Transactional Memory Concurrency Verification with Landslide

Ben Blum

bblum@cs.cmu.edu

Abstract

Hardware transactional memory is a recently-introduced concurrent programming paradigm which allows programmers to elide locks for performance in low-contention workloads. However, it comes at a cost in implementation complexity: fast-path code must be accompanied by backup paths to handle transaction failure. We extend Landslide, a popular stateless model checker, with a concurrency model for transactional memory and evaluate it on several real-world transactional benchmarks and data structure implementations.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming; D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords landslide terminal, baggage claim, ground transportation, ticketing

1. Introduction

Transactional Synchronization Extensions (TSX) [23] is an instruction set extension for x86 CPUs which adds hardware-based transactional memory. The processor uses its existing cache coherence algorithm to check for memory conflicts with other cores while temporarily staging a sequence of memory accesses. If no other CPU accesses the same memory during the transaction, the access sequence is committed to main memory atomically (with respect to visibility by other CPUs). Otherwise, the accesses are discarded, the CPU's local state is reverted, and the transaction returns a failure code.

This feature can be used to replace conventional locking in performance-critical concurrent programs. When a concurrent workload accesses largely thread-local data, or disjoint sections of a shared data structure, the contention rate between threads is low, and transactions will often succeed. Compared to programs which use conventional locks, which use bus-locking atomic accesses even in the fastest code path, TSX provides substantial performance improvements in such programs [10, 12, 44]. However, the possibility of transaction failure introduces additional implementation complexity: programmers must also provide a backup plan to safely resolve contention between threads, usually involving conventional synchronization. These backup paths must coordinate not only with other backup paths but also with other fast paths which another thread may begin after the original transaction failed, which even in the simplest transactions requires complex synchronization sequences [10]. This introduces an additional dimension of nondeterminism into an already concurrent program, and moreover, because transaction failure is expected to be rare, obscure interleavings between failure paths are difficult to expose during stress testing.

This motivates the use of stateless model checking (MC) [19] to comprehensively verify these transactional programs, fast paths failure paths and all. MC aims to force the system to execute all possible thread interleavings under a given test case, exhaustively checking for bugs or verifying their absence in the corresponding state space. Many such model checkers exist, varying in interleaving granularity, memory analysis, types of programs checked, and search ordering strategy [9, 24, 25, 27, 28, 30, 32, 38, 43]. This work builds upon Landslide [4], a simulator-based tester which checks both user- and kernel-level programs and incorporates data-race analysis [16, 36] to find new preemption points at memory access granularity. Our contributions are as follows:

1. We extend Landslide's concurrency model to include transaction failure as an additional source of nondeterminism;
2. We provide a proof sketch that our implementation matches TSX's execution semantics,
3. We evaluate the extended Landslide on several transactional programs, analyzing both its bug-finding and verification performance.

The paper is organized as follows. Section 1 introduces the problem domain and motivates our research. The other sections state the rest of the paper.

SIGBOVIK '18 Pittsburgh, PA, USA

ACH . . . $15.00

2. Background

This section introduces the fundamental concepts and prior work in both hardware transactional memory and stateless model checking, which we propose to combine.

Hardware transactional memory. TSX was implemented on consumer hardware for the first time by Intel's Haswell architecture [22], which extends the x86 instruction set to provide `xbegin`, `xend`, and `xabort` for beginning, committing, and aborting transactions, respectively. Higher-level programming languages or compilers may offer libraries or intrinsics to access these instructions; for C and C++ GCC provides intrinsics named `_xbegin()` and so on [18]. Figure 1 shows an example program using TSX to synchronize access to a shared counter, including a failure path which defaults to a conventional lock. This example actually has a bug, which we will discuss in the next section; the reader is encouraged to try to spot it before then. Several related works formally prove the correctness of transactional memory *implementations* [13, 20, 21, 34], but verifying the client programs written to use transactions remains an open problem.

Under software transactional memory (STM) [2], memory conflicts with other threads are the only reason for transaction failure (apart from programmer-supplied explicit aborts); hence, depending on program semantics, some transactions may be guaranteed to succeed. However, hardware transactions (HTM) may also fail for several other reasons such as random system interrupts or exhausting the CPU's cache capacity. Because timer interrupts can in principle occur at any moment, and with arbitrary frequency (observable by the program, perhaps as a result of a heavily-loaded system), in this paper we will simplify the failure model by saying that HTM transactions can fail for any reason. We defer discussion of programs which distinguish the reason for aborts through the failure code to Section 5.

Stateless model checking. Model checking (MC) [19] is a testing technique for systematically executing and verifying the possible thread interleavings of a concurrent program. The main research challenge is to cope with exponential explosion of the state space, which is sized $O(n^k)$ for a program with n operations and k threads. Some *stateful* MCs explicitly store and compare all visited states of the program being tested [24], which both keeps track of test coverage and allows identifying identical states to avoid testing redundant interleavings. By contrast, *stateless* MC (henceforth abbreviated simply as MC) stores only the current sequence of execution events to avoid a prohibitive memory footprint. Reduction algorithms [1, 11, 17, 25, 25, 45] can then analyze the memory accesses in that sequence to identify interleavings observationally-equivalent under Mazurkiewicz trace theory [31] and hence safe to skip. The resulting state spaces are still exponentially-sized, but only in the number of conflicting operations rather than all operations. Of these, Landslide uses Dynamic Partial Order Reduction (DPOR) [17] to prune its state spaces.

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

Figure 1. Example transactional program. If the top branch aborts, execution will revert to the return of `_xbegin()` and control will drop into the `else` branch. The programmer can then use explicit synchronization to resolve the conflict.

MCs may instrument programs to introduce thread switches at varying granularity, which affects the number of operations n. Some target distributed systems, instrumenting only message-passing events [42]; some run multithreaded programs natively, instrumenting only the pthread API for performance [38]; and some insert compiler instrumentation on statically-identified memory accesses [32, 43]. Landslide traces every memory access through the use of a simulated environment [29], which is important for identifying data races to use as new preemption points [8], as well as for identifying when a memory conflict may cause transaction aborts. With regard to checking for bugs, the "model" the name refers to being checked may be an external formal specification, the program's own internal consistency checks, or a set of expected properties encoded in the tool itself. Landslide uses the latter two cases, checking for assertion failures as well as deadlocks, use-after-frees, and segfaults. For this work we also detect use of `xend` outside of a transaction or `xbegin` within one as a bug.

3. Design

This section presents our formalization of transactional memory in Landslide's framework of thread concurrency. We make two major simplifications: simulating transaction aborts as immediate failure injections, and treating transaction atomicity as a global mutex during data-race analysis; and provide corresponding equivalence proofs.

Notation. Let $I = TN_1@L_1, TN_2@L_2, ...TN_n@L_n$, with N_i a thread ID and L_i a code line number, denote the execution sequence of a program as it runs according to the specified thread interleaving.[1]

[1] This serialization of concurrent execution is told from the perspective of all CPUs at once and hence assumes sequential consistency. For discussion of relaxed memory models refer to Section 5.

3.1 Example

Consider again the program in Figure 1. Note that the C-style x++ operations, when compiled into assembly [41], results in multiple memory accesses which can be interleaved with other threads.

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

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

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

or, henceforth abbreviated for clarity:

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

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

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

However, the x++ operation from the failure path (correspondingly 6*a*, 6*b*, 6*c*) *can* be thusly separated with conflicting accesses interleaved in between, since the mutex only protects the failure path against other failure paths, but not against the transaction itself. So (assuming x is intended to be a precise counter rather than a sloppy one), we observe a bug in the following interleaving.[2]

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

Prior work [10] proposed the idiom shown in Figure 2 to exclude this family of interleavings, which shows that correctly synchronizing even the simplest transactions may be surprisingly difficult or complex.

3.2 Modeling Transaction Failure

Left unstated in interleavings such as $\mathbf{T1}@1-2c, \mathbf{T2}@1-3, \mathbf{T1}@4-7$[3] are HTM's execution semantics, namely:

1. any modifications to shared state (such as 2*c*) by **T1** are not visible to **T2** during its execution, despite **T2** being executed afterwards, and
2. all local and global state changes by **T1** between lines 1 and 2*c* are discarded when jumping to line 4.

While use of TSX in production requires the performance advantage of temporarily staging such accesses in local CPU cache, model checking such programs need be concerned only with the program's *observable* behaviours. We claim that MCing the simpler interleaving $\mathbf{T1}@1, \mathbf{T2}@1-3, \mathbf{T1}@4-7$ is an equivalent verification as MCing the one above; in fact, this interleaving suffices to check all observable behaviours of all interleavings of all subsets of $\mathbf{T2}@1-3$ with all subsets of $\mathbf{T1}@2a-2c$, whether they share a memory conflict or not. Stated formally:

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

[3] For a clearer example we reorder T1's write to x before T2's part here.

```
    prevent_transactions = false;

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

Figure 2. Variant of the program in Figure 1, with additional synchronization to protect the failure path from the transactional path. The optional line 0 serves to prevent a cascade of failure paths for the sake of performance by allowing threads to wait until transacting is safe again.

Lemma 1 (Equivalence of Aborts). *Let:*

- $\mathbf{Ti}@\alpha$ *be an HTM begin operation,*
- $\mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_n$ *be the transaction body (with* β_n *the HTM end call),*
- $\mathbf{Ti}@\phi_1 \ldots \mathbf{Ti}@\phi_m$ *be the failure path, and*
- $\mathbf{Ti}@\omega_1 \ldots \mathbf{Ti}@\omega_l$ *be the subsequent code executed unconditionally.*[4]

Then, for any interleaving prefix[5]

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

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

1. $\mathbf{Ti}@\alpha, \mathbf{Tj}@\gamma_1 \ldots \mathbf{Tj}@\gamma_j, \mathbf{Tk}@\kappa_1 \ldots \mathbf{Tk}@\kappa_k, \mathbf{Ti}@\phi_1 \ldots$ *(conflicting case), or*
2. $\mathbf{Ti}@\alpha, \mathbf{Ti}@\beta_1 \ldots \mathbf{Ti}@\beta_b \ldots \mathbf{Ti}@\beta_n, \mathbf{Tj}@\gamma_1 \ldots \mathbf{Tj}@\gamma_j, \mathbf{Tk}@\kappa_1 \ldots \mathbf{Tk}@\kappa_k$ *(independent case)*

exists and is observationally equivalent.

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

[5] Without loss of generality: for any number of other threads Tj/Tk, and for any number of thread switches away from Ti during the transaction.

Proof Sketch. We case on whether the operations by Tj and/or Tk have any memory conflicts (read/write or write/write) with $\mathrm{Ti}@\beta_1 \ldots \mathrm{Ti}@\beta_n$. If so, then the hardware will abort Ti's transaction, discarding the effects of $\mathrm{Ti}@\beta_1 \ldots \mathrm{Ti}@\beta_n$ and jumping to $\mathrm{Ti}@\phi_1$, satisfying case 1. Otherwise, by DPOR's definition of transition dependence [17], $\mathrm{Ti}@\beta_{b+1} \ldots \mathrm{Ti}@\beta_n$ is independent with the transitions of Tj and Tk, may be successfully executed until transaction commit, and reordering them produces an equivalent interleaving, satisfying case 2. □

The second part of our claim follows naturally.

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

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

3.3 Memory Access Analysis

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

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

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

Proof Sketch. We prove one case for each variant definiton for data races supported in Landslide [8]. For each, we semiformally state what it means to race in an execution with synchronizing HTM instructions.

- **Limited Happens-Before.** To race in I they must be reorderable at instruction granularity, at least one with a thread switch immediately before or after. [33, 36].
  - $I \Rightarrow I'$: If $\mathrm{Ti}@\mu, \mathrm{Tj}@\nu$ race in I, then they cannot both be in successful transactions, or else placing $\mathrm{Ti}@\mu$ within the boundaries of $\mathrm{Tj}@\nu$'s transaction would cause the latter to abort, invalidating $\mathrm{Tj}@\nu$, or vice versa. Hence they will not both hold m in I'. Otherwise their lock-sets and DPOR dependence relation remain unchanged.
  - $I' \Rightarrow I$: If $\mathrm{Ti}@\mu, \mathrm{Tj}@\nu$ race in I', both corresponding threads cannot hold m; WLOG let Ti not hold m during $\mathrm{Ti}@\mu$. Then in I, $\mathrm{Ti}@\mu$ is not in a transaction. With the remainder of their lock-sets still disjoint, and still not DPOR-dependent, $\mathrm{Tj}@\nu$ (or its containing transaction) can then be reordered directly before or after $\mathrm{Ti}@\mu$.
- **Pure Happens-Before.** WLOG fix $\mathrm{Ti}@\mu \prec \mathrm{Tj}@\nu \in I$. Then to race in I there must be no pair of synchronizing instructions $\mathrm{Ti}@\epsilon$ (a release edge) and $\mathrm{Tj}@\chi$ (an acquire edge) such that

$$\mathrm{Ti}@\mu \prec \mathrm{Ti}@\epsilon \prec \mathrm{Tj}@\chi \prec \mathrm{Tj}@\nu \in I$$

  to update the vector clock epoch between $\mathrm{Ti}@\mu$ and $\mathrm{Tj}@\nu$ [16, 35].
  - $I \Rightarrow I'$: If $\mathrm{Ti}@\mu, \mathrm{Tj}@\nu$ race in I, then they cannot both be in successful transactions, or else Lemma 1 normalization would provide the corresponding HTM end and begin for $\mathrm{Ti}@\epsilon$ and $\mathrm{Tj}@\chi$ respectively. Hence there will be no unlock/lock pair on m in I' to satisfy the above sequence.
  - $I' \Rightarrow I$: If $\mathrm{Ti}@\mu, \mathrm{Tj}@\nu$ race in I', then they cannot both hold m, or else $\mathsf{lockify}_m$ would provide the corresponding unlock and lock for $\mathrm{Ti}@\epsilon$ and $\mathrm{Tj}@\chi$ respectively. Hence there will be no HTM end/begin pair in I to satisfy the above sequence.

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

3.4 Implementation

Our implementation of HTM-equivalent semantics has been incorporated by the Landslide maintainers upstream. It is available open-source at `https://github.com/bblum/landslide`. Programs should be ported to the Pebbles userland [14, 15], their use of compiler HTM intrinsics should be replaced with the Landslide stubs provided in `410user/inc/htm.h`, and HTM nondeterminism can then be enabled with the `-X` command-line flag. We have also extended its Iterative Deepening implementation [8] with a new option (`-M` flag) to to optimize for completion time by prioritizing the maximal state space job and cancelling all others (which maintains its soundness guarantee) which we will use in our evaluation. All test cases therein are also available in the repository linked above.

buggy test	params	Quicksand mode cpu (s)	wall (s)	int's	Maximal state space mode (-M) cpu (s)	wall (s)	int's	SS size (est.)
htm1 (assertion)	2,1	45.78	9.70	21	*9.47	*6.40	21	*213*
	2,2	84.14	13.59	*33	*10.39	*7.70	49	*1536*
	2,3	131.91	20.44	*73	*12.83	*9.67	113	*10752*
	2,4	255.75	37.56	257	*18.63	*15.86	257	*73728*
	3,1	114.06	17.45	*15	*9.50	*6.79	21	*13653*
	3,2	109.60	26.16	49	*10.72	*7.97	49	*393216*
	3,3	124.80	20.40	*73	*13.84	*11.01	113	*11010048*
	3,4	227.49	35.15	*161	*31.37	*28.53	257	*301989888*
	4,1	53.08	9.79	*15	*9.82	*7.00	21	*873813*
	4,2	117.07	19.09	*33	*11.54	*8.55	49	*100663296*
swapbug (deadlock)	2,1	70.95	13.45	*16	*38.96	*13.15	109	*194*
	2,2	107.28	*17.45	*146	*44.73	19.47	281	*1620*
	2,3	280.05	38.70	*352	*60.30	*35.55	718	*12748*
	2,4	617.94	*81.50	*834	*108.58	82.60	1820	*97823*
	3,1	*1275.04	*163.42	*771	–	*>30m*	–	*184984*
	3,2	–	*>30m*	–	–	*>30m*	–	*3099225*
avl_insert (segfault+)	2,2	488.07	64.77	*83	*81.00	*40.30	336	*379982*
	2,3	2670.87	*330.45	*3066	*1331.79	1274.36	13926	*96248131*
	2,4	*3259.37	*436.50	*1639	–	*>30m*	–	*36019973*
	3,1	222.02	40.04	*28	*69.99	*24.25	78	*1572107*
	3,2	*1569.09	*216.85	*209	–	*>30m*	–	*1402363529*

Table 1. Landslide's bug-finding performance on various test configurations. Quicksand's workqueue approach optimized for fast bug-finding is compared against our maximum-state-space-prioritizing approach for fast verification. For each, we list the CPU-time and wall-clock time elapsed, plus the number of interleavings of the ultimately buggy state space tested, before the bug was found. * marks the winning measurements between each series. Lastly, Landslide's state space size estimation [39], though approximate at best, confers a sense of the exponential explosion.

4. Evaluation

To the best of our knowledge, this is the first work to test transactional programs in a model-checking environment, so no other MC State of the Art[6] exists to compare to in controlled experiments. Nevertheless, we pose the following evaluation questions.

1. How quickly does Landslide find bugs in incorrect transactional programs of varying sizes?
2. How quickly does Landslide verify correct transactional programs of varying sizes?
3. By the way, should MC research papers quantify variance in their CPU-time performance experiments?

Our evaluation suite comprises several hand-written unit tests and [10]'s microbenchmarks and transactional AVL tree and separate-chaining hashmap, as follows.

- `htm1`: The bug from Figure 1.
- `htm2`: The fixed version as in Figure 2.
- `counter`: Microbenchmark version of `htm2` which replaces the complex failure path with a simple `xadd`.
- `swap`: Microbenchmark that swaps values in an array.
- `swapbug`: `swap` modified to introduce circular locking in the failure path.
- `avl_insert`: AVL tree concurrent insertion test.
- `avl_fixed`: `avl_insert` with the AVL bug fixed (spoilers!!).
- `map_basic`: Separate-chaining hashmap concurrent insertion test.
- `map_basicer`: `map_basic` modified with a larger initial size to skip the resizing step.

The notation `testname`(K,N) will denote a test configuration of K threads, each running N iterations of the test logic. All tests were run on an 8-core 2.7GHz Core i7 with 32 GB RAM.

4.1 Bugs

Table 1 presents our bug-finding results. We configured Landslide to run the Quicksand algorithm [8] shown left, as well as to prioritize the maximal state space as discussed above, shown right, each with a time limit of 30 minutes. We draw three main conclusions from this data.

[6] The author's DJ name.

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

Figure 3. Unmodified code from `htmavl.cpp` showing the previously-unknown bug found by Landslide. The transaction path fails to check `_retry`, leading to data races and corruption just as in `htm1`.

Finding bugs quickly. As the test parameters increase, the multiplicative factor in bug-finding speed (2-4x, eyeballing) is generally smaller than that of the total number of interleavings (10-100x). In other words, should they exist, Landslide find bugs reasonably quickly in these transactional programs despite prohibitive exponential explosion in total state space size. This corroborates the prior work [8], extending its good news to the world of HTM.

New bugs. In addition to the bugs we intentionally wrote in `htm1` and `swapbug`, to our pleasant surprise Landslide also found a previously-unknown bug in the transactional AVL tree, exposed by `avl_insert` with any parameters higher than (2,1). Figure 3 shows the root cause, essentially the `htm1` bug in disguise. This manifested alternately as a segfault and as a consistency-check assertion failure. The presence of `while (_retry);` makes the necessary preemption window extremely small in practice (between it and `_xbegin()`), whereas MC is blind to such matters of chance.[7] Moreover, we conclude that even Figure 2's protocol's very proposer getting it wrong motivates the need for MC on such programs, and suggests TSX primitives should be encapsulated behind higher-level abstractions such as lock elision [26], which can be verified in isolation with smaller state spaces then trusted in turn when checking their client programs [37].

Performance comparison. Quicksand's ability to find bugs in fewer distinct interleavings does not necessarily correlate with better performance. Most of our tests are too small for its approach to pay off, with `swapbug`(3,1) and `avl_insert` as its notable wins. While the prior work showed plenty more wins [8], future MCs could prioritize state spaces using not just size estimation but state space maximality as well to soften the trade-off both for smaller tests and for verification. Speaking of which...

[7] Considering the loop does not affect the test's possible behaviours, only its *likely* ones, we removed it in Landslide's version of the test to keep the state space size manageable.

4.2 Verification

Landslide proved the following tests correct. With no bugs to find, comparing the different testing modes against each other is meaningless; we simply present their state space sizes and runtime (using `-M`) in Table 2.

test	params	cpu (s) *(or †ETA)*	SS size *(or †est.)*
`htm2`	2,1	18.57	294
	2,2	133.78	4902
	2,3	1986.98	79017
	3,1	11672.15	467730
	3,2	*†10d 14h*	*†13763999*
`counter`	2,1	5.57	30
	2,2	15.53	384
	2,3	155.00	5280
	2,4	2211.10	75264
	3,1	57.90	1960
	3,2	10028.93	329888
`swap`	2,1	32.98	193
	2,2	3652.91	101150
	3,1	*†8d 12h*	*†411312*
`avl_insert`	2,1	1083.35	40062
`avl_fixed`	2,1	1079.03	45078
	2,2	*†2762y*	*†8714863*
	3,1	*†12d 2h*	*†11498545*
`map_basic`	2,1	*†10d 17h*	*†16388977*
`map_basicer`	2,1	877.44	28635
	2,2	*†2d 7h*	*†5925634*
	3,1	*†468d 13h*	*†35893653*

Table 2. Transactional tests verified (or not) by Landslide.

Several larger tests we were unable to complete before the submission deadline are also shown. These are indicated with † and their listed ETAs and state space sizes represent Landslide's estimate after a timeout of 1 hour (hopefully enough for data-race PPs to saturate and estimates to stablize, although note the two estimate types use different algorithms so may disagree significantly). Though these tests proved beyond our reach, in contrast with the instant gratification expected of bug-finding, the verification guarantee may in some cases be worth the estimated time requirement for widely-used industrial implementations. Future work may also expand our coverage upon this frontier [7].

4.3 Variance

Because many of the verification tests are long-running, and we are writing this too close to the submission deadline (who doesn't), we regret being unable to present every performance measurement above as an average of multiple samples with error bars [40]. Nevertheless, we make some effort to address variance.

We noticed significant slowdowns when varying several aspects of our experimental environment. For example, multiple Landslides running at once slow each other down, likely arising from kernel resource contention as Landslide uses `fork()` to save simulation state. Table 3 shows the impact of running a single Landslide instance with `-M` on `counter`(2,2) with various programs also running, despite never saturating our test system's 8 CPUs.

Table 4 shows the impact of Landslide needing to automatically annotate `counter`(2,2) being run for the first time, rather than reusing existing instrumentation, as well as the variance of Quicksand mode. Because Quicksand repeats more work across jobs, rather than comparing it to a baseline, we show in the right column how many total interleavings each approach executed. (The maximal state space alone comprises 384 in all cases.) The quicksand variance was surprisingly bimodal, with 6 samples in a distribution of 106.14±1.22 and 4 in 142.11±1.67, suggesting two distinct scheduling patterns for its workqueue threads. Future work should figure out why.

We conclude that MC performance evaluations must address experimental environment variables to ensure consistent performance between runs. So doing, multiple samples and error bars are then necessary only when using nondeterministic search ordering strategies such as Iterative Deepening. Otherwise, considering the low variance shown in Table 3, they need be shown only for a token small test to provide reader some assurance of similar consistency in the larger tests (especially if the time tradeoff to measure them would sacrifice testing larger state spaces to begin with).

For all tests outside of Table 3, we fixed our environment by measuring all performance numbers with Firefox and Chrome as the only other significant machine load. We believe the exponential differences among completion times justifies the absence of error bars, which one would expect to show 2% variance if extrapolating from these results. The numbers of interleavings in each state space are, of course, deterministic and do not vary across runs.

load	cpu (s)	vs self avg	vs baseline
none	14.95 ± 0.17	0.99-1.02x	(baseline)
vid-L	15.13 ± 0.10	0.99-1.01x	1.01x
ff/c	15.55 ± 0.16	0.99-1.02x	1.04x
sm5	16.63 ± 0.07	0.99-1.01x	1.11x
ff/c+vid-S	17.09 ± 0.34	0.98-1.05x	1.14x
ls	19.11 ± 0.32	0.97-1.03x	1.28x
ff/c+ls	19.66 ± 0.50	0.96-1.02x	1.31x

Table 3. Performance variance on `counter`(2,2) with other programs running on the test machine. vid-L is full-screen video (played locally with `mplayer`), ff/c is Firefox and Chrome (idle, ≤20 tabs), sm5 is StepMania 5.1 (during gameplay [5]), vid-S is full-screen video (streamed via Crunchyroll), ls is a 2nd instance of Landslide. Average of 10 samples, ±*N* is 1 stddev.

Landslide mode	cpu (s)	total int's
verif (`-M`)	14.95 ± 00.17	403
reinstrument	24.72 ± 00.10	403
quicksand (no `-M`)	120.53 ± 18.62	2128

Table 4. Performance variance on `counter`(2,2) in various modes. Average of 10 samples, ±*N* is 1 stddev.

5. Limitations

This section should be mandatory in all systems papers.

Transaction failure codes. When a transaction fails, `_xbegin()` returns a failure code denoting the reason, or combination thereof, therefor [18]. If a program then cases on that failure code to select between different backup paths, model checking it by simply injecting a single type of failure may be unsound. For example, a program executing a transaction which is guaranteed to never conflict with any other threads, and hence never abort without `_XABORT_RETRY`, could legally `assert(false);` in its failure path, while our approach in Section 3 would erroneously trigger that assertion and report a bug. Likewise, the transactional data structures from [10] abstract away any spurious `_XABORT_RETRY` aborts behind a retry loop; a MC unwise to that idiom would call it an infinite loop bug.

Our HTM implementation includes an experimental feature to track the set of abort codes possible for each transaction. `_XABORT_RETRY` is always enabled; then, it harnesses DPOR's existing memory analysis to identify when `_XABORT_CONFLICT` is possible, and instruments `_xabort()` calls to record any user-supplied codes for `_XABORT_EXPLICIT`. This comes at a cost of even more state space explosion, increasing the exponent at each `_xbegin()` preemption point by (usually) 1 plus however many distinct `_xabort()` codes the program uses.

Many such branches may be equivalent; for example, explicit and conflict aborts need not be tested separately in transactions whose failure paths do not distinguish the cause, and the perennial `_XABORT_RETRY` abort can be skipped entirely if the client abstracts it away behind a retry loop. In fact, applying both reductions simultaneously amounts to the STM concurrency model [2][8] and may reduce the state space even smaller than the original. Testing them by hand [9] reduced `map_basicer`(2,1)'s 28635 interleavings to 11577, and produced the same 384-interleaving state space on `counter`(2,2) (in which DPOR triggers conflict aborts on every transaction any-

[8] Proof left to future work. We've proved enough here already.

[9] Using visual inspection of the test to trust the reductions' soundness.

way). Future work could use static or dynamic flow analysis to identify such reduction opportunities automatically; for now, this feature is disabled by default but accessible via the `-A` command-line option (in addition to `-X`).

Relaxed memory orderings. Section 3's formalization of thread interleavings does not account for read/write reorderings possible on relaxed consistency architectures [3]. In fact, even after [10]'s proposed fix, our running example program is still incorrect on Total Store Order (TSO) architectures such as x86. Despite stores being totally-ordered, x86 may still reorder stores after subsequent loads. Accordingly, an execution of $8, 9a, 9b, 9c$ in Figure 2 may be locally visible to another thread as $9a, 8, 9b, 9c$, and hence an apparent interleaving of

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

is possible (reordered accesses underlined for emphasis). An `mfence` barrier is needed between lines 8 and 9 to solve this problem on TSO [6]. On Partial Store Order (PSO) architectures, even more barriers may be necessary.

Because Landslide's concurrency model includes only instruction-level thread nondeterminism, not per-CPU memory buffer reorderings, our current HTM implementation cannot find this bug. In fact, it erroneously verifies the corresponding test `htm2(3,1)` in 3 CPU-hours, with 467730 distinct interleavings in total, none of which include the above-listed sequence. Recent work extended DPOR to support TSO and PSO memory nondeterminism [45]; if incorporated into Landslide, we could find or verify the absence of such bugs. Visual inspection of [10]'s HTM data structures found no barriers used in this implementation pattern; we would urge any reader interested in using those to add them in by hand first.

6. Conclusion

Stateless model checking research is a perpetual existence of staring up the sheer cliff face that is the exponential curve. As new concurrency paradigms emerge, we make our living by adapting our reduction algorithms and search strategies to them to climb ever higher on that curve, eking out a few more loop iterations or slightly higher thread counts in our verification guarantees. Whether that is beautiful in its imperfection or cause for despair is merely a matter of perspective. Also, the author hopes their committee won't mind the publication venue when they cite this in their thesis.

Acknowledgments

We thank Mario Dehesa-Azuara for generously providing the HTM data structures used in our evaluation, the anonymous SIGBOVIK program committee, and anyone who actually read this unusual paper all the way through. This work was supported in part by the U.S. Army Research Office under grant number W911NF0910273.

References

[1] P. Abdulla, S. Aronis, B. Jonsson, and K. Sagonas. Optimal dynamic partial order reduction. In *Principles of Programming Languages*, POPL '14. ACM, 2014.

[2] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman. Compiler and runtime support for efficient software transactional memory. In *Programming Language Design and Implementation*, PLDI '06. ACM, 2006.

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12), Dec. 1996.

[4] B. Blum. Landslide: Systematic dynamic race detection in kernel space. Master's thesis, Carnegie Mellon University, May 2012.

[5] B. Blum. A boring follow-up paper to "Which ITG stepcharts are turniest?" titled, "Which ITG stepcharts are crossoveriest and/or footswitchiest?". In *Conference in Celebration of Harry Q. Bovik's 2^6th Birthday*, SIGBOVIK '17. ACH, 2017.

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

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

[8] B. Blum and G. Gibson. Stateless model checking with data-race preemption points. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016. ACM, 2016.

[9] S. Burckhardt, P. Kothari, M. Musuvathi, and S. Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV. ACM, 2010.

[10] M. Dehesa-Azuara and N. Stanley. Hardware transactional memory with Intel's TSX. `http://www.contrib.andrew.cmu.edu/~mdehesaa/`, 2016.

[11] B. Demsky and P. Lam. SATCheck: SAT-directed stateless model checking for SC and TSO. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015. ACM, 2015.

[12] D. Dice, Y. Lev, M. Moir, and D. Nussbaum. Early experience with a commercial hardware transactional memory implementation. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV. ACM, 2009.

[13] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Electron. Notes Theor. Comput. Sci.*, 259, Dec. 2009.

[14] D. Eckhardt. Pebbles kernel specification. `http://www.cs.cmu.edu/~410-s18/p2/kspec.pdf`, 2018.

[15] D. Eckhardt. Project 2: User level thread library. `http://www.cs.cmu.edu/~410-s18/p2/thr_lib.pdf`, 2018.

[16] C. Flanagan and S. N. Freund. FastTrack: Efficient and precise dynamic race detection. In *Programming Language Design and Implementation*, PLDI '09. ACM, 2009.

[17] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Principles of Programming Languages*, POPL '05. ACM, 2005.

[18] GNU Foundation. X86 transaction memory intrinsics. `https://gcc.gnu.org/onlinedocs/gcc-4.8.2/gcc/X86-transactional-memory-intrinsics.html`, 2016.

[19] P. Godefroid. VeriSoft: A tool for the automatic analysis of concurrent reactive software. In *Computer Aided Verification*, CAV '97. Springer-Verlag, 1997.

[20] R. Guerraoui, T. A. Henzinger, and V. Singh. Completeness and nondeterminism in model checking transactional memories. In *Concurrency Theory*, CONCUR '08. Springer-Verlag, 2008.

[21] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Principles and Practice of Parallel Programming*, PPoPP '08. ACM, 2008.

[22] P. Hammarlund, R. Kumar, R. B. Osborne, R. Rajwar, R. Singhal, R. D'Sa, R. Chappell, S. Kaushik, S. Chennupaty, S. Jourdan, et al. Haswell: The fourth-generation Intel core processor. *IEEE Micro*, (2), 2014.

[23] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *International Symposium on Computer Architecture*, ISCA '93. ACM, 1993.

[24] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[25] J. Huang. Stateless model checking concurrent programs with maximal causality reduction. In *Programming Language Design and Implementation*, PLDI 2015. ACM, 2015.

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

[27] C. S. Jensen, A. Møller, V. Raychev, D. Dimitrov, and M. Vechev. Stateless model checking of event-driven applications. In *Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015. ACM, 2015.

[28] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with portend. In *Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII. ACM, 2012.

[29] K. P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux J.*, 1996(29es), Sept. 1996.

[30] T. Leesatapornwongsa, M. Hao, P. Joshi, J. F. Lukman, and H. S. Gunawi. SAMC: Semantic-aware model checking for fast discovery of deep bugs in cloud systems. In *Operating Systems Design and Implementation*, OSDI'14. USENIX Association, 2014.

[31] A. Mazurkiewicz. Trace theory. In *Advances in Petri Nets 1986, Part II on Petri Nets: Applications and Relationships to Other Models of Concurrency*. Springer-Verlag, 1987.

[32] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *Operating Systems Design and Implementation*, OSDI'08. USENIX Association, 2008.

[33] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Principles and Practice of Parallel Programming*, PPoPP '03. ACM, 2003.

[34] J. O'Leary, B. Saha, and M. R. Tuttle. Model checking transactional memory with Spin. In *Principles of Distributed Computing*, PODC '08. ACM, 2008.

[35] E. Pozniansky and A. Schuster. Efficient on-the-fly data race detection in multithreaded C++ programs. In *Principles and Practice of Parallel Programming*, PPoPP '03. ACM, 2003.

[36] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *Workshop on Binary Instrumentation and Applications*, WBIA '09. ACM, 2009.

[37] J. Simsa. *Systematic and Scalable Testing of Concurrent Programs*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2013.

[38] J. Simsa, R. Bryant, and G. Gibson. dBug: Systematic evaluation of distributed systems. In *Systems Software Verification*, SSV'10. USENIX Association, 2010.

[39] J. Simsa, R. Bryant, and G. Gibson. Runtime estimation and resource allocation for concurrency testing. Technical Report CMU-PDL-12-113, Carnegie Mellon University, December 2012.

[40] T. VII. What, if anything, is epsilon? In *Conference in Celebration of Harry Q. Bovik's 2^6th Birthday*, SIGBOVIK'14. ACH, 2014.

[41] T. VII. ZM$\sim\sim$ # PRinty# C with ABC! In *Conference in Celebration of Harry Q. Bovik's 2^6th Birthday*, SIGBOVIK'17. ACH, 2017.

[42] J. Yang, T. Chen, M. Wu, Z. Xu, X. Liu, H. Lin, M. Yang, F. Long, L. Zhang, and L. Zhou. MODIST: transparent model checking of unmodified distributed systems. In *Networked Systems Design and Implementation*, NSDI'09. USENIX Association, 2009.

[43] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby. Efficient stateful dynamic partial order reduction. In *Workshop on Model Checking Software*, SPIN '08. Springer-Verlag, 2008.

[44] R. M. Yoo, C. J. Hughes, K. Lai, and R. Rajwar. Performance evaluation of Intel(R) transactional synchronization extensions for high-performance computing. In *High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2013.

[45] N. Zhang, M. Kusano, and C. Wang. Dynamic partial order reduction for relaxed memory models. In *Programming Language Design and Implementation*, PLDI 2015. ACM, 2015.