for example, from the same test program with different K, N parameters. For any future researchers who wish to study such patterns, I have made the estimation logs from the HTM experiments (§6.3) freely available at `https://github.com/bblum/thesis-htm-logs`. Nevertheless, I firmly believe that using machine learning as a bug-finding heuristic without compensating extremely carefully for its inherent biases would run counter to this thesis's central tenets of transparency and user agency. Any overall search heuristic must be designed to support the human user first and foremost, that she may readily understand partial verification results and participate effectively in the test case refinement strategies discussed upchapter.

Chapter 9

Conclusion

"If you are a god, Zeus, as the stories claim, then why did you create evolution? Why did you make a world that can only grow through cruelty and pain?"
For a moment—surely this meant death was near—she thought she heard him answer: "My child, I was shaped by the gods that came before me, as you were shaped by me. The choice I had was between creation and oblivion, life and death. And I chose life, because any life is better than no life, because as long as there is life, there is hope—if not for us, then for some generation to come."
"Then how are you a god, if you can offer so little?" she whispered, feeling death creep closer.
"I am a god because I take upon myself the burden of creation," the statue replied.
"Then we are all gods," Alexandra said, and pushed the button.

—The Talos Principle

Concurrency testing is, in a way, a microcosm for research in general: an exponentially large problem that invariably thwarts they whoever try to solve it "perfectly"; it bears fruit only to one who can find their own human-oriented meaning in partial results and compromises. I speak of course literally on one side of this analogy, figuratively on the other. This thesis has demonstrated such meaning for stateless model checking, choosing perhaps one of the most challenging demographics of programmers in need of concurrency testing, the undergraduate operating systems student, as its target audience, ultimately helping 73% of participating students to find bugs in their code, 56% to fix any such bugs, and 34% to fix all such bugs, before their project deadlines. Along the way, I justified Landslide's applicability to problems beyond CMU's walls as well by showing positive results at other universities, and by addressing a new concurrency model developed by the hardware industry in just the last decade. As I write this I am also collaborating with 15-410 staff to integrate Landslide as an official part of the course grading infrastructure, so in some sense the naïve wish to make an "automatic TA" that drew my attention to grad school to begin with has also been fulfilled.

Let us revisit the first half of the thesis statement, which in §1.2 I mentioned would serve as the work's overarching theme: that by *combining theoretically-founded automatic reduction techniques and user-informed heuristic ones,* stateless model checking can meet human testing needs. How have each of the major contributions of this work embodied this theme? Let me also briefly address: What lasting lessons have we learned from each,

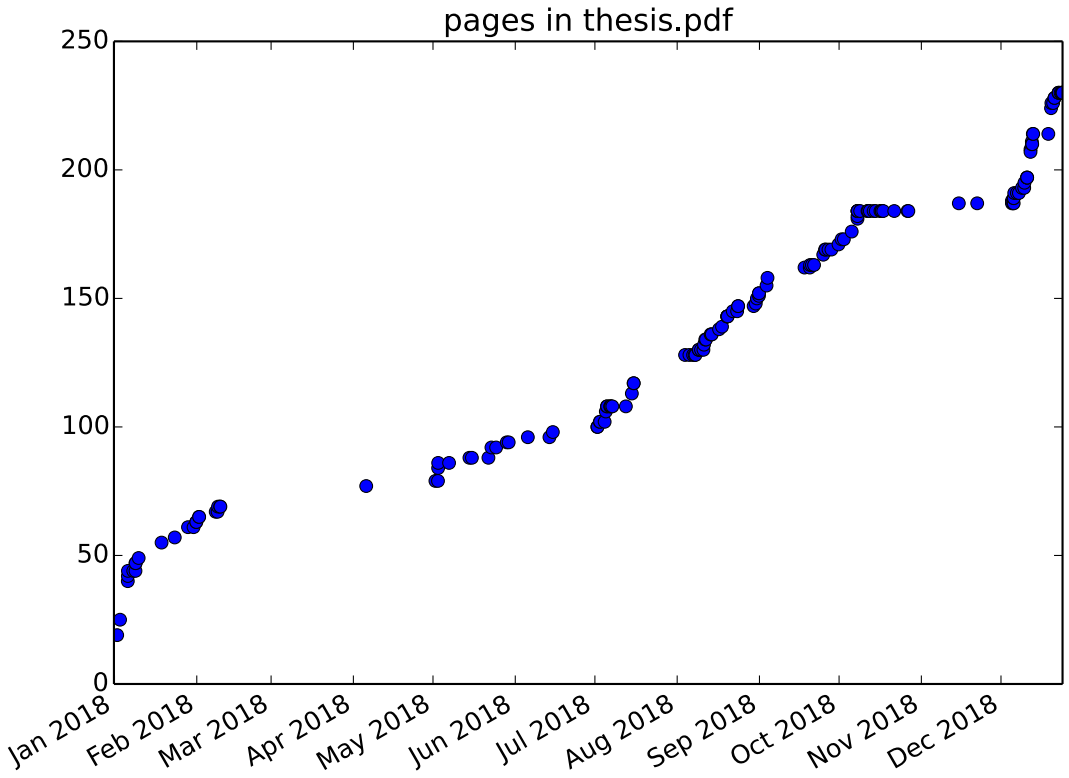
which any future research building hereupon would be wise to heed?

- In Chapter 4, the soundness of data-race-driven Iterative Deepening combined with Quicksand's reliance on state space estimation effectively balanced formal verification with heuristic bug-finding speed better than any prior work. Even as modern programming languages such as Rust [KN18] threaten to obsolete the data-race preemption point proof by statically forbidding data races to begin with, Iterative Deepening's general approach remains relevant alongside ICB [MQ07], each providing a different tradeoff between bug-finding and completion time, and future model checkers should offer a mixture of different kinds of partial verification results depending on the user's wishes.
- In Chapter 5, Landslide's behind-the-scenes yield-loop blocking algorithm allowed it to automatically instrument all student implementations with no false positives (what false positives did exist were minor unrelated Landslide bugs, all fixed quickly whenever reported), and exposing partial results such as data race reports was shown to sometimes help students reach deeper understanding. Although high-level undergraduate operating systems courses represent a very small fraction of all programmers in higher education, let alone all those in the world, Landslide's adventures in 15-410 argue that computer science classes should more intentionally teach principled debugging techniques and tools rather than let students struggle to reach correctness independently. Also, any future debugging tool should heed my students' survey feedback and take friendly user interface design more seriously than I did with Landslide.
- In Chapter 6, the equivalence of HTM's interface to failure injection and global locks allowed Landslide to avoid both performance overhead and state space explosion when simulating aborts, and heuristic use of abort code reduction and abstraction reduction allowed verifying algorithms and data structures up to thread/iteration counts previously infeasible. Emulating complex execution semantics with simpler, already-understood primitives, and the retry set reduction, both allowing substantial yet sound state space reduction, are the obvious take-aways for any future model checker wishing to tackle HTM. I expect the latter should easily apply to other opportunities for failure-injection beyond transactions as well, such as memory allocation, filesystems, and distributed systems. More in general, the verification section demonstrated the value of compartmentalizing one's verification efforts, checking modules separately against a common, well-understood interface to reach yet further up the exponential cliff.

For as long as people have written science fiction books, we have fantasized about robots that can make any complex intuitive decision a human can in addition to the superhuman arithmetic ability that comes standard with silicon. Recent trends in machine learning have pursued this aesthetic, using statistics and pattern recognition to provide convincing output even on never-before-seen inputs, but few such systems are designed to also alert the user when their output confidence is low. As tempting as it is to fantasize an oracle capable of telling for certain whether your program is free of concurrency bugs, that would be a disservice to both user and machine: the human skill that artificial intel-

ligence should really try to emulate is knowing its limits and asking for help when it gets confused. Here I have demonstrated that stateless model checking can realistically fulfill this ideal, and I envision a future of concurrency verification where human and program may cooperate to debug and verify more complex software than ever before.

It's become fashionable among my peers lately, whenever a new exploit is announced in the news, to declare that security is doomed, that computers were a mistake, that we should all retire and become llama farmers, and so on. There is no doubt that we are burdened with massive technical debt from the mistakes made during computer science's infancy. Nevertheless, safety-oriented programming languages and type systems are growing ever more popular, and meanwhile formal verification techniques to check existing code, written in the old unsafe ways, grow ever more powerful. Landslide's ability to reach impressionable student minds is my little contribution in this long battle. I have hope that we may one day live in peace with our silicon creations, understanding and respecting their limitations just as they complement ours.


pages in thesis.pdf
250
200
150
100
50
0
Jan 2018
Feb 2018
Mar 2018
Apr 2018
May 2018
Jun 2018
Jul 2018
Aug 2018
Sep 2018
Oct 2018
Nov 2018
Dec 2018

Bibliography

Only a doctor of philosophy, Darth.

—Robert Marsh

[AAA+15] 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. 1, 176, 178

[AAJS14] Parosh Aziz 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. 1, 17, 62, 66, 77, 100, 149, 158, 164, 177

[ABB+86] 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. 22

[AD52] T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *Annals of Mathematical Statistics*, 23(2):193–212, 06 1952. 129

[Ada80] Douglas Adams. *The Restaurant at the End of the Universe*. Pan Books, 1980. 52

[AG96] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996. 3, 10, 172

[AM17] 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. 180

[AN12] 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. 180

[ATLM+06] 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. 12, 146

[BAD+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. 1, 183

[BBC+10] 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. 1, 17, 58, 76, 107, 178

[BG16] 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. 1, 79, 91, 98, 108, 111, 112, 149

[BG18] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In Sorelle A. Friedler and Christo Wilson, editors, *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*, volume 81 of *Proceedings of Machine Learning Research*, pages 77–91, New York, NY, USA, 23–24 Feb 2018. PMLR. 190

[Bir89] Andrew D. Birrell. An introduction to programming with threads. Technical Report 35, Digital Equipment Corporation Systems Research Center, 1989. 29

[BKMN10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Proceedings of the Fifteenth Edition of ASPLOS on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 167–178, New York, NY, USA, 2010. ACM. 113

[Blum12a] Ben Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2012. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf. 1, 3, 30, 39, 42, 44, 45, 59, 115, 138

[Blum12b] Ben Blum. Concurrency primitives in sync.rs with static enforcement. https://github.com/rust-lang/rust/issues/3145, August 2012. 183

[Blum16] Ben Blum. Soundness proofs for iterative deepening. Technical Report CMU-PDL-16-103, Carnegie Mellon University, September 2016. http://www.pdl.

cmu.edu/PDL-FTP/associated/CMU-PDL-16-103.pdf. 91

[Blum17] Ben Blum. A boring follow-up paper to "Which ITG stepcharts are turniest?" titled, "Which ITG stepcharts are crossoveriest and/or footswitchiest?". In *Proceedings of the 11th ACH SIGBOVIK Conference in Celebration of Harry Q. Bovik's 2^6th Birthday*, sigbovik '17, pages 54–61, Pittsburgh, PA, USA, 2017. ACH. 71

[Blum18a] 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. 155, 160, 189

[Blum18b] Ben Blum. Demonstration of the need for a barrier in TSX failure paths. https://gist.github.com/bblum/85f64858a35a74641be228f191144911, 2018. 172

[Blum18c] Ben Blum. what happens if you make even the smallest syscall during HTM. https://gist.github.com/bblum/10fbcc8ed170192c987fe400031c3e68, 2018. 155

[Blum18d] Ben Blum. *Practical Concurrency Testing, or: How I Learned to Stop Worrying and Love the Exponential Explosion*. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2018. http://reports-archive.adm.cs.cmu.edu/anon/2018/CMU-CS-18-128.pdf. 1, 176

[BMT+07] 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. 160, 181

[BO01] Randal E. Bryant and David R. O'Hallaron. Introducing computer systems from a programmer's perspective. In *Proceedings of the 32nd Technical Symposium on Computer Science Education (SIGCSE)*, Charlotte, NC, February 2001. ACM. 20

[BO10] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Addison-Wesley Publishing Company, USA, 2nd edition, 2010. 20

[Boe11] Hans-J. Boehm. How to miscompile programs with "benign" data races. In *Hot Topics in Parallelism*, HotPar'11, pages 3–3. USENIX Association, 2011. 107

[Boe12] Hans-J. Boehm. Position paper: Nondeterminism is unavoidable, but data races are pure evil. In *Relaxing Synchronization for Multicore and Manycore Scalability*, RACES '12, pages 9–14. ACM, 2012. 107

[BRV15] 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. 147, 176

[CBM10] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. Gambit: Effective unit testing for concurrency libraries. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 15–24, New York, NY, USA, 2010. ACM. 110

[CD88] Eric C. Cooper and Richard P. Draves. C threads. Technical Report CMU-CS-88-154, Carnegie Mellon University, June 1988. http://reports-archive.adm.cs.cmu.edu/anon/1988/CMU-CS-88-154.ps. 29

[CDE08] 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. 184

[CFY+06] Russ Cox, Cliff Frey, Xiao Yu, Nickolai Zeldovich, and Austin Clements. Xv6, a simple Unix-like teaching operating system. https://pdos.csail.mit.edu/6.828/2018/xv6.html, 2006. 180, 185

[CGS13] 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. 1, 104, 175, 183

[Chi14] Vamsi Chitters. leveldb-tsx. https://github.com/vamsikc/leveldb-tsx/, 2014. Spinlock implementation found at `ext/xsync/include/spinlock-rtm.hpp`. 162

[CHM16] 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. 172, 181

[CK18] Tej Chajed and Frans Kaashoek. Verifying concurrent software using movers in CSPEC. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation*, OSDI '18, pages 307–322, Carlsbad, CA, USA, 2018. USENIX Association. 185

[CMM13] 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. 1, 72, 73, 101, 149, 176, 178

[CPA93] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The Na-

chos 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. 180

[CSL+13] 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. 183

[CSW18] 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. 149, 172, 181

[CWG+11] 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. 1, 183

[CWLS18] 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. 182

[DAS16] Mario Dehesa-Azuara and Nick Stanley. Hardware transactional memory with Intel's TSX. Technical Report CMU-CS-18-124, Carnegie Mellon University, May 2016. http://reports-archive.adm.cs.cmu.edu/anon/2018/CMU-CS-18-124.pdf. 149, 159, 168, 172, 181

[Das18] Jeffrey Dastin. Amazon scraps secret AI recruiting tool that showed bias against women. *Reuters*, October 2018. https://www.reuters.com/article/idUSKCN1MK08G. 190

[DELD18] 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. 184

[DGK09] Aleksandar Dragojević, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM. 146

[DGLM09] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Electronic Notes in Theoretical Computer Science (ENTCS)*, 259:245–261, December 2009. 158,

181

[DJR17] 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. 172, 181

[DL15] 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. 1, 17, 149, 177

[DLMN09] 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. 181

[DMB08] 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. 177, 184

[EA03] 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. 1, 14, 19, 107, 113, 127, 179

[Eck18a] David A. Eckhardt. Project 2: User level thread library. `http://www.cs.cmu.edu/~410-f18/p2/thr_lib.pdf`, 2018. 3, 4, 11, 20, 180

[Eck18b] David A. Eckhardt. Pebbles kernel specification. `http://www.cs.cmu.edu/~410-f18/p2/kspec.pdf`, 2018. 3, 20, 23, 180

[Eck18c] David A. Eckhardt. Synchronization (2). `http://www.cs.cmu.edu/~410-s18/lectures/L08b_Synch.pdf`, 2018. 117

[Eck18d] David A. Eckhardt. Paradise Lost. `http://www.cs.cmu.edu/~410-s18/lectures/L13a_Lost.pdf`, 2018. 116

[Eck18e] David A. Eckhardt. Deadlock (1). `https://www.cs.cmu.edu/~410-s18/lectures/L11_Deadlock.pdf`, 2018. 26

[Eck18f] David A. Eckhardt. Personal communication, 2018. 23

[ED09] Erez Eisen and Amit Duvdevani. Can't stop. In *Legend of the Black Shawarma*, Infected Mushroom. Perfecto Records, 2009. Track 5. 97

[EDLC+12] 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 Program-*

ming Systems Languages and Applications, OOPSLA '12, pages 467–484, New York, NY, USA, 2012. ACM. 179

[EMBO10] 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. 1, 179

[EWD35] Edsger W. Dijkstra. Over de sequentialiteit van procesbeschrijvingen. Circulated privately. http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD35.PDF, not dated. 20

[FB13] Eric Feigelson and G. Jogesh Babu. Beware the Kolmogorov-Smirnov test! https://asaip.psu.edu/Articles/beware-the-kolmogorov-smirnov-test, 2013. 129

[Fey64] Richard Feynman. The motion of planets around the sun, March 1964. 9

[FF09] 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. 1, 4, 70, 76, 79, 83, 110, 152, 178

[FG05] 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. 16, 59, 60, 62, 66, 76, 77, 92, 151, 158, 177

[Fis22] 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. 131

[GÖ31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. 14

[GHS08] 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. 158, 181

[GHS09] 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. 181

[GK08] 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. 158, 181

[GNU16] The GNU Foundation. X86 transaction memory intrinsics. https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html, 2016. 13, 146, 155, 156

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag, Berlin, Heidelberg, 1996. 62, 66, 77, 158, 177

[God97] 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. 2, 13, 175

[GSC+16] 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. 185, 189

[Gun14] Haryadi Gunawi. CMSC 23000 (CS 230): Operating systems (Autumn 2014). https://www.classes.cs.uchicago.edu/archive/2014/fall/23000-1/, 2014. 24

[GW94] Patrice Godefroid and Pierre 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. 13, 177

[GWZ+11] 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. 178

[Hac14] Mike Hachman. Intel finds specialized TSX enterprise bug on Haswell, Broadwell CPUs. *PCWorld*, 2014. 146

[HH16] 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. 1, 178

[HKM+88] John H. Howard, Michael L. Kazar, Sherri G. Menees, David A. Nichols, M. Satyanarayanan, Robert N. Sidebotham, and Michael J. West. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):51–81, February 1988. 118

[HKO+14] 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*, 34(2):6–20, 2014. 3, 13, 146, 181

[HM93] 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. 3, 12, 145, 181

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21(8):666–677, August 1978. 182

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. 16, 91, 98, 176, 181

[HP16] Petr Hudeček and Michal Pokorný. The deadlock empire. https://deadlockempire.github.io/, 2016. 180

[HPJW+92] 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. *ACM SIGPLAN Notices*, 27(5):1–164, May 1992. 104, 183

[Hua15] 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. 1, 17, 63, 100, 111, 149, 177

[Huf54] David A. Huffman. The synthesis of sequential switching circuits. Technical Report 274, Massachusetts Institute of Technology, January 1954. 30

[HYN+14] 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. 179

[IEEE09] IEEE Standards Association. Information technology – Portable Operating System Interface (POSIX®) Base Specifications, Issue 7. Standard, International Organization for Standardization, Geneva, CH, September 2009. 29

[inria89] inria. The Coq proof assistant. https://coq.inria.fr/, 1989. 185

[Intel13] Intel. Hardware lock elision overview. https://software.intel.com/en-us/node/683688, 2013. 146, 163

[Intel17] 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. 146

[Intel18] Intel. Transactional synchronization extensions (TSX) overview. https://software.intel.com/en-us/node/524022, 2018. 13, 146, 149

[JMR+15] 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. 3, 147, 176, 177, 180

[Jos16] Anthony Joseph. CS162: Operating systems and systems programming. https://cs162.eecs.berkeley.edu/, 2016. 24

[KAJV07] 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. 1, 104, 175

[KdS1868] Ibn Khallikān and William MacGuckin de Slane. *Ibn Khallikan's Biographical Dictionary*, volume III, pages 71–73. Oriental translation fund of Great Britain and Ireland, 1868. https://books.google.com/books?id=eeE80B81GfgC. 1

[KEH+09] 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. 184, 189

[Key18] Os Keyes. The misgendering machines: Trans/HCI implications of automatic gender recognition. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):88:1–88:22, November 2018. 190

[KGW10] 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. 1, 184, 188

[Kha1274] Ibn Khallikān. وفيات الأعيان وأنباء أبناء الزمان *(Deaths of Eminent Men and History of the Sons of the Epoch)*. 1274. 1

[King76] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. 147, 184

[KLSV17] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proceedings of the 46th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2(POPL):17:1–17:32, December 2017. 100, 111, 147, 149, 176, 178

[KN18] Steve Klabnik and Carol Nichols, with contributions from the Rust community. *The Rust Programming Language*. No Starch Press, Incorporated, 2018. https://doc.rust-lang.org/1.31.0/book/. 10, 104, 183, 194

[Kor85] Richard E. Korf. Iterative-deepening-A: An optimal admissible tree search. In

Proceedings of the 9th International Joint Conference on Artificial Intelligence - Volume 2, IJCAI '85, pages 1034–1036, San Francisco, CA, USA, 1985. Morgan Kaufmann Publishers Inc. 83

[KSTW06] 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. 67

[KTTHN14] 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. 183

[KZC12] 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. 80, 89, 107, 109, 110, 113, 179

[Lam78] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, July 1978. 18, 19, 60, 70, 178

[Lam02] Leslie Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002. 176

[Law96] Kevin P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 1996(29es), September 1996. 35, 76, 179

[LBM09] 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. 180

[LCB11] 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. 1, 183

[LCF+14] 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. 181

[Lea18] Susan Leavy. Gender bias in artificial intelligence: The need for diversity and gender theory in machine learning. In *Proceedings of the 1st International Workshop on Gender Equality in Software Engineering*, GE '18, pages 14–16, New York, NY, USA, 2018. ACM. 190

[Ler09] Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, July 2009. 185

[LHJ+14] 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. 104, 175

[LLLG16] 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. 1, 177

[LPSZ08] 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. 182

[LTQZ06] Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting atomicity violations via access interleaving invariants. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XII, pages 37–48, New York, NY, USA, 2006. ACM. 1, 25

[Lu14] Chris Lu. unaligned writes are bad. `https://gist.github.com/kalenedrael/323bab7e624f88d0edde`, 2014. 10

[LVK+17] 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. 176

[LVT+11] 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. 176

[Maz87] 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. 16

[MBL06] Milo Martin, Colin Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17–17, July 2006. 146, 152, 182

[MCE+02] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Fors-

gren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002. 35, 76, 176, 179

[MCKO08] 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. 160, 181

[MKM14] 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. 179

[MLR15] 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. 1, 184, 187

[MQ07] 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. 1, 17, 71, 72, 76, 100, 101, 105, 110, 164, 178, 187, 190, 194

[MQB+08] 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. 1, 98, 111, 141, 175

[Mun11] Randall Munroe. Significance. `https://xkcd.com/882/`, 2011. 131

[MW07] Paul McKenney and Jonathan Walpole. What is RCU, fundamentally? `https://lwn.net/Articles/262464/`, 2007. 176

[MWCG99] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 21(3):527–568, May 1999. 185

[Mye85] 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. 67

[Nac17] William Naciri. Bug finding methods for multithreaded student programming projects. Master's thesis, Virginia Tech University, June 2017. 181

[ND13] 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. 147, 176

[NP17] 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. 182

[NS07] 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. 181

[NSZ+17] 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. 185

[NWT+07] 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. 1, 179

[OAA09] 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. 1, 183

[OC03] 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. 18, 19, 152, 179

[OST08] 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. 158, 181

[Ous16] John Ousterhout. CS 140: Operating systems (Winter 2016). `http://web.stanford.edu/~ouster/cgi-bin/cs140-winter16/index.php`, 2016. 24

[PPTT90] 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. 22

[PRB09] 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. 3, 4, 23, 180

[PS03] 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. 70, 76, 152, 178

[R18] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2018. 129, 131

[RG01] 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. 146

[SATH+06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *Proceedings of the Eleventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM. 146, 181

[SBG10] 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. 49, 52, 90, 98, 141, 176, 183

[SBG12] 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. http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-12-113.pdf. 4, 67, 76, 79, 84, 88, 178

[SBGH11] 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. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-11-113.pdf. 104, 175

[SBGH12] 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. http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-101.pdf. 1, 17, 101, 103, 177, 190

[SBN+97] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, November 1997. 18, 47, 89, 99, 178

[Sei13] Austin Seipp. Transactional memory on Haswell. https://gist.github.com/thoughtpolice/7123036, 2013. 159, 163, 168

[Sen08] 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. 109, 110, 179

[SES+12] 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. 19, 179

[SFJ09] Michael J. Sullivan and Elizabeth Fong-Jones. Pathos reference kernel. Unpublished (implementation private to CMU 15-410 course staff), 2009-2018. 20

[SI09] 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. 4, 19, 79, 83, 110, 113, 152, 179

[Sim13] Jiri Simsa. *Systematic and Scalable Testing of Concurrent Programs*. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2013. http://reports-archive.adm.cs.cmu.edu/anon/2013/CMU-CS-13-133.pdf. 104, 163, 168, 178

[SMAT+07] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore, and Bratin Saha. Enforcing isolation and ordering in STM. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 78–88, New York, NY, USA, 2007. ACM. 146, 152

[Spe17] Robyn Speer. How to make a racist AI without really trying. https://blog.conceptnet.io/posts/2017/how-to-make-a-racist-ai-without-really-trying/, July 2017. 190

[SS87] F. W. Scholz and M. A. Stephens. K-sample Anderson–Darling tests. *Journal of the American Statistical Association*, 82(399):918–924, 1987. 129

[SSN+09] 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. 149

[Sul17a] Michael J. Sullivan. pebwine – the worst thing I have ever done. Unpublished (implementation private to CMU 15-410 course staff), 2017. 23

[Sul17b] Michael J. Sullivan. *Low-level Concurrent Programming Using the Relaxed Memory Calculus*. Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, USA, 2017. http://reports-archive.adm.cs.cmu.edu/anon/2017/CMU-CS-17-126.pdf. 1, 30, 172, 176, 183

[SVZN+13] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: A verified compiler for relaxed-

memory concurrency. *Journal of the ACM*, 60(3):22:1–22:50, June 2013. 185, 189

[SZ18] Fritz Scholz and Angie Zhu. *kSamples: K-Sample Rank Tests and their Combinations*, 2018. R package version 1.2-8. https://CRAN.R-project.org/package=kSamples. 129

[TDB14] 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. 72

[TKL+12] 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. 177

[TT18] Jason Torchinsky and David Tracy. This is why you can't unlock a car door if someone is trying to open it at the same time. *Jalopnik*, August 2018. https://jalopnik.com/1827837275. 1

[Tur37] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 1937. 10, 73

[Var07] Vargomax V. Vargomax. Generalized Super Mario Bros. is NP-complete. In *Proceedings of the 1st ACH SIGBOVIK Conference in Celebration of Harry Q. Bovik's 2^6th Birthday*, sigbovik '07, pages 87–88, Pittsburgh, PA, USA, 2007. ACH. 94

[VTSL12] 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. 182

[VWW96] Robert Virding, Claes Wikström, and Mike Williams. *Concurrent Programming in ERLANG (2nd Edition)*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1996. 10, 104, 183

[Vyu11] Dmitry Vyukov. Relacy race detector. https://github.com/dvyukov/relacy, 2011. 176

[WR15] 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. 72, 176, 183

[YCGK08] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Ef-

ficient 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. 91, 98, 176

[YCW+09] 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. 1, 104, 175

[YH10] Jean Yang and Chris Hawblitzel. Safe to the last instruction: Automated verification of a type-safe operating system. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 99–110, New York, NY, USA, 2010. ACM. 184

[YHLR13] 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. 181

[YML99] 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. 176

[YNPP12] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 485–502, New York, NY, USA, 2012. ACM. 1, 178, 188

[YSR12] 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. 1, 176

[ZKW15] 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. 3, 17, 91, 147, 149, 164, 172, 176, 177

[ZLJ16] 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. 182