# Practical Concurrency Testing

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

Ben Blum

January 2017

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.*

## Abstract

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

In this thesis, I propose to make stateless model checking more practical for human use by way of several new techniques. I have built Landslide, a stateless model checker specializing in student projects for undergraduate operating systems classes. Landslide includes a novel algorithm for automatically managing multiple state spaces according to their estimated completion times, which I will show quickly finds bugs should they exist and also quickly verifies correctness otherwise. I will 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. Finally, I will explore broader impact by extending Landslide to test some real-world programs and to be used by students at other universities.

# Contents

# Chapter 1

# Introduction

Modern computer architectures have turned to increasing CPU core count, rather than clock speed, to improve processing power [54]. To take advantage of multiple cores for performance, programmers must write software to execute *concurrently* – using multiple *threads* which execute multiple parts of a program's logic simultaneously. However, when threads access the same shared data, they may interleave in unexpected ways which change the outcome of their execution. When an unexpected interleaving produces undesirable program behaviour, for example, by corrupting shared data structures, we call it a *concurrency bug*. Concurrency bugs are notoriously hard for programmers to find and debug because the specific thread interleaving required to trigger them arises at random during normal execution, and often with very low probability.

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

*Stateless model checking* [30] is an alternative way to test for concurrency bugs, or to verify their absence, which provides more reliable coverage, progress, and verification than stress testing. A stateless model checker tests a program by forcing it to execute a new unique thread interleaving on each iteration of the test, capturing and controlling the randomness in a finite state space of all possible interleavings.

Unfortunately, the size of these state spaces is exponentially proportional to the size of the tested program. For even moderately-sized programs, there may be more possible ways to interleave every thread's every instruction than particles in the universe. Accordingly, a programmer who wants her test to make reasonable progress through the state space must choose a subset of ways that her threads could interleave, focusing on fully testing that subset, while ignoring other possibilities she doesn't care about. However, it is difficult to choose a subset of thread interleavings that will produce a meaningful, yet feasible test. Until com-

puters can automatically navigate this trade-off in some intelligent way, programmers will continue to fall back to the random approach of stress testing.

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

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

*Thanks to the new algorithms, heuristics, and concurrency models I have developed, stateless model checking is an appropriate and accessible concurrency testing technique for programmers in both educational and real-world settings.*

I have built Landslide [7], a stateless model checker for thread libraries and kernels, and I have developed some techniques for automatically choosing the best thread interleavings to test and for automatically instrumenting operating system kernels in an educational setting. This thesis will comprise three major contributions:

- **Meaningful state spaces (Chapter 3).** I will present *Iterative Deepening*, a new algorithm for navigating the trade-off in how many preemption points to test at once. Iterative Deepening incorporates state space estimation [70] to decide on-the-fly whether each state space is worth pursuing, and uses data race analysis [66] to find new preemption point candidates based on a program's dynamic behaviour. This section will include a large evaluation of the technique, comparing its performance to three prior work approaches across 600+ unique tests. I will show that Iterative Deepening of preemption points outperforms prior work in terms both of finding bugs quickly and of completely verifying correctness when no bug exists.

  This work is **completed** and was published at OOPSLA 2016.

- **Educational use (Chapters 4 and 5).** For the past three semesters, I have offered a fully-automated version of Landslide to students in 15-410, CMU's undergraduate Operating System Design and Implementation class [22, 23], for use as a debugging aid during the thread library project. I will continue these user studies, and use the data to evaluate the suitability of stateless model checking in an educational setting.

  So far, the fully-automatic testing mode is available only for 15-410 thread library projects. To prove these techniques are relevant beyond CMU's walls, I will extend Landslide to handle Pintos kernel projects from other universities [61]. I will then collaborate with those schools to deploy Landslide to their operating systems students.

  This is **ongoing work**; I have run the user study for 4 semesters so far and am proposing to continue them and extend the existing evaluation, as well as deploy a Pintos port.

- **Transactional Memory (Chapter 6).** Transactional Memory (TM) is a relatively new

concurrent programming technique [37] which is not yet addressed by modern model checkers. I will extend Landslide's concurrency model to support both hardware (HTM) and software (STM) variants of TM, and test several "real-world" TM programs.

This will be **new work**, involving some implementation effort, some proofs, and designing a new evaluation.

# Chapter 2

# Background

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

Thesis proposal note: I've written this section with the intent of reusing it in the thesis. The other chapters, being necessarily more impermanent, are aimed directly at you (my thesis committee) as the audience. This section is written with the intention for programmers of any level to read.

## 2.1 Concurrency Bugs

### 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.

**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 [50], and weak memory consistency models [4] – 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 [52], the simultaneity of threads is largely irrelevant to the program's behaviour. However, "thread-unsafe" languages such as C, C++,

[1] With some exceptions in kernel-level programming, which I discuss later.

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

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

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

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

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

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.

**Identifying bugs.** Even if a program's behaviour is nondeterministic, that does not necessarily mean it has a bug. After all, many programs use random number generation to intentionally generate different outputs. We say a *concurrency bug* occurs when one or more of a program's nondeterministic behaviours is both *unanticipated* and *undesired*. Most often, a concurrency novice who programs with shared state will consider the possible interleavings where one thread's access sequence occurs entirely before the other's, but neglect to consider intermediate outcomes in which the threads' access sequences are interleaved. Consider the program in Figure 2.1: Any output between 2 and 2000 is possible[2], but whether this constitutes a bug is a matter of perspective. Was the program written to count to 2000, or was it written to compute a randomized distribution? In this thesis, we make no attempt to reason about the "intent" of programs, so we further restrict *concurrency bug* to denote a program behaviour which is mechanically identifiable, according to commonly-accepted notions of what programs behaviours are always bad. Bug conditions include assertion failures, memory access errors (i.e., segmentation fault or bus error), heap errors (i.e., use-after-free or overflow), deadlocks, and infinite loops (which must be identified heuristically [73]).

### 2.1.2 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

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

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

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

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

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

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

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

Each such abstraction provides certain semantics about what thread interleavings can arise surrounding their use. When building a tool for testing concurrent programs, one may include some computational understanding of the behaviour of any, or all, of these abstractions. Annotating a certain abstraction's semantics treats it as a trusted concurrency primitive in its own right, and allows the testing tool to reduce the possible space of interleavings (or the set of false positive data-race candidates reported, etc.), at the cost of increasing the implementation and theoretical complexity of the analysis. In this thesis, I will consider locks and descheduling to be the only concurrency primitives, and assume the others listed above are implemented using those as building blocks (an exercise for the reader [23]).

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

### 2.1.3 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 [37] 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 coherence algorithms, or in software, via library calls and a log-based commit approach. Software transactions (STM) [3] can be used on any commodity processor, but must impose runtime overhead associated with logging. Hardware transactions (HTM) [20, 77] 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 [36] 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 instructions can be used to synchronize a simple shared access without locking overhead in the common case, using GCC's compiler intrinsics [29].

In this thesis, I 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. My treatment of TM will be largely agnostic to whether a transactional interface is backed by HTM or STM, and I will extend it to cover STM as well, as discussed in Section 6.

## 2.2 Stateless Model Checking

Model checking [30] 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 the state space (or the CPU budget) is exhausted. During each execution, it forces threads to execute serially, thereby confining the program's nondeterminism to controlled thread switches. Some model checkers explicitly store the set of visited program states as a means of identifying equivalent interleavings [38]. This approach is called *stateful* model checking. In this thesis, I focus on *stateless* model

**(All threads)**

```
if ((status = _xbegin()) == _XBEGIN_STARTED) {
    if (too_much(foo)) {
        _xabort();
    } else {
        foo++;
        _xend();
    }
} else if (status != _XABORT_EXPLICIT) {
    mutex_lock(&m);
    foo++;
    mutex_unlock(&m);
}
```

Figure 2.3: An example program demonstrating the various HTM primitives. If the transaction in the top branch aborts, whether from a conflict or from the programmer's intention, execution will revert to the return of `_xbegin`, and `status` will be assigned an error code indicating the abort reason. The programmer can then use explicit synchronization, such as a mutex, to resolve the conflict.

checking, which instead analyzes the sequence of execution events to avoid a prohibitive memory footprint. Henceforth I will abbreviate "stateless model checking" simply as "model checking" for brevity.

**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 [31]. 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 [25]. Dynamic analysis, our approach, identifies program behaviours that are definitely wrong, so each bug report is accompanied by concrete evidence of the violation. Assertions, segfaults, use-after-free of heap memory, and deadlock are examples of such failures we check for, although a checker may also include arbitrary program-specific predicates.

**Preemption points.** During execution, a model checker identifies a subset of the program's operations as "interesting", i.e., points at which interrupting the current thread to run a different one is likely to produce different behaviour. These points, called *preemption points*, may be identified by any combination of human intuition and machine analysis. Typical preemption points include the boundaries of synchronization APIs (e.g., `mutex_lock`) or accesses to shared variables. Considering that at each preemption point multiple threads exist as options to run next, the set of possible ways to execute the program can be viewed as a tree. The number of preemption points in each execution define the depth of this tree,

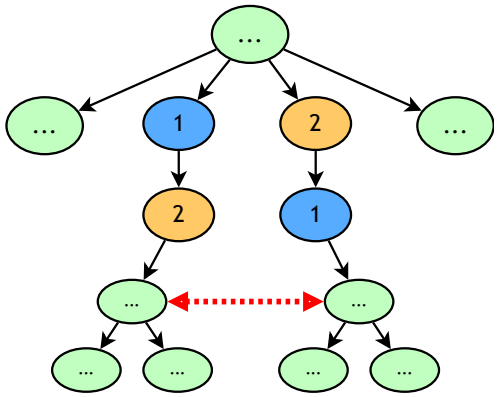



Figure 2.4: DPOR identifies independent transitions by different threads which can commute without affecting program behaviour. Here, if the transitions marked 1 and 2 have no shared memory conflicts, the states marked with the red arrow are guaranteed identical. Hence, only one of the subtrees need be explored.

and the number of threads available to run define the branching factor. Hence, in a program with $n$ preemption points and $k$ threads available to run at each, the state space size is $O(n^k)$. Addressing the scaling problem in this exponential relation is the central research problem for all model checkers.

**Reduction techniques.** Dynamic Partial Order Reduction [28] (henceforth, DPOR) is the premier algorithm for mitigating the exponential explosion that arises as program size increases. It identifies equivalent execution sequences according to Mazurkiewicz trace theory [53], and tests at least one execution from each equivalence class. Intuitively, if two thread transitions between preemption points do not conflict on any shared resource access, reordering them produces an equivalent interleaving, i.e., the same program behaviour. Figure 2.4 shows a part of an execution tree in which the operations by threads 1 and 2 are independent, and hence only one of the subtrees need be tested. Over the years, researchers have developed many enhancements to DPOR, such as Optimal DPOR [1], parallelizable DPOR [71], SAT-directed model checking [19], Maximal Causality Reduction [39], and DPOR for relaxed memory architectures [78].

**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 [55] 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

```
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 2.5: 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.

completed. Preemption sealing [6] 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.

## 2.3 Data Race Analsyis

Data race analysis [64] identifies pairs of unsynchronized memory accesses between threads. Two instructions are said to race if:

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

relation, described below).

In Figure 2.5, the pairs of lines marked with comments (A1 and A2, B1 and B2) race.

A data race analysis may be either *static* (inspecting source code) [25] or *dynamic* (tracking individual accesses arising at run-time) [66]. 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).

**Variants of Happens-Before.** Most prior work focuses on *Happens-Before* [46] as the order relation between accesses. [72] and [58] 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 2.5(a). We call such unreported access pairs *false negatives*. [58] introduces the *Limited Happens-Before* relation, which will report such potential races by considering only blocking operations like `cond_wait` to enforce the order. However, consider the similar program in Figure 2.5(b), in which the access pair ceases to exist in the alternate interleaving. Limited Happens-Before will report all potential races, avoiding false negatives [66], but at the cost of necessarily reporting some such *false positives*.

In Chapter 3, we use the Limited Happens-Before relation for our analysis. The justification for this is that, while stand-alone data-race analyses must avoid inundating the user with false alarms [25], my work incorporates data-race analysis in an internal feedback loop, and reports only directly observed failures to the user. Hence, I accept some overhead from false positives for the sake of more thorough testing.

## 2.4 Education

In this thesis I 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

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 Introduction to Computer Systems [10]. Both P2 and P3 are built using the *Pebbles* system call specification, outlined in Table 2.1

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

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

**P2**

The thread library project [23] has two main components: implementing concurrency primitives, and implementing thread lifecycle and management routines. The required concurrency primitives are as follows:

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

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

The thread lifecycle/management routines are as follows:

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

- `thr_join(tid, statusp)` blocks the calling thread until the thread with the specified thread ID exits, then returns, collecting its exit status.

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

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

#### P3

In P3, students implement a kernel which provides the same system calls shown in Table 2.1, previously provided by the reference kernel. Pebbles adopts the Mach [2] distinction between *tasks*, which are resource containers, and *threads*, each of which executes within a single task. This requires less implementation complexity than the more featureful Plan 9's `rfork` [62] or Linux's `clone` models.

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

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

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

### 2.4.2 Pintos

The Pintos kernel architecture [61] is used at several universities, including Berkeley, Stanford, and the University of Chicago. The Pintos basecode implements a rudimentary kernel, consisting of a context switcher, round-robin scheduler, locking primitives, and program

loader. upon which students add more features in several projects. Most relevant to this thesis, the basecode provides the following functions/libraries, among others:

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

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

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

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

Some schools further offer a virtual memory project, extending the provided VM with a frame table and supplemental page table and fault handler [35, 60], or supplemental HTTP server and `malloc` assignments [41]. Being largely architectural/algorithmic projects rather than concurrency-oriented ones, I am not concerned with these assignments in this thesis. 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.

# Chapter 3

# State Spaces

In this chapter, I will present a new algorithm for model checkers to automatically choose which preemption points to use to test reasonably-sized state spaces. This is the first of the three projects I am proposing for this thesis, and has already been published at OOPSLA 2016 [9]. The thesis will largely restate the paper, and also go into more detail on some ideas which didn't make it into the paper, such as measuring the variance in each experimental data series. Here in the proposal I will simply summarize the content of the paper.

## 3.1 Motivation

This project is motivated by my experience contributing to Parrot [16], a (partially) determinizing runtime for multithreaded programs which allowed some nondeterminism in performance-critical sections to achieve high performance. We combined Parrot with dBug [68], a model checker, to check the correctness of the nondeterministic sections of programs which remained. Unfortunately, dBug was able to finish a state space for fewer than half of these programs in our evaluation in a 24-hour time limit, even after the reduction Parrot provided. In that project, our model checker was limited to a fixed set of preemption points; in other words, it would always test different programs with the same granularity of thread interleavings. Consequently, the resulting completion time for each test would vary wildly, unpredictably falling on one side or the other of our arbitrary time limit.

I argue that this is the wrong usage model for model checkers to present to users who wish to test for concurrency bugs. Users want concrete bug reports (obviously), or in absence of those, they want some assurance of the program's correctness. However, users do not live in a $\infty$-sized fantasy land [74], and are generally willing to wait only for some fixed finite time. If full verification of all possible interleavings is not possible within that time, a *partial verification* guarantee for some smaller state spaces is better than an "I don't know" verdict for one single state space.

## 3.2 Design

Hence, I propose a more user-friendly framework for model checking, in which many different state spaces are tested, and higher priority is given to state spaces which appear possible to complete before the user's patience runs out or their project is due. At the heart of this framework is an algorithm I call **Iterative Deepening**, by analogy with the eponymous technique in chess artificial intelligence [45]. In both chess and model checking, Iterative Deepening makes progressively "deeper" searches of an exponentially-sized state space, repeatedly increasing the size of a subset state space to be explored, until a prescribed time limit is exceeded. While in chess, the size bound is determined by a single depth parameter, the size of model checking state spaces is determined by any number of different combinations of preemption points being enabled or disabled.

**Quicksand.** I have built a wrapper program, called Quicksand, which manages the execution of multiple simultaneous Landslide instances. Quicksand relies on state space estimation [70] to decide at runtime how to prioritize these jobs. Landslide can provide these estimates to a reasonable accuracy before actually testing a large fraction of interleavings for a given state space. Quicksand seeks to maximize completed state spaces, as each one serves as a guarantee that all interleavings possible with its preemption points were tested. Moreover, because Iterative Deepening treats the set of preemption points as mutable, it can add new preemption points reactively based on any runtime analysis. Landslide begins with a statically-coded set of synchronization API preemption points (such as used by dBug [68]), and during testing, runs a dynamic data-race analysis [24, 58, 66] to identify new candidate preemption points.

**Iterative Deepening.** With a limited CPU budget, Quicksand must avoid running tests that are likely to be fruitless. Hence, it separates the available preemption point sets into a set of *suspended* jobs (partially-explored state spaces with high ETAs), and a set of *pending* jobs (untested ones with unknown ETAs). When Landslide reports an ETA exceeding the time limit, Quicksand compares with other pending and suspended jobs to find another one more likely to complete in time. The Iterative Deepening algorithm implements this comparison, presented formally in Algorithm 1. Its main feature is understanding that when one Landslide instance is testing a superset of preemption points compared to another, the state space is correspondingly guaranteed to contain a superset of possible thread interleavings.

**Data-race preemption points.** As mentioned, runtime data race analysis produces new potential sites at which preempting the program may produce new behaviour. 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, shown formally in Algorithm 2. The new state spaces may expose a failure, in which case Landslide reports a data-race bug, or complete successfully, indicating a benign or false-positive data race. They may also uncover a new data-race candidate entirely, in which case Quicksand iteratively advances to a superset state space containing PPs for both racing access pairs. Being constrained by a CPU budget, time may run out before completing a data race's associated state space, in which case Quicksand reports a potential false positive that the user must handle.

**Algorithm 1:** Suspending exploration of a state space in favour of a potentially smaller one.

**Input**: $j_0$, the currently-running job
**Input**: $\mathscr{P}$, the list of pending jobs, sorted by decreasing heuristic priority
**Input**: $\mathscr{S}$, 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.
```

## 3.3 Verification

### 3.3.1 Convergence of Iterative Deepening

When the a test's state spaces are small enough that Quicksand can exhaustively check all of them within a given CPU budget, the resulting verification claim turns out to be equivalent to a naïmodel checker which preempts on every memory access. In this way, the Iterative Deepening algorithm provides the "best of both worlds" for the tradeoff mentioned in Section 3.1: when tests are too large, Quicksand falls back on prioritization heuristics to find bugs quickly; when tests are tractable, Quicksand's verification is equal to the strongest from any single-state-space technique.

The theorem statement is as follows:

**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.*

The proof is available at [8].

**Algorithm 2:** Adding new jobs with data-race PPs.

**Input**: $j_0$, the currently-running job
**Input**: $\mathscr{J}$, the set of all existing (or completed) jobs
**Input**: $\alpha$, an instruction reported by the MC as part of a racing access pair
**if** $\forall j \in \mathscr{J}$, *PPSet(*$j_0$*)* $\cup\ \alpha \not\subseteq$ *PPSet(j)* **then**
  AddNewJob(PPSet($j_0$) $\cup\ \alpha$, HeuristicPriority($\alpha$))
**end**
**if** $\forall j \in \mathscr{J}$, *PPSet(j)* $\neq \{yield, \alpha\}$ **then**
  AddNewJob($\{yield, \alpha\}$, HeuristicPriority($\alpha$))
**end**

### 3.3.2 Pruning False Positives

I have also proved a (somewhat less remarkable) theorem concerning the soundness of a new tactic for pruning certain types of false-positive data race candidates (under Limited HB). A "malloc-recycle" false positive occurs when two threads seem to race on the same memory address, but only because the `malloc`ed block containing that address was `free`d and reallocated in between. If the transitions of the two threads were reordered, the address collision would disappear and there would be no race. The proof is available at [8]; the theorem statement is:

**Theorem 2** (Soundness of eliminating malloc-recycle candidates)**.** *If a malloc-recycle candidate is not a false positive, DPOR will test an alternate thread interleaving in which the accesses can race without fitting the malloc-recycle pattern.*

## 3.4 Evaluation

I evaluated Quicksand on the product of the following two questions: For an arbitrary fixed CPU budget...

- ...does Quicksand find more bugs than a single-state-space (SSS-MC) approach...
- ...does Quicksand provide more total verifications than SSS-MC...

...where the SSS-MC control is configured to preempt...

- ...on synchronization API boundaries only?
- ...on every single shared memory access?

I also conducted a separate experiment focused on the issue of *nondeterministic* data-races: candidates that require model checking to expose in the first place, which would be false-negatives during a single-execution analysis.

I also also ran several different configurations of Quicksand in each of these experiments, alternately using Limited Happens-Before or Pure Happens-Before for its data-race analysis. Both of these cases outperformed all the other cases of SSS-MC, so this was mostly useful for comparing the two techniques against each other.

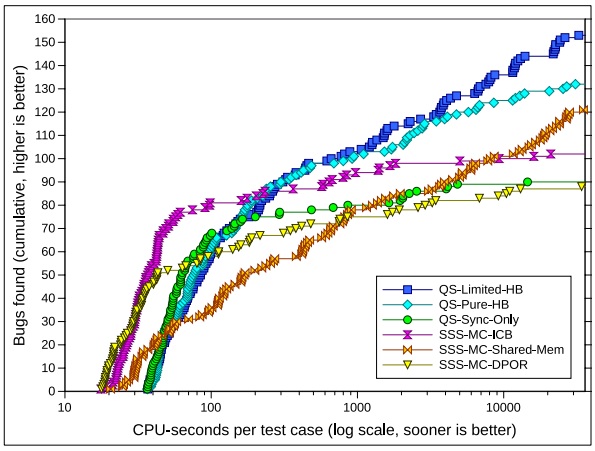
### 3.4.1 Setup and Test Suite

The test suite for all these experiments was 157 student projects: 79 P2 thread libraries from CMU, and 78 Pintos kernels from Berkeley and the University of Chicago (Section 2.4). I paired each submission with several test cases, 6 for each P2 and 3 for each Pintos. In total there were 629 unique pairs of a test case and a student project, henceforth called simply "tests". (Savvy mathematicians will note that doesn't quite add up – some of the Pintoses were incomplete, and could only run 1 or 2 tests.)

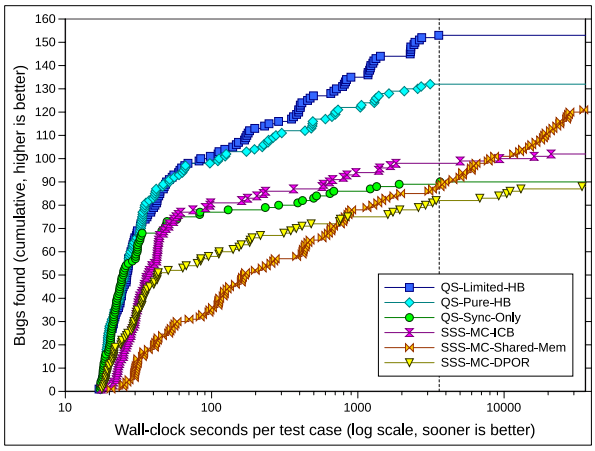
For each of the 629 tests, I ran Quicksand and SSS-MC for 10 CPU-hours each in each of the configurations named above. Quicksand, being inherently parallel, was given 10 CPUs for 1 wall-clock hour, while SSS-MC control experiments were given 10 wall-clock hours on 1 CPU (which is being charitable: comparing CPU-time gives the controls perfectly efficient hypothetical scaling from parallelization).

### 3.4.2 Results

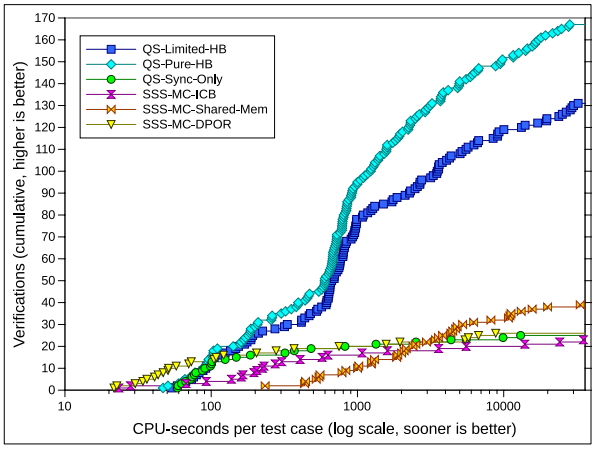
The results are shown in Figure 3.1. The first 3 graphs all plot a cumulative distribution of their respective achievements, with elapsed time on the X-axis (as a log scale, to emphasize detail in the "easy" tests). The captions largely explain each result. To summarize, Quicksand outperforms the state-of-the-art on all metrics, excepting the impact of parallelization start-up overhead for very small CPU-time limits. The Limited HB data-race strategy generally finds bugs faster when they exist, but suffers on total verifications compared to Pure HB, because of its false positives which create excess redundant state spaces for Landslide to check.



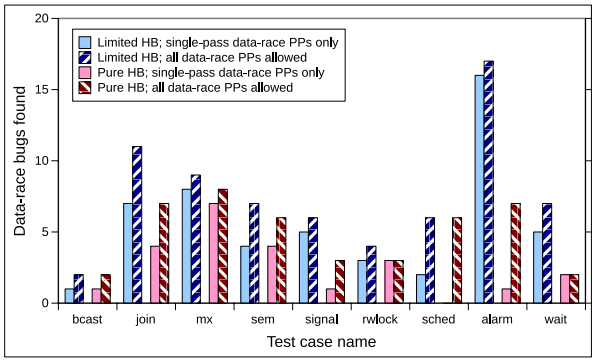

(a) Bugs found as a function of elapsed CPU time. After the break-even point at ∼200 seconds, Quicksand outperforms all control experiments.




(b) Bugs found by elapsed wall-clock time. This presentation emphasizes that the "break-even" point in (a) is just an artifact of parallelization start-up overhead. Quicksand is parallelized tenfold; the vertical line indicates its 1 hour limit.




(c) Total verifications provided as a function of CPU time. The lines for SSS-MC-ICB and SSS-MC-DPOR are artificially penalized to exclude tests with data races, as preempting on sync APIs alone is not sufficient to verify those. SSS-MC-Shared-Mem, however, is not penalized thus; its failure here is due to the massive computational overhead from so many preemption points.




(d) Plot of how many data-race bugs were found using "single-pass" data-race candidates versus "nondeterministic" ones. With both Limited and Pure HB approaches, the MC's ability to find new data-race candidates in obscure thread interleavings led to more bugs found in all test cases. Between Limited and Pure HB, Pure HB is more reliant on this phenomenon, as Limited HB can often find more potential races on the first pass.

Figure 3.1: Many kernels died to bring us this information.

# Chapter 4

# Pebbles

This chapter proposes my plan to evaluate Landslide's effectiveness as a debugging aid for students in an educational setting. This is the second of the three projects I am proposing for this thesis, and is currently ongoing work.

## 4.1 Motivation

In my MS thesis [7], I solicited students at CMU's Operating Systems Design and Implementation class (henceforth, "15-410") to volunteer at the end of the P3 project to annotate their kernels and try debugging them with Landslide. However, the annotation burden undermined Landslide's purpose: the only students willing to spend free time on manual instrumentation were biased to be those who were already doing well in the class, and hence least likely to benefit from Landslide's debugging potential. (Actually, even the best 15-410 students still have concurrency bugs, but in principle, an educational tool should reach the more struggling students, the so-called "middle" and/or "bottom" of the class.) Requiring annotations hurts Landslide's case as a grading tool, as well: TAs need to understand the kernel to begin with in order to annotate correctly, and while achieving such understanding they may as well grade it by hand, as before.

Since then, I've extended Landslide to support testing Project 2 (P2) thread libraries (Section 2.4.1) as well. Because P2 mandates specific function names for the project's internal APIs – most importantly, for the concurrency primitives – Landslide can automatically annotate arbitrary student implementations with no manual effort required of the user (whether student or TA). The addition of Quicksand and Iterative Deepening (Chapter 3) partly fulfills this purpose, freeing student attention from the issue of which state spaces to test. This chapter will detail my further techniques and evaluation which are specific to educational use.

## 4.2 Implementation Details

Every stateless model checker must make some assumptions about the tested programs' concurrency model [56]. However, arbitrary programs may break conventional disciplines of concurrent programs, while still being bug-free. For example, thread communication via ad-hoc `yield` loops may appear to automated tools as a possible infinite loop, livelock, or deadlock. This is especially true of student code, written by people who are just learning concurrent programming discipline for the first time, and/or written under the time pressure of a project deadline. To be an effective tool for struggling students, a model checker should somehow coax adversarial programs to fit its concurrency model, to effectively test for real bugs, rather than rejecting them outright on some stylistic or disciplinary grounds.

Fully-automatic instrumentation of student P2s has been no walk in the park. I have equipped Landslide with several powerful algorithms and heuristics for handling the most common anti-patterns in student submissions.

- **Yield-blocking.** Landslide recognizes open-coded busy-wait loops used for ad-hoc synchronization, and is able to treat threads as blocked (or "disabled" in model-checking parlance), avoiding getting stuck in an infinitely-long interleaving (or "cyclic state space") which should never arise during normal execution. The heuristically-driven algorithm is as follows:
  - Whenever a thread performs a `yield`, `xchg`, or other atomic instruction, Landslide increments a per-thread counter to track its (supposed) busy-wait loop iterations. A thread's counter is reset any time it performs some "interesting" activity not likely to appear in a true busy-wait loop: any condvar, semaphore, or rwlock invocation (but not mutexes), and the beginning or end of any thread library function (create, join, or exit). [1] The consequence of any this heuristic's inaccuracy is that an unusual wait-loop might be classified as an infinite loop bug anyway.
  - When a thread's loop counter reaches some heuristic limit (10 for `yield`s, 100 for `xchg`s), Landslide speculatively marks the thread blocked (or "disabled", in model-checking parlance), just as though it had invoked `deschedule`. It also retroactively disables that thread at all preceding `yield`s/`xchg`s in that sequence, which prevents DPOR from trying to use each as a preemption point, and avoids a state space explosion (by a factor of the heuristic yield limit). [2]
  - When retroactively disabling a thread across all its preceding loop iterations, Landslide's state space estimator must account for the "pruning" of duplicate subtrees at those (now disabled) preemption points. If in any previous thread interleaving, DPOR tagged the now-yield-blocked thread to interleave at another thread's preemption point, the estimator would have included that potential subtree in its computation of how much unexplored state space exists. Accordingly, in this case Landslide will invert the estimation algorithm, including propagating the reduced subtree size all the way to the state space's root. [3]

[1] Implemented in `check_user_yield_activity` and `check_user_xchg` in `user_sync.c`.
[2] Implemented in `update_blocked_transition` in `user_sync.c`.
[3] Implemented in `untag_blocked_branch` in `estimate.c`.

- Landslide can precisely identify when another thread may trigger the yield-blocking one to fall out of its loop, by analyzing the shared memory conflicts involving only accesses performed in the loop. At any such memory conflict, Landslide will reenable the yielding thread. (If the other thread's access does not fulfill the yielding thread's wait condition, the latter will just re-trigger the heuristic and become blocked again.) [4]

This approach is similar to the Fair-Bounded Search algorithm in [14], although it avoids a major assumption of the latter (threads `yield` if and only if not making progress), and also avoids the need to iteratively deepen the yield bound by instead fixing it as a heuristic constant.

- **False-positive deadlock avoidance.** In contrast to its treatment of data races, Landslide must never report a false-positive *bug*. If its heuristics falsely identify a thread as blocked, and all other threads are truly blocked, waiting on some progress from that thread, Landslide would report a deadlock bug, and confuse students horribly.

  The yield-loop heuristic assumes that "too many" yields or atomics should not arise during normal, non-looping execution of thread library routines. Though extremely rare, this assumption can be violated by an adversarial student submission. More often, Landslide can falsely block threads in the special case of `mutex_test` (see [9]), where it uses preemption points within the implementation of `mutex_lock` itself. False deadlocks can also arise from the heuristic blocking of ICB [55] (used in Quicksand's control experiments, Section 3.4).

  When a deadlock arises under conditions where one or more threads are heuristically blocked, Landslide attempts to refute it as a false positive by artificially unblocking all heuristically-blocked threads. [5] Landslide then repeats this process a heuristic constant number of times (128), allowing the program that many chances to make progress before proclaiming deadlock. [6] Note that this heuristic cannot miss true deadlocks as false negatives: if the deadlock is true, each thread will immediately trigger the yield-blocking heuristic again, bringing the system back into deadlocked state as many times as necessary to exhaust the heuristic limit.

- **Lock hand-off.** A common, though discouraged, idiom for implementing thread destruction involves one thread "handing off" ownership of a mutex to another. That thread will then release the lock with no corresponding acquire in its own execution. Although Landslide cannot easily recognize at what point the latter thread's accesses are protected by that lock for the sake of data-race analysis, potentially leading to false positive races, it must release the lock in its bookkeeping to avoid false *negatives* if any later access that should be protected by that lock isn't. When releasing a lock, if absent from the current thread's lockset, Landslide searches the locksets of all other existing threads, and releases it there. Landslide can instead optionally be configured to treat

[4] Implemented in `check_unblock_yield_loop` in `user_sync.c`.

[5] If any threads are ICB-blocked, I prioritize waking those before trying to wake any yield-blocked threads. Waking all threads at once here can lead to unsoundness.

[6] Implemented in `try_avoid_fp_deadlock` in `arbiter.c`.

lock hand-off as a style warning or as an outright bug, as a matter of discipline. [7]

However, some of the less common offenses are both more difficult to handle algorithmically, and also more worrying from a pedagogical point of view. For the following cases, I configured Landslide to abort, and warn the student that they must find a better solution before it could test their code (in other words, I promoted these patterns to be treated as bugs).

- **Busy-wait loops** containing neither `yield` nor `xchg` (nor any other atomic instruction), such as `while (!other_thread_ready) continue;`. This blurs the line between anti-pattern and concurrency bug: because it does not yield the CPU, a uni-processor machine must wait for the next timer tick (several milliseconds!) before making any progress; also, because it does not use atomic instructions, an optimizing compiler may reorder or even delete the loop's accesses. Landslide also cannot easily identify it as similar to message-passing, appearing indistinguishable from a thread-local infinite computation, which is of course impossible to judge for halting [73].

  In such a case, Landslide will issue a bug report with the special message: *I have run a loop in [function name] an alarming number of times. This version of Landslide cannot distinguish between this loop being infinite versus merely undesirable. Please refer to the "Synchronization (2)" lecture.*

- **Recursive mutex use** (i.e., locking the same mutex twice in the same thread, then subsequently unlocking it twice). While it would not be difficult for Landslide's locksets to support recursive locking (using a nesting counter instead of a boolean flag), that would assume the corresponding mutex implementation provides the same semantics, which is risky business with student code. Furthermore, recursive locking is not an obvious solution to any of P2's challenges; far more often, it arises when a student's mutex tries to `malloc` some internal state, which itself requires a mutex for safe allocation, which can lead to stack overflow and a crash.

  In such a case, Landslide will issue a bug report with the special message: *This version of Landslide cannot debug recursive implementations of mutex_lock. Please examine this stack trace and determine for yourself whether it indicates a bug.*

## 4.3 Landslide as a Debugging Tool

In the Spring 2015, Fall 2015, and Spring 2016 semesters, I've made Landslide (with Quicksand) available to 15-410 students during the last week of P2. I introduced stateless model checking in a guest lecture given to the class, and required students to pass the thread library hurdle [23] to ensure a minimum of basic functionality necessary to use Landslide.

To analyze its effectiveness as a debugging tool for students, I configured Landslide to record a snapshot of the inputs and results of each use by the students: which options were used (test program and time limit), a snapshot of the student's code, and the result of the

[7] Implemented in `lockset_remove` in `lockset.c`.

test (completed, timed out, bug found, or ctrl-C'ed). I manually interpreted the snapshots to determine:

- How many unique bugs the student found (discounting multiple runs that produced "the same" bug);
- How many of those bugs were deterministic versus concurrency bugs (did Landslide find the bug on the first interleaving or did it need to test multiple);
- How many of each category of bug the students fixed (determined when a subsequent run of Landslide on the test case failed to find the bug again).

**Results – bugs found.** Of 90 two-student groups in those 3 semesters, 47 tested their thread libraries with Landslide. Table 4.1 shows that Landslide found in total 44 deterministic bugs and 85 concurrency bugs. It found at least one concurrency bug for 32 groups (68%), 24 of whom (51%) were able to fix at least one bug, verifying their update with a succesful re-run of the same test. [8] I view this as a positive result – among other statistics, Landslide was able to help the majority of students debug, among those who found concurrency bugs at all – although it is difficult to control for the amount of time students spent with Landslide that would otherwise have been spent on old-fashioned stress testing.

| | Number of groups | | | |
|---|---|---|---|---|
| Number | Deterministic bugs | | Concurrency bugs | |
| of bugs | found | fixed | found | fixed |
| 0 | 27 | 32 | 15 | 23 |
| 1 | 6 | 5 | 7 | 11 |
| 2 | 9 | 6 | 12 | 8 |
| 3 | 2 | 1 | 7 | 2 |
| 4 | 1 | 1 | 3 | 1 |
| 5 | 2 | 2 | 2 | 1 |
| 11 | | | 1 | 1 |
| Total bugs | 44 | 34 | 85 | 53 |

Table 4.1: Summary of how many bugs were found and/or fixed by how many groups. Each column counts how many groups found/fixed the number of bugs in each row; for instance, two groups found 5 concurrency bugs, one of which fixed all 5.

**Results – impact on grades.** Next, I investigated the impact Landslide ultimately had on students' performance. I collected the ultimate P2 and P3 grades from the last 6 semesters, forming three categories: students from the past 3 semesters who volunteered to use Landslide, students from those semesters who opted out, and students from the other 3 semesters, for whom Landslide was not available at all. I hoped to measure a correlation between use of Landslide and ultimate P2 grades, which would show that it helps students submit more

[8] The table's data presentation is somewhat confusing: the dependent columns count the number of *groups*, categorized by how many bugs they found, not the raw number of *bugs*, instead indicated in the "Total bugs" row at the bottom. So, the number of total bugs is the sum of the products between each cell in that column and its corresponding number of bugs, e.g., $44 = 27 \times 0 + 6 \times 1 + 9 \times 2 + 2 \times 3 + 1 \times 4 + 2 \times 5$. I'll try to find a better way to present this data in the thesis.

correct P2s, and/or a correlation with ultimate P3 grades, which would (tentatively) show that it teaches better ways of thinking that students retain for future projects (i.e., measuring actual learning).

Unfortunately, this turned out to be a negative result, as shown in Figure 4.1. The distribution of Landslide users' ultimate grades is indistinguishable from that of students for whom Landslide was unavailable. While there was a 3% increase in P2 grades between opt-out-ers and Landslide users, it's prone to selection bias: perhaps those who opted out of Landslide were already the worst of the class who simply didn't have any free time for it. The distribution of P3 grades is indistinguishable as well. In the thesis, I will discuss possible factors contributing to this failure, and how they might be controlled for in future work.

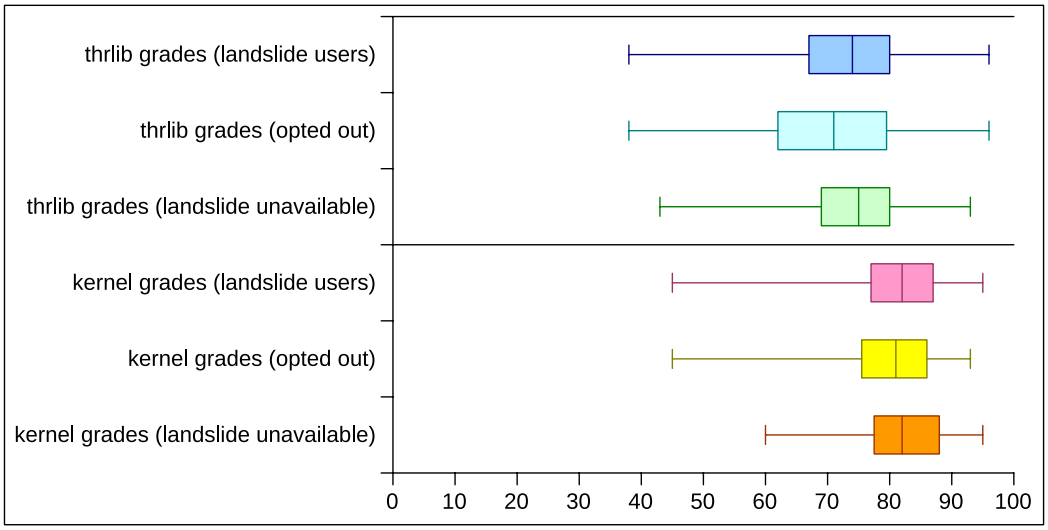



Figure 4.1: Distribution of P2 (thrlib) and P3 (kernel) grades among students who did or didn't use Landslide.

**Future work.** One evaluation question remains which I will answer in the thesis: what is the accuracy of Landslide's automation heuristics? In other words, were false-positive infinite loops or deadlocks ever reported where Landslide failed to recognize an ad-hoc synchronization? (I expect a near 100% success rate, having tuned the heuristics by hand on an (independent) set of P2 submissions already.) Finally, I will conclude this chapter in the thesis by discussing how future work could adapt Pebbles's interface to support fully-automatic model checking without compromising the educational value of P3.

# Chapter 5

# Pintos

The Pintos architecture [61], outlined in Section 2.4.2, provides an opportunity to test Landslide's pedagogical mettle beyond CMU's walls. Supporting Pintos as well as Pebbles will broaden Landslide's impact considerably, as Pintos is already used by many more universities than Pebbles.

The scheduler and concurrency primitives which Pintos's basecode provides limit the remaining concurrency-critical code left for students to write, which limits the degree to which Landslide can test student submissions. This is in contrast to the Pebbles assignments at CMU, in which students must implement all the functionality listed in Section 2.4.2. On the other hand, the upside of Pintos providing substantially more basecode than Pebbles is that most, if not all, of Landslide's kernel instrumentation can be done automatically, using the names of the core scheduler functions already provided. For the experiments in Chapter 3, I already implemented rudimentary support to test Pintos kernels, but more remains to be done.

## 5.1 Existing Progress

So far, just enough Pintos support exists to have run Chapter 3's experiments. This includes:

- a new set of annotations (e.g., `mutex_lock` in Pebbles is now called `sema_down`;
- handling several quirks of the basecode's scheduling behaviour;
- extending the heap tracker to handle Pintos's page allocator `palloc` as well as the kernel `malloc`, as well as the fact that kernel memory is not direct-mapped like in Pebbles;
- new thread-liveness code to detect when a test case terminates;
- skipping some busy-wait loops in the boot sequence used for device communication;
- an extra fall-back case in Landslide's timer injection algorithm;
- and extending the lock-set and vector-clock analyses to include disabling interrupts.

All these features apply to behaviours included with the stock base-code, so should apply generally to all but the most adventurous student implementations.

## 5.2 Proposed Work

There are still a few cases where the existing instrumentation is not fully general. When preparing Chapter 3's experiments, I manually adjusted some student submissions to be compatible with Landslide's existing annotation process. In order to distribute Landslide for student use beyond CMU, it will need to support these cases automatically. Examples of such ad-hoc fixes I would need to automate include:

- Allowing for the priority scheduler's runqueues to be implemented as an array of queues, rather than the single queue head as provided in the basecode.
- Insert the `tell_landslide_sleeping` annotation more intelligently than looking for a `list_insert_ordered` call (a basecode function which most, but not all, students use).
- Some versions of the basecode distribute a `sema_up` implementation which `yield`s unconditionally, which Landslide must bypass.
- Some students use `timer_sleep` or `while (!flag) continue` when they should use `yield`, which can lead to false-positive deadlocks if not automatically replaced.

When fixing my sample of Pintoses by hand, there were also myriad deterministic bugs, such as use-after-frees e.g. arising from unsafe `strlen` calls. As with Pebbles student experiments, however, I intend the students to encounter such bug reports and fix them on their own, despite not being concurrency bugs. Automatically ensuring stable determinized execution of arbitrary student code is beyond the scope of this thesis (if not outright impossible).

### 5.2.1 New emulation platform

Most practically, I will need to free Landslide from its dependence on Simics, which requires paid licenses for its use. Other candidate emulation (simulation or virtualization) platforms for Landslide include Bochs, QEMU, VMWare, and Xen. I have begun work porting Landslide to Bochs, which is open-source and provides both instruction and memory tracing, and which the Berkeley, Stanford, and U. of Chicago classes already use as their simulator.

### 5.2.2 Experimental goals

I aim to run a similar user study with Pathos students as I've done so far with Pebbles. After finding one or more Pintos-using classes at other schools to collaborate with, I will distribute Landslide to the students during the 'threads' and 'userprog' projects, have them run a similar suite of tests to those in Chapter 3's experiments, and collect and analyze the results. I plan to analyze the overall incidences and types of bugs found between Pebbles and Pintos populations, which will perhaps shed light on the advantages and/or shortcomings of either project, and lead to recommendations for improving either project's educational value (whether by incorporating MC or otherwise).

# Chapter 6

# Transactional Memory

In this chapter I propose to extend Landslide's concurrency model to support Transactional Memory (TM) [37]. This is the last of the four projects I am proposing for this thesis and so far exists only in dreams.

TM is a mechanism by which programmers may avoid conventional concurrency primitives, optimizing for performance in the common case when threads do not conflict. A transactional program surrounds its critical section(s) with transaction begin/end statements, which ensure that no other thread can observe an intermediate state during the transaction. If a conflict is observed, the transaction *aborts*, rolling the program back to the transaction's initial state, and executing an optional back-up code path. The programmer may also explicitly abort the transaction using an abort statement. An example transactional program is shown in Figure 6.1 (slightly extended from the prior example in Section 2.1.3).

TM may be implemented either in hardware (HTM) [36], or in software (STM) [3]. Though their interfaces to the programmer are similar, their semantics demand a slightly different treatment from Landslide's perspective. The key difference is that HTM transactions may fail for any reason, beyond the scope of the program's behaviour, such as the CPU's cache being too full. STM transactions, on the other hand, will fail only if an actual conflict is observed from another thread. Consider again the example program: The transactions of the two threads do not conflict, so they may abort only under HTM. However, when they abort for a reason other than a conflict on `foo` or `bar`, the assertions in the backup code will fail. Hence, some programs which are correct under STM may contain bugs under HTM.

Supporting TM in Landslide will consist of two major steps: extending the concurrency model to incorporate failure injections and extending DPOR to determine when transaction aborts are possible. Failure injections can model the semantics of transaction aborts, as each `_xbegin()` call can nondeterministically return either success (`_XBEGIN_STARTED`) or some abort failure code (such as `_XABORT_CONFLICT` or `_XABORT_CAPACITY`). Moreover, the extent of this nondeterminism depends on the underlying TM mechanism, which I will discuss shortly. (The `_XABORT_EXPLICIT` code is never generated nondeterministically but rather by a user invocation of `_xabort()` from within the success path.)

Initially `int foo, bar = 0; mutex_t m;`

**Thread 1**

```
if (_xbegin() ==
        _XBEGIN_STARTED) {
    foo++;
    _xend();
} else {
    mutex_lock(&m);
    assert(foo > 0 ||
           bar > 0);
    mutex_unlock(&m);
}
```

**Thread 2**

```
if (_xbegin() ==
        _XBEGIN_STARTED) {
    bar++;
    _xend();
} else {
    mutex_lock(&m);
    assert(foo > 0 ||
           bar > 0);
    mutex_unlock(&m);
}
```

Figure 6.1: Example transactional program, written using GCC's transactional memory compiler intrinsics [29]. These transactions would only abort if backed by HTM, while under STM their disjoint memory accesses guarantee them to succeed.

## 6.1 HTM

### 6.1.1 Mutex Isomorphism

When modelling TM in Landslide, we do not care about fidelity to performance characteristics or non-observable roll-back semantics. The goal of model checking is to exercise all observable program behaviours, so Landslide can model the execution of transactional programs using existing primitives if possible. In the first stage of the project, I will prove that a transactional program using is equivalent to one with a global mutex swapped for its `xbegin`s and `xend`s. Here I will assume a retry-only policy for handling aborts; in other words, transaction aborts could never direct the program to execute a different logic branch entirely. This will allow Landslide to test all observable TM behaviours using its existing infrastructure for mutexes, rather than relying on the platform providing accurate TM emulation.

### 6.1.2 Abort Nondeterminism

Next, I will extend the concurrency model to support the nondeterminism arising from transaction aborts. During execution, Landslide will inject a failure to force threads to branch into backup code paths. Failure injections add an extra "dimension" of non-determinism: at each `xbegin` operation which is a preemption point, Landslide may force a normal context switch to re-interleave threads, or it may inject a transaction abort to test the backup code. (This also avoids the need to speculatively execute and/or roll-back failing transactions. Being not testing the TM implementation itself, I'll assume it correctly rolls-back failed transactions non-observably.)

HTM transactions can fail for several reasons outside of a programmer's control, including cache evictions, false sharing (disjoint simultaneous accesses which nevertheless share

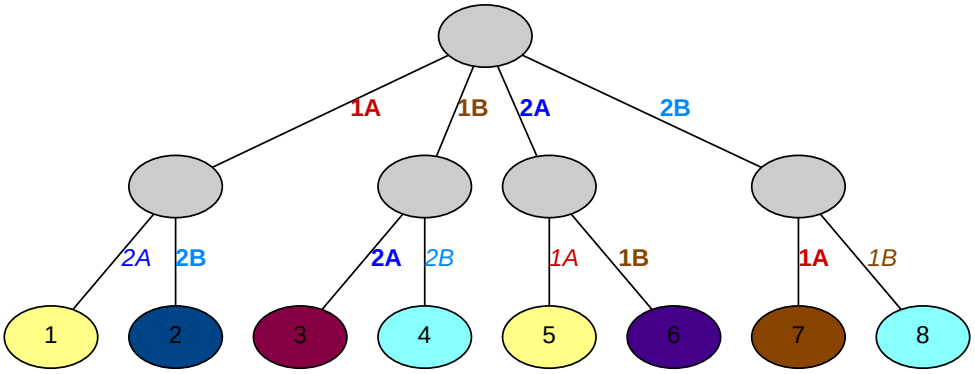



Figure 6.2: State space corresponding to the program in Figure 6.1. Conflicting transitions are marked in bold; transitions which are independent from their predecessors are italicized. End states colored the same (the cyan and yellow ones) are equivalent.

the same cache line), and page faults or interrupts [48]. For example, in Figure 6.1, the globals `foo` and `bar` are likely to share the same cache-line and lead to false sharing conflicts. Accordingly, Landslide's DPOR implementation will consider failure injections always "enabled" under HTM.

### 6.1.3 Reduction Challenge

I've identified at least one case under abort nondeterminism in which the current implementation of DPOR will fail to achieve a possible reduction. Let 1*A*, 1*B*, 2*A*, and 2*B* denote the transitions which execute the `if` and `else` branches by each thread in Figure 6.1 respectively. Note that 1*A* conflicts with 2*B* (thread 1 writes `foo`; thread 2 reads). and 1*B* with 2*A* (same for `bar`). Meanwhile, 1*A* and 2*A* are independent (disjint writes to `foo` or `bar` only), as are 1*B* and 2*B* (reads versus reads). Figure 6.2 provides a visual aid.

There are two possible reductions: pruning 2*A*, 1*A* after testing 1*A*, 2*A*, and pruning 2*B*, 1*B* after testing 1*B*, 2*B*; in other words, branches 5 and 8 can be skipped. However, our current DPOR implementation will tag the 2*B* subtree to explore after observing the conflict in branch 2 (likewise the 2*A* subtree from branch 3), but then have no memory of whether it was subsequently supposed to run 1*A* or 1*B*, and have to try both. To fix this will require an optimization analogous to "sleep sets" [28]: the tag to explore the 2*A* or 2*B* subtree must include an annotation to remember whether or not the conflict arose after a failure injection in 1.

## 6.2 STM

STM transactions abort only when multiple threads conflict directly on the same memory address(es) – the same sense of conflict that DPOR already considers. Because Landslide already computes memory conflicts among each pair of transitions, it will be natural to extend DPOR to consult the conflict set when deciding whether to exercise a failure injection. I will

extend Landslide in this manner, and even support programs with both types of transactions, as long as the different `xbegin` invocations are suitably annotated.

## 6.3 Hybrid HTM/STM

A recent paper [12] introduced several ways of combining HTM and STM in the same program, nesting transactions of different types. It presented several semantics for such transactions transactions, the most interesting of which being "open nesting", in which a nested transaction's state becomes visible to other threads even during a containing transaction. That state can then be rolled back if the latter aborts. I plan to develop a theoretical model for how such transactions would affect Landslide's concurrency model, and justify its correctness as an extension of the HTM or STM models alone.

## 6.4 Evaluation Plan

Because this will be the first model checker for testing transactional programs (to my knowledge; see Section 7.4), I don't expect any prior work will exist for a direct performance comparison. Nevertheless, I would pose two main questions to evaluate Landslide's proposed TM support:

- Does Landslide model TM concurrency accurately enough to find all the bugs and verify all the correct programs in a benchmark suite?
- To what degree can an improved DPOR reduce these programs' state spaces? Assuming I achieve the reduction challenge (Section 6.1.3), how much benefit is conferred, and how commonly across different types of programs?

Part of this work will be assembling a benchmark suite itself. I will investigate the benchmarks used in the Hybrid HTM/STM paper [12] (porting them to C if necessary), as well as more from various personal projects found online such as [67] and [18].

# Chapter 7

# Related Work

This chapter provides a brief tour of the related work.

## 7.1 History of Stateless Model Checking

**Hall of Fame.** Stateless model checking dates back to Verisoft [30], the 1997 tool which first attempted to exhaustively explore the possibile ways to interleave threads. Since then, researchers have built many tools along the same lines to test many kinds of programs. The most popular model checker, according to citation count, is Microsoft Research's CHESS [56], a checker for userspace C++ programs which preempts at synchronization APIs. Other checkers include MaceMC [43], MoDist [75], SAMC [47], ETA [69], each of which limit thread communication to the boundaries of a message-passing API; dBug [68], another CMU original, similar to CHESS; and finally SPIN [38] and Inspect [76] which can preempt at any shared variable access. Perhaps by now, Landslide itself deserves a spot on this list: I first introduced it in my MS thesis [7], and it is most notable for checking all shared memory accesses and supporting kernel-level code through the use of a simulator.

**Search Strategies** To date a number of techniques have been proposed to mitigate exponential explosion, the Sisyphean rock of stateless model checkers. Dynamic Partial Order Reduction (DPOR) [28], the baseline approach, prunes redundant interleavings by identifying independent thread transitions which can commute without changing the program state. It is a *sound* reduction algorithm, meaning it will never fail to test a possible program behavior, despite skipping many execution sequences. DPOR has since been extended in several ways: a provably optimal version which guarantees to explore no more than one interleaving from each equivalence class [1], and an extension of the algorithm to include nondeterminism arising from weak memory models [78]. Recently, SATCheck and Maximal Causality Reduction have emerged as better-performing alternatives to DPOR. These algorithms use SMT solvers [17] to identify additional equivalences by analyzing the values read and written by each memory access.

**Other theoretical advances.** A number of other techniques address the problem that even with DPOR, a state space may be too large. Iterative Context Bounding (ICB) [55] is a search ordering heuristic which prioritizes interleavings with fewer preemptions, according

to the insight that most bugs require fewer preemptions to expose. Bounded Partial Order Reduction [14] adapts DPOR to be soundly compatible with ICB. Preemption Sealing [6] allows programmers to exclude preemption points arising from trusted source code modules, reducing the state space by limiting the scope of the test. Probabilistic Concurrency Testing [11] eschews DPOR's depth-first approach entirely (as well as any potential for completing the state space), randomly sampling a broad cross-section of the state space and providing probabilistic bounds on uncovering bugs.

## 7.2 Concurrency in Education

I built Landslide upon the Pebbles [22] curriculum and concurrency model, naturally, as it is closest to home. Pintos [61] has recently emerged as the most popular educational kernel (by count of top CS schools in the USA who teach by it); it trades off the prevalence of its concurrency challenges to cover various OS topics more broadly, especially advanced scheduling algorithms. Pintos is the stand-alone evolution of its predecessor, Nachos [13], which originally ran as a UNIX process with simulated device drivers. Xv6 [15], 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 [5], which perhaps highlights the importance of related work on model-checking event-driven applications [40].

To my knowledge, this is the first study of model checking in an educational setting, although teaching concurrency is not itself an unstudied problem. Eytani et al. [26] present a promising framework for testing concurrent programs, which can incorporate model checking as well as static analysis, resource exhaustion, data-race analysis, and coverage analysis. However, it lacks an evaluation, and makes mention of its educational value only in its future work remarks. Lönnberg et al. [49] present a survey of how students think about concurrent program behaviour and debugging, while Kolikant [44] investigates how students form cognitive patterns about concurrent programming that could either aid or stunt their reasoning. Both of these studies could help optimize Landslide's bug reports for clarity and student enlightenment.

## 7.3 Other Concurrency Techniques

Data race analysis, originating with the lockset-only analysis of Eraser [64], has since grown into a mature field. Race detectors are largely distinguished by their particular flavour of the Happens-Before (HB) relation: some tools [27, 63] soundly avoid false positives using "Pure" HB in the Lamport sense [46]; others [58, 66] introduced the "Limited" HB relation to find more potential races in a single pass. I implemented both approaches in Landslide and evaluated their trade-offs in Section 3.4. Recently, the Causally-Precedes relation [72] emerged as a refinement of Limited HB which avoids the most common cases of false positives; it would be a welcome enhancement in Landslide.

Replay analysis extends single-pass data-race analysis to apply model checking (in limited quantities) to classify data-race candidates by their impact on program behaviour. It was first introduced by Narayanasamy et al. [57], which compares the program states immediately after the access pair for differences, preferring still to err on the side of false positives. RaceFuzzer [65] avoids false positives by requiring an actual failure be exhibited, as we do, although it uses random schedule fuzzing rather than model checking. Portend [42] 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. It uses symbolic execution to test input nondeterminism as well as schedule nondeterminism, although its verification properties are not as strong as Quicksand's.

## 7.4 Transactional Memory

Transactional memory (TM), first introduced in 1993 [37], has received renewed attention in recent years with the announcement of Intel's Haswell architecture [36], which supports hardware transactions using new x86 instructions. Because hardware transactions can fail for any reason, not just on memory conflicts, software TM remains relevant, and recent works [12, 34] enhance it to be nested with hardware transactions, or to be used on weak memory architectures, respectively; which in turn produce ever more complicated concurrency models for people like me to tackle. Testing approaches for transactional programs are sparsely represented in the literature so far. Several related works [21, 32, 33] are building up to formal proofs of the correctness of TM *implementations*, but not that of client programs which use TM. SI-TM [48] introduces techniques for reducing HTM's abort rates, but without fully eliminating aborts, MCs must still consider them possible anywhere. Getting closer, McRT STM [59] uses SPIN [38] to model check an STM implementation up to 2 threads running 1 transaction each with up to 3 memory accesses. This kind of verification is an important stepping stone for trusting the results Landslide will provide. *Learning from Mistakes* [51] studies the characteristics of many types of concurrency bugs; it found that TM could potentially fix some, but not all, of the studied bugs, although in some cases it must be combined with other concurrency primitives, which indicates a major need for verifying transactional code with model checking.

# Chapter 8

# Conclusion and Timeline

Stateless model checking is a powerful technique for testing concurrent programs, capable in theory of rooting out any nondeterministic bug or providing total verification on any program, but suffers several problems which relegate that theory to fantasy land [74]. Chief among those problems is exponential explosion of state spaces, making it difficult to decide in advance which combinations of preemption points can be productively tested within a given time limit. Another major problem is the manual annotation effort required to test certain types of concurrent programs, which is especially relevant for operating systems students who implement their own kernels. In this document I have proposed a thesis which will solve both problems. I leave you now with a proposed timeline for bringing each project to fruition.

- 2017 January: Submit thesis proposal, submit revised IRB proposal for planned user studies.
- 2017 ???uary: Present thesis proposal (whenever the committee is free).
- 2017 February-March: Deploy 15-410 student user study for more data (Section 4.3).
- 2017 March-August: Port Landslide to Bochs for use in Pintos OS classes (Section 5).
- 2017 March-August: Implement, evaluate, and write paper about transactional memory project (Section 6).
- 2017 August-October: Polish and deploy Pintos user study (Section 5).
- 2017 ???ember: Resubmit paper about 410 user studies (Section 4.3) (previously rejected from SIGCSE) (whenever an appropriate conference deadline arises).
- 2017 October-November: Deploy 410 student study again for more data.
- 2017 December: Begin writing thesis.
- 2018 January-April: Write thesis. Also fill in miscellaneous gaps, such as the "variance" Quicksand experiments (Section 3) and measuring Landslide's heuristics' accuracy (Section 4.3).
- 2018 February-March: Deploy Pintos student and 410 TA studies one final time.
- 2018 April-May: Defend & graduate.

# Bibliography

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 373–384, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535845. URL http://doi.acm.org/10.1145/2535838.2535845. 2.2, 7.1

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

[3] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 26–37, New York, NY, USA, 2006. ACM. ISBN 1-59593-320-4. doi: 10.1145/1133981.1133985. URL http://doi.acm.org/10.1145/1133981.1133985. 2.1.3, 6

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

[5] Jeremy Andrus and Jason Nieh. Teaching operating systems using Android. In *Proceedings of the 43rd ACM Technical Symposium on Computer Science Education*, SIGCSE '12, pages 613–618, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1098-7. doi: 10.1145/2157136.2157312. URL http://doi.acm.org/10.1145/2157136.2157312. 7.2

[6] Thomas Ball, Sebastian Burckhardt, Katherine E. Coons, Madanlal Musuvathi, and Shaz Qadeer. Preemption sealing for efficient concurrency testing. In *Proceedings of the 16th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 420–434, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-12001-6, 978-3-642-12001-5. doi: 10.1007/978-3-642-12002-2_35. URL http://dx.doi.org/10.1007/978-3-642-12002-2_35. 2.2, 7.1

[7] Ben Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, USA, May 2012. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-CS-12-118.pdf. 1, 4.1, 7.1

[8] Ben Blum. Soundness proofs for iterative deepening. Technical Report CMU-PDL-16-

103, Carnegie Mellon University, September 2016. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-16-103.pdf. 3.3.1, 3.3.2

[9] Ben Blum and Garth Gibson. Stateless model checking with data-race preemption points. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 477–493, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984036. URL http://doi.acm.org/10.1145/2983990.2984036. 3, 4.2

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

[11] 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. ISBN 978-1-60558-839-1. doi: 10.1145/1736020.1736040. URL http://doi.acm.org/10.1145/1736020.1736040. 7.1

[12] Keith Chapman, Antony L. Hosking, and J. Eliot B. Moss. Hybrid STM/HTM for nested transactions on OpenJDK. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 660–676, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4444-9. doi: 10.1145/2983990.2984029. URL http://doi.acm.org/10.1145/2983990.2984029. 6.3, 6.4, 7.4

[13] Wayne A. Christopher, Steven J. Procter, and Thomas E. Anderson. The Nachos instructional operating system. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 4–4, Berkeley, CA, USA, 1993. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1267303.1267307. 7.2

[14] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. Bounded partial-order reduction. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications*, OOPSLA '13, pages 833–848, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2374-1. doi: 10.1145/2509136.2509556. URL http://doi.acm.org/10.1145/2509136.2509556. 4.2, 7.1

[15] Russ Cox, Cliff Frey, Xiao Yu, Nickolai Zeldovich, and Austin Clements. Xv6, a simple Unix-like teaching operating system. http://pdos.csail.mit.edu/6.828/2012/xv6.html. 7.2

[16] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A practical runtime for deterministic, stable, and reliable threads. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, pages 388–405, New York, NY,

USA, 2013. ACM. ISBN 978-1-4503-2388-8. doi: 10.1145/2517349.2522735. URL http://doi.acm.org/10.1145/2517349.2522735. 3.1

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

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

[19] Brian Demsky and Patrick Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 20–36, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814297. URL http://doi.acm.org/10.1145/2814270.2814297. 2.2

[20] Dave Dice, Yossi Lev, Mark Moir, and Daniel Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 157–168, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-406-5. doi: 10.1145/1508244.1508263. URL http://doi.acm.org/10.1145/1508244.1508263. 2.1.3

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

[22] David Eckhardt. Pebbles kernel specification. http://www.cs.cmu.edu/~410-f16/p2/kspec.pdf, 2016. 1, 7.2

[23] David Eckhardt. Project 2: User level thread library. http://www.cs.cmu.edu/~410-f16/p2/thr_lib.pdf, 2016. 1, 2.1.2, 2.4.1, 4.3

[24] Laura Effinger-Dean, Brandon Lucia, Luis Ceze, Dan Grossman, and Hans-J. Boehm. IFRit: Interference-free regions for dynamic data-race detection. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '12, pages 467–484, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1561-6. doi: 10.1145/2384616.2384650. URL http://doi.acm.org/10.1145/2384616.2384650. 3.2

[25] Dawson Engler and Ken Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM. ISBN 1-58113-757-5. doi: 10.1145/945445.945468. URL http://doi.acm.org/10.1145/945445.945468. 2.2, 2.3

[26] Yaniv Eytani, Klaus Havelund, Scott D. Stoller, and Shmuel Ur. Towards a framework and a benchmark for testing tools for multi-threaded programs: Research articles. *Concurr. Comput. : Pract. Exper.*, 19(3):267–279, March 2007. ISSN 1532-0626. doi: 10.1002/cpe.v19:3. URL https://doi.org/10.1002/cpe.v19:3. 7.2

[27] Cormac Flanagan and Stephen N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542490. URL http://doi.acm.org/10.1145/1542476.1542490. 7.3

[28] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, pages 110–121, New York, NY, USA, 2005. ACM. ISBN 1-58113-830-X. doi: 10.1145/1040305.1040315. URL http://doi.acm.org/10.1145/1040305.1040315. 2.2, 6.1.3, 7.1

[29] The GNU Foundation. X86 transaction memory intrinsics. https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html, 2016. 2.1.3, 6.1

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

[31] Kurt Gödel. Über formal unentscheidbare sätze der principia mathematica und verwandter systeme, I. *Monatshefte für Mathematik und Physik*, 38(1):173–198, 1931. 2.2

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

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

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

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

[36] Per Hammarlund, Rajesh Kumar, Randy B Osborne, Ravi Rajwar, Ronak Singhal, Reynold D'Sa, Robert Chappell, Shiv Kaushik, Srinivas Chennupaty, Stephan Jourdan, et al. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, (2):6–20, 2014. 2.1.3, 6, 7.4

[37] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM. ISBN 0-8186-3810-9. doi: 10.1145/165123.165164. URL http://doi.acm.org/10.1145/165123.165164. 1, 2.1.3, 6, 7.4

[38] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997. ISSN 0098-5589. doi: 10.1109/32.588521. URL http://dx.doi.org/10.1109/32.588521. 2.2, 7.1, 7.4

[39] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 165–174, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737975. URL http://doi.acm.org/10.1145/2737924.2737975. 2.2

[40] Casper S. Jensen, Anders Møller, Veselin Raychev, Dimitar Dimitrov, and Martin Vechev. Stateless model checking of event-driven applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 57–73, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3689-5. doi: 10.1145/2814270.2814282. URL http://doi.acm.org/10.1145/2814270.2814282. 7.2

[41] Anthony Joseph. Cs162: Operating systems and systems programming. https://cs162.eecs.berkeley.edu/, 2016. 2.4.2

[42] Baris Kasikci, Cristian Zamfir, and George Candea. Data races vs. data race bugs: Telling the difference with portend. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 185–198, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2150997. URL http://doi.acm.org/10.1145/2150976.2150997. 7.3

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

[44] Yifat Ben-David Kolikant. Learning concurrency: Evolution of students' understanding of synchronization. *Int. J. Hum.-Comput. Stud.*, 60(2):243–268, February 2004. ISSN

1071-5819. doi: 10.1016/j.ijhcs.2003.10.005. URL http://dx.doi.org/10.1016/j.ijhcs.2003.10.005. 7.2

[45] 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. ISBN 0-934613-02-8, 978-0-934-61302-6. URL http://dl.acm.org/citation.cfm?id=1623611.1623684. 3.2

[46] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. ISSN 0001-0782. doi: 10.1145/359545.359563. URL http://doi.acm.org/10.1145/359545.359563. 2.3, 7.3

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

[48] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. Si-tm: Reducing transactional memory abort rates through snapshot isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 383–398, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2305-5. doi: 10.1145/2541940.2541952. URL http://doi.acm.org/10.1145/2541940.2541952. 6.1.2, 7.4

[49] Jan Lönnberg, Anders Berglund, and Lauri Malmi. How students develop concurrent programs. In *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*, ACE '09, pages 129–138, Darlinghurst, Australia, Australia, 2009. Australian Computer Society, Inc. ISBN 978-1-920682-76-7. URL http://dl.acm.org/citation.cfm?id=1862712.1862732. 7.2

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

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

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

[53] A Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets:*

*Applications and Relationships to Other Models of Concurrency*, pages 279–324, New York, NY, USA, 1987. Springer-Verlag New York, Inc. ISBN 0-387-17906-2. URL http://dl.acm.org/citation.cfm?id=25542.25553. 2.2

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

[55] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 446–455, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250785. URL http://doi.acm.org/10.1145/1250734.1250785. 2.2, 4.2, 7.1

[56] Madanlal Musuvathi, Shaz Qadeer, Thomas Ball, Gerard Basler, Piramanayagam Arumuga Nainar, and Iulian Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1855741.1855760. 4.2, 7.1

[57] Satish Narayanasamy, Zhenghao Wang, Jordan Tigani, Andrew Edwards, and Brad Calder. Automatically classifying benign and harmful data races using replay analysis. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 22–31, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-633-2. doi: 10.1145/1250734.1250738. URL http://doi.acm.org/10.1145/1250734.1250738. 7.3

[58] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '03, pages 167–178, New York, NY, USA, 2003. ACM. ISBN 1-58113-588-2. doi: 10.1145/781498.781528. URL http://doi.acm.org/10.1145/781498.781528. 2.3, 3.2, 7.3

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

[60] John Ousterhout. CS 140: Operating systems (winter 2016). http://web.stanford.edu/~ouster/cgi-bin/cs140-winter16/index.php, 2016. 2.4.2

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

10.1145/1508865.1509023. 1, 2.4.2, 5, 7.2

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

[63] 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. 7.3

[64] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, November 1997. ISSN 0734-2071. doi: 10.1145/265924.265927. URL http://doi.acm.org/10.1145/265924.265927. 2.3, 7.3

[65] Koushik Sen. Race directed random testing of concurrent programs. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 11–21, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-860-2. doi: 10.1145/1375581.1375584. URL http://doi.acm.org/10.1145/1375581.1375584. 7.3

[66] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-793-6. doi: 10.1145/1791194.1791203. URL http://doi.acm.org/10.1145/1791194.1791203. 1, 2.3, 3.2, 7.3

[67] Uri Shamay. Benchmark Intel TSX (transactional synchronization extension) hardware transactional memory on my sandbox. https://github.com/cmpxchg16/tsx.me, 2014. 6.4

[68] Jiri Simsa, Randy Bryant, and Garth Gibson. dBug: Systematic evaluation of distributed systems. In *Proceedings of the 5th International Conference on Systems Software Verification*, SSV'10, pages 3–3, Berkeley, CA, USA, 2010. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1929004.1929007. 3.1, 3.2, 7.1

[69] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Efficient Exploratory Testing of Concurrent Systems. Technical Report CMU-PDL-11-113, Carnegie Mellon University, November 2011. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-11-113.pdf. 7.1

[70] Jiri Simsa, Randy Bryant, and Garth Gibson. Runtime estimation and resource allocation for concurrency testing. Technical Report CMU-PDL-12-113, Carnegie Mellon University, December 2012. URL http://www.pdl.cmu.edu/PDL-FTP/Storage/CMU-PDL-12-113.pdf. 1, 3.2

[71] Jiri Simsa, Randy Bryant, Garth Gibson, and Jason Hickey. Concurrent systematic testing at scale. Technical Report CMU-PDL-12-101, Carnegie Mellon University, May 2012. URL http://www.pdl.cmu.edu/PDL-FTP/associated/CMU-PDL-12-101.pdf. 2.2

[72] Yannis Smaragdakis, Jacob Evans, Caitlin Sadowski, Jaeheon Yi, and Cormac Flanagan. Sound predictive race detection in polynomial time. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 387–400, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103702. URL http://doi.acm.org/10.1145/2103656.2103702. 2.3, 7.3

[73] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1), 1937. doi: 10.1112/plms/s2-42.1.230. URL http://plms.oxfordjournals.org/content/s2-42/1/230.short. 2.1.1, 4.2

[74] 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 $2^6$th Birthday*, SIGBOVIK '07, pages 87–88, Pittsburgh, PA, USA, 2007. ACH. URL http://sigbovik.org/2007/papers/proceedings.pdf. 3.1, 8

[75] Junfeng Yang, Tisheng Chen, Ming Wu, Zhilei Xu, Xuezheng Liu, Haoxiang Lin, Mao Yang, Fan Long, Lintao Zhang, and Lidong Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, pages 213–228, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1558992. 7.1

[76] Yu Yang, Xiaofang Chen, Ganesh Gopalakrishnan, and Robert M. Kirby. Efficient stateful dynamic partial order reduction. In *Proceedings of the 15th International Workshop on Model Checking Software*, SPIN '08, pages 288–305, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 978-3-540-85113-4. doi: 10.1007/978-3-540-85114-1_20. URL http://dx.doi.org/10.1007/978-3-540-85114-1_20. 7.1

[77] 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. 2.1.3

[78] Naling Zhang, Markus Kusano, and Chao Wang. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 250–259, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737956. URL http://doi.acm.org/10.1145/2737924.2737956. 2.2, 7.1