Making Lock-free Data Structures Verifiable with Artificial Transactions

Xinhao Yuan David Williams-King Junfeng Yang Simha Sethumadhavan

Columbia University

{xinhaoyuan,dwk,junfeng,simha}@cs.columbia.edu

Abstract

Among all classes of parallel programming abstractions, lock-free data structures are considered one of the most scalable and efficient thanks to their fine-grained style of synchronization. However, they are also challenging for developers and tools to verify because of the huge number of possible interleavings that result from fine-grained synchronizations.

This paper addresses this fundamental problem between performance and verifiability of lock-free data structure implementations. We present TXIT, a system that greatly reduces the set of possible interleavings by inserting transactions into the implementation of a lock-free data structure. We leverage hardware transactional memory support from Intel Haswell processors to enforce these artificial transactions. Evaluation on six popular lock-free data structure libraries shows that TXIT makes it easy to verify lock-free data structures while incurring acceptable runtime overhead. Further analysis shows that two inefficiencies in Haswell are the largest contributors to this overhead.

Categories and Subject Descriptors

D.1.3 [**Concurrent Programming**]: Parallel programming; D.2.4 [**Software/Program Verification**]: Model checking; B.8.2 [**Performance Analysis and Design Aids**]

General Terms

Design, Verification, Performance, Measurement

Keywords

Lock-free data structures, software model checking, state space reduction, artificial transactions, transactional memory

1. Introduction

Parallel programs have become increasingly pervasive, driven by the rise of multicore hardware and the massive computations needed by the cloud and big data applications. A crucial building block for these programs is lock-free data structures, which export high-level, intuitive interfaces, such as a stack, queue, or hash table interface, and synchronize concurrent operations via only low-level primitives of shared memory accesses and atomic instructions. For instance, a lock-free stack's push operation may get the current stack top, append the new element, set the stack top to the new element using an atomic compare-and-swap instruction (CAS), and repeat if the CAS fails. (See §2 for a code example.)

Compared to typical lock-based code, lock-free data structures offer two key advantages. First, they often run faster and scale better with the number of cores, especially under high contention [19, 27]. This is because they synchronize in a more fine-grained manner, eliminating unnecessary waits and context switches when operations access different parts of shared memory. Second, they guarantee system-wide progress regardless of scheduling, by definition. This *lock-freedom* property soundly eliminates issues of deadlock, live lock, convoying, and priority inversion that plague lock-based code [20].

Because of these advantages, it is unsurprising that lock-free data structures are used in widespread applications such as MySQL[1] and at companies such as Facebook [2]. Almost every high-level programming language environment has a (semi-)standard lock-free library, including C++'s boost::lockfree [1], and portions of C#'s System.Collections.Concurrent namespace [6] and Java's java.util.concurrent package [3].

Despite their importance, lock-free data structures remain extremely hard to get right, even for experts, as evidenced by subtle bugs in a substantial amount of lock-free code published at good magazines and referred journals [5]. A key reason is that concurrent executions of lock-free data structures produce a vast set of possible shared memory access interleavings, or *schedules*; the number of schedules grows exponentially with the number of shared memory accesses performed. Each schedule may lead to a different, sometimes correct and sometimes buggy, result, so all schedules must be validated for correctness, a daunting task for developers.

While much effort [17, 23, 28, 29, 33–37] is dedicated to building effective tools to check parallel programs, few handle lock-free data structures. This is because of fine-grained nature of lock-free algorithms, which produce an extremely large state space even after advanced state space reduction techniques [15, 16, 18, 30].

This paper presents TXIT, a system that simplifies the verification[2] of lock-free data structures. TXIT dramatically reduces the set of schedules by instrumenting the code of a lock-free data structure to group instructions into *artificial transactions*, each of which is guaranteed to run atomically. These transactions are added post facto after developers have written the code, hence we call them

PLOS '15, October 04–07, 2015, Monterey, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3942-1/15/10...$15.00.
http://dx.doi.org/10.1145/2818302.2818309

[1] `mysys/lf_{dynarray,alloc-pin,hash}.c` in MySQL 5.6.20.

[2] Instead of verifying data structures by itself, TXIT reduces the number of possible schedules to a manageable size that one can exhaustively check/verify with other tools. By combining TXIT with our modified dBug model checker, we showed that it is possible to verify all schedules for a data structure with respect to one test-case program.

artificial. A tool now needs to verify only the schedules of artificial transactions, an exponential reduction from the set of all shared memory access schedules. Once the data structure is deployed in production, TXIT continues to enforce these transactions for correctness, and leverages hardware support to reduce the overhead of transactions. TXIT thus automatically offers high assurance for legacy and new applications that use lock-free data structures, while retaining performance better than typical lock-based code.

Adding artificial transactions on execution can be quite dangerous, as in arbitrary code it may introduce deadlocks or live locks, demonstrated in prior study [11]. Fortunately, TXIT does not suffer from this problem thanks to the obstruction-freedom of lock-free algorithms, where a thread can always make progress with arbitrary transactions enforced.

A key challenge TXIT faces is the tradeoff of performance vs verifiability (i.e., the number of schedules to verify). The granularity of artificial transactions determines the performance and verifiability of a lock-free data structure. Small transactions may not reduce the number of schedules adequately, while larger transactions yield fewer schedules, making the data structure much easier to verify, but increasing increase the probability of transaction conflicts, causing higher overhead for handling transaction aborts and retries. Thus, it is crucial for TXIT to select a good plan to place transactions such that (1) all schedules of the data structure can be verified given a testing time budget and (2) the data structure with the inserted artificial transactions gives close to maximum performance under this testing budget. To tackle the challenge, we designed a heuristic search engine for empirically finding a high-performing transaction placement plan given a testing budget (§2).

We implemented TXIT for C/C++ lock-free data structures. It leverages the LLVM compiler [24] to instrument programs and insert artificial transactions, the Pyevolve genetic programming engine [8] to search for an optimal transaction placement plan, the dBug model checker [31] to systematically check schedules of transactions, and TSX – the hardware transactional memory support readily available in the 4th generation Intel Core processors (codenamed "Haswell") [10] –to enforce artificial transactions (§3).

Evaluation on six popular lock-free data structures (§4) shows that:

1. TXIT computes transaction placement plans such that the resultant data structures on the given test cases can be verified within several minutes by dBug.
2. The normalized execution time of TXIT ranges from 1.55–4.30× using Haswell TSX.
3. According to our micro-benchmarking results, the overhead is primarily due to performance pathologies in Haswell TSX. For instance, transactional reads are (1) up to 1.63× slower than non-transactional ones and (2) are almost always *slower* than transactional writes.

Contributions. To the best of our knowledge, TXIT is the first system that leverages transactional memory to aid verification of lock-free data structures. Our additional contributions include the idea of artificial transactions, the heuristic search engine for placing transactions, the results of verifying several popular lock-free data structures, and the discovery of the performance pathologies in the Haswell TSX support, along with our suggestions for improvements which we believe will benefit others wanting to use this feature.

2. Overview

In this section, we show a lock-free data structure example to illustrate the difficulty of writing and verifying such data structures; we show how TXIT makes it easy to verify the example; and we describe the recommended usage of TXIT.

```
        void push(stack *s, element *e) {
          element *top;
          do {
push.1:     top = s->top;
push.2:     e->last = top;
push.3:   } while (CAS(&s->top, top, e) != top);
        }

        element *pop(stack *s) {
          element *top, *last;
          do {
pop.1:      top = s->top;
pop.2:      last = top->last;
pop.3:    } while (CAS(&s->top, top, last) != top);
          return top;
        }
```

(a)

`stack *s` is initialized as A→B→C→D

thread 1	thread 2
`element *x, *y;`	`element *x, *y;`
`t1.1: x = pop(s);`	`t2.1: x = pop(s);`
`t1.2: y = pop(s);`	`t2.2: y = pop(s);`
`t1.3: free(y)`	`t2.3: push(s, x);`
	`t2.4: free(y)`

(b)

Figure 1. A lock-free stack (a) and a failure-causing test case (b).

2.1 An Example

Figure 1 shows a lock-free stack example and a test case exercising the stack. The `push` and `pop` operations appear correctly implemented because they use CAS to detect that the stack top is changed and retry accordingly. However, the code actually suffers from a subtle bug that causes the same element to be popped twice. Consider this scenario: after thread 1 gets the stack top and sets `last` to point to element B, thread 2 pops two elements and the pushes back element A. Now, when thread 1 runs the CAS instruction to detect conflicts, the stack top is still A, so the CAS succeeds but incorrectly sets the stack top to point to B. When thread 1 continues to pop the next element, it gets B, causing a double free. This bug is the classic ABA bug [9, 26], which is common in lock-free data structure implementations.

Finding this bug is hard because even the simple test case shown in figure 1.(b) has an enormous number of schedules, estimated by dBug to be 9×10^{22}. After utilizing state-of-the-art state space reduction technique, dynamic partial order reduction (DPOR) [15], the number of schedules is still estimated to be 2×10^{7}.

2.2 TXIT Work Flow

We describe how to make the stack example easy to verify with TXIT. To reduce the set of schedules, TXIT inserts artificial transactions. It starts by transforming each operation of the stack into a transaction, maximizing verifiability. Concretely, TXIT makes `push` and `pop` transactions, reducing the number of schedules down to only 10, eliminating the ABA bug in `pop`.

This baseline transaction placement plan may incur high overhead, so TXIT performs a search to find a good transaction placement plan. It guides the search using an evaluation function that (1) quantifies performance by measuring the execution time of the test case and (2) ensures that the estimated number of schedules is smaller than the testing budget ("estimated" because counting the precise number requires fully exploring the schedules).

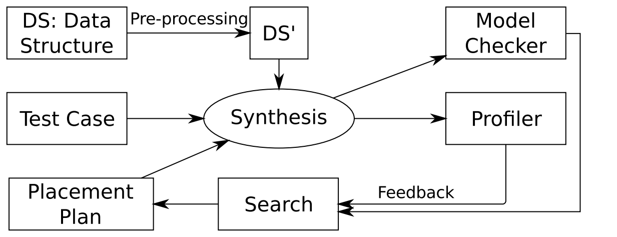

Figure 2. The architecture of TXIT. Placement plans are synthesized into a proposed program, which is profiled and checked, and which feeds into new placement plans.

After TXIT finds an optimal plan with good performance and a verifiable set of schedules, it outputs a new stack implementation with transactions inserted. Developers then run their favorite tools to verify the correctness of this implementation, and deploy the implementation in production environments, where transactions are running under hardware support with minimal overhead.

2.3 Recommended Usage

While in principle developers can use TXIT in any development stage, we recommend a specific stage—after traditional testing, but before deployment—because we believe TXIT is the most useful in this stage. During active development, the code frequently changes, and for each version of the code, TXIT may produce a different transaction placement plan, so its usefulness is limited. TXIT provides high assurance by reducing the set of schedules, and the removed schedules may effectively hide bugs. Thus, developers should do testing/verification as usual without TXIT to find as many bugs as possible, and turn on TXIT as a final step to get high assurance in production environments.

3. Implementation

In this section, we show how we utilize the architecture support and model checker to build TXIT, shrinking the schedule space of lock-free algorithms using artificial transactions. We will first describe an overall architecture, then explain each component in detail.

3.1 Architecture

TXIT takes a lock-free data structure and a comprehensive[3] test case for input. The lock-free data structure is given as a library with exported interface functions. The whole workflow is shown in Figure 2.

Taking the data structure and test case as input, TXIT gradually improves the current placement plan by synthesizing checkable and runnable programs, feeding them into the model checker and profiler, and using their reports as feedback. After a number of iterations, the system outputs the best placement plan it has found.

Note that the system does not verify the source program. It finds a transaction placement plan to reduce the set of schedules needed to reason about for further verification. The goal is to get the best performance with a set of schedules that is still verifiable within the given budget.

3.2 Pre-processing

Given the data structure as a code library, it needs to be pre-processed before synthesizing and profiling. All the pre-processing procedures are done using LLVM IR transformations.

We first try to *flatten* the library so that most function calls are inlined, which expands the control flow and data flow, so they become cleaner and easy to deal with statically. Due to theoretical and resources constraints on the compiler, not all function calls can be expanded; in which case we leave them untouched. Note that this may incur more pressure on the L1 code cache. Since the lock-free data structures are relatively small, in our experiments we did not observe any slowdown due to the extra pressure.

After the transformation, we identify all memory accesses in the library, and intercept them by appending hook functions. This is much more heavyweight than the original memory accesses. We only enable this instrumentation during model checking.

[3] We rely on experts with domain knowledge to provide test cases with good coverage for testing desired properties.

3.3 Model Checker Enhancement

To perform model checking on the program under a given transaction placement, we leverage the model checker dBug. dBug intercepts synchronization functions (such as `pthread_mutex_lock`) to track and control the scheduling. But it is not aware of any instruction level memory accesses, nor transactions. We modified dBug to support the semantics of transactions, by extending it with two synchronization primitives: `TXBegin()` and `TXEnd(read_set, write_set)`. `TXBegin` starts a transaction and `TXEnd` commits the transaction with read/write sets collected by instrumentation. dBug simulates the schedule by enforcing the total order of all synchronizations. During the checking `TXBegin` suspends all other threads so that only one transaction can be active for a given time.

3.4 Intel Haswell TSX Runtime

To utilize the hardware TSX support, we wrapped the instruction level interface into routines, `tx_begin` and `tx_end`, to start and commit a transaction. The tricky detail is how TSX deals with conflicts. TSX detects conflicts by monitoring local cache lines involved in the ongoing transactions. Once such a cache line is invalidated or degraded (e.g. from "exclusive" state to "shared" state), TSX will discover the conflict against concurrent transactions in other threads, and abort the local transaction to resolve the conflict. This makes the local transaction exit transactional mode, and jump to a fallback branch prepared before the transaction, which does not guarantee the progress of transactions. One could let failed transactions retry until success, which could simply lead to a live lock. The optimization guidelines of TSX [22] require the programs to always provide a non-transactional fallback path for each transaction and must not simply let the transaction retry. Since the lock-free code is not aware of transaction enforcement, we need to provide our own fallback path. We used an exponential back-off strategy to resolve the contention in a decentralized way, which we believe is more scalable than global critical sections as the number of conflicting objects increases.

There are some situations where transaction aborts are not because of conflicts. For example, accessing an unmapped page will cause a page fault and cause an abort unless in a non-transactional fallback. TSX provides a value in `EAX` to identify the reason of an abort. TXIT runtime will detect such situations and fallback to a global critical section.

TXIT also leverages dBug to evaluate the verifiability of a placement plan. This is done by extracting the schedule space estimation from dBug. dBug computes this by dividing total number of states explored by the sum of probability of each explored state.

3.5 Genetic Search

TXIT performs a genetic search on transaction placement plans, which are represented as boolean vectors. Each element in a plan vector denotes whether or not to insert a transaction bounary at a given location. We only consider boundaries before and after memory access instructions, so the size of a plan vector is fixed to twice the number of memory access instructions in the given program. The genetic search maintains a population of placement

Library	Data Structures Selected
boost::lockfree [1]	stack (BLFS), queue (BLFQ)
folly (Facebook) [2]	producer-consumer-queue (FPCQ)
liblfds [4]	stack (LFDSS), queue (LFDSQ)
nbds [7]	skiplist (NBDSSL)

Table 1. Evaluated lock-free data structures.

plans, and updates the population by randomly picking existing plans for crossover and mutation. Initially, the population is generated with random boolean vectors with the fixed size.

4. Evaluation

Our evaluation focuses on three research questions:

- Can artificial transactions reduce the number of schedules effectively?
- Can TXIT find transaction-placement plans that offer good verifiability and performance?
- What is the overhead of TXIT with current hardware transactional memory? Where does the overhead come from?

4.1 Evaluation Setup

Our evaluation is performed on a work station with 16 GB of memory and Intel(R) Core(TM) i7-4770. This CPU has four cores and up to two hyper-threads per core, but we disabled hyper-threading per recommendation of the Intel manual. We locked the CPU frequency to 3 GHz to avoid inaccurate measuring caused by frequency scaling. The workstation runs Debian with Linux 3.11.

We selected 6 popular open source implementations of lock-free data structures, shown in Table 1.

Folly from Facebook contains two data structures claimed to be lock-free, including FPCQ and atomic hash array. However, TXIT detects a deadlock after adding transactions to the atomic hash array. It turns out this data structure is actually not lock-free: it spin-waits on hash array slots, violating lock-freedom. We thus leave this data structure out in our evaluation.

We used the following test cases to exercise the data structures. For general stacks and queues, the test cases spawn three threads, where each thread pushes two elements onto the stack/queue and then pops them out. For single consumer/producer queue (FPCQ), the test case spawns a producer thread and a consumer thread, where each thread pushes/pops the queue six times. For skip-list, the test case spawns three threads, where thread $i \in \{0, 1, 2\}$ inserts an element with keys $\{i, i+3, i+6\}$ into the skip-list, making the threads fully interleave in the insertion process.

4.2 Reduction on the Number of Schedules

Here we evaluate how artificial transactions reduce the number of schedules with unified transactions size. Specifically, we show how the number of schedules grows (or shrinks) as the transaction size varies. For each test, we group every n shared memory accesses into a transaction and use dBug to determine the number of schedules. Here $s_{\text{est}}(1)$ indicates that each instruction is in its own transaction, and $s_{\text{est}}(\infty)$ indicates that all instructions within an operation of the lock-free data structure are in one transaction. We run dBug for up to 10^4 iterations to estimate the number of schedules; any S_{est} smaller than 10^4 is a exact result. Figure 3 shows the result with y-axis in log scale, demonstrating huge reduction in the number of schedules as the transaction size grows.

4.3 Performance and Verifiability Tradeoff Results

In this section, we evaluate how well TXIT makes the performance and verifiability tradeoffs. Given different testing budgets s_{budget}

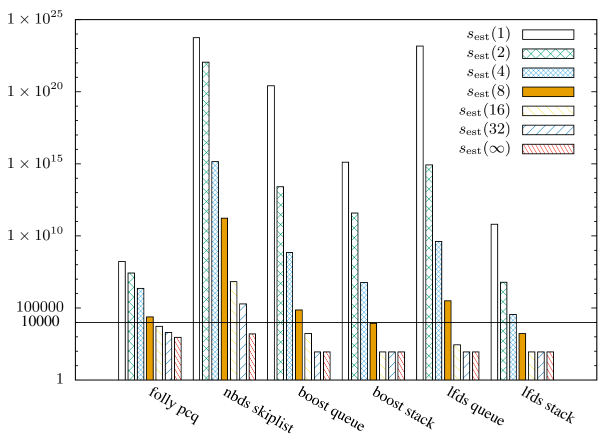

Figure 3. Number of schedules (in log scale) vs transaction size. The horizontal line is at 10^4, and any result below this line is exact.

Budget	Base-line	2×10^3	2×10^4	2×10^5	2×10^6
FPCQ	2.458	N/A	2.363	2.601	1.553
NBDSSL	2.166	2.142	1.952	2.355	1.935
BLFQ	3.649	3.628	3.232	3.321	3.063
BLFS	3.747	3.521	3.133	3.308	3.126
LFDSQ	3.184	4.304	2.705	2.569	2.565
LFDSS	2.481	2.776	1.956	2.916	2.047

Table 2. Normalized execution time of the test cases with transactions over without (smaller is better). The baseline column shows the normalized execution time at the starting point of the search for each data structure when every operation is made a transaction.

expressed as the number of schedules that developers can afford to test, TXIT's heuristic search engine explores the possible transaction placement plans, evaluates the plans according to the resulting state space and performance reports, and evolves them using genetic algorithms. In a few cases, increasing the testing budget did not improve performance because a solution for a smaller budget is faster.

To better understand these cases, we adjusted the evaluation criteria slightly to direct the search toward a solution whose number of schedules is close to the budget. We used the following genetic search parameters: population size of 70 and 80 generations, resulting in 5600 iterations for each data structure and testing budget.

Table 2 shows the results. Each cell shows the normalized overhead of running a test case with transactions over without. The baseline column shows the normalized overhead for the starting point of the search, i.e., when each operation of the data structure becomes one transaction and the verifiability is maximized. TXIT did not find a valid placement plan for FPCQ when $s_{\text{budget}} = 2 \times 10^3$ because that baseline solution already has more schedules than the budget.

Running the parallel test cases on these placement plans shows that our searching system is able to reduce the running time of the baseline enforcement, to between 89.3% and 63.2%. Compared to the original performance without transactions, the numbers show the factors from 155.3% to 430.4% across all placement plans from search.

To understand what costs contributed to the overhead, we rerun these plans but using test cases that spawn only one thread; results

Budget	Base-line	2×10^3	2×10^4	2×10^5	2×10^6
FPCQ	1.273	N/A	1.539	1.530	1.863
NBDSSL	1.155	1.159	1.193	1.363	1.222
BLFQ	1.291	1.369	1.326	1.409	1.443
BLFS	1.299	1.399	1.404	1.476	1.378
LFDSQ	1.080	1.380	1.491	1.440	1.584
LFDSS	1.090	1.175	1.253	1.405	1.392

Table 3. Normalized execution time of single-threaded test cases with transactions over without (smaller is better).

```
  1 MOVL %eax, 0x000(%rdi)  ---- start of the first pass
    ...
256 MOVL %eax, 0x1F8(%rdi)  ---- end of the first pass
257 MOVL %eax, 0x000(%rdi)  ---- repeat (second pass)
    ...
```

Figure 4. Microbenchmark for evaluating TSX overhead.

from these experiments measure the operational cost of the transactions without any conflicts.

Table 3 shows the results. Operational cost is quite high, ranging from 17.5% to 53.9%. Even if we insert only one transaction for each operations, as in the baseline, the operational cost is still observable (8% to 29.9%). The implication is that the operational cost of Haswell transactional memory is relatively large compared to the size of the transactions we insert. We examine this cost in greater detail below.

4.4 Understanding Haswell TSX overhead

We have seen that transactions in the current Haswell implementation carry significant overhead even if there is no conflict. Since the operations of the evaluated lock-free data structures take hundreds to thousands of cycles, we wrote a microbenchmark at the same scale to study the behavior of Haswell transactions.

Microbenchmark. We designed the microbenchmark to be memory intensive to resemble the behavior of lock-free data structures. The benchmark consists of auto-generated functions that access a given work space of memory that fits in L1 data cache (2 KB in our experiments). We first generate a instruction sequence that accesses the work space with a given stride, then we generate the benchmark function by repeating the sequence up to a given total instruction length. The structure of the microbenchmark is shown in Figure 4.

Cache Settings To compare the performance under different cache settings, we prepare a separate memory space to fill out the L1 cache, so that we can observe the overhead comparison of cold and warm cache. We place data in L1 for the warm case and in L2 for the cold case (to avoid L3 or main memory delay). We also run tests where the relevant L1 cache lines are all initially in a modified state, and where they are in an exclusive state. There is a significant flushing cost (compared to cache hit without eviction) when evicting modified cache lines, since TSX needs to flush dirty cache lines even for cache hits. Taking all combinations into account, we have four cases: WarmModified (WM), WarmExclusive (WE), ColdModified (CM) and ColdExclusive (CE).

We measured the TSX characteristics in each case. According to the structure of the microbenchmark, there ought to be two phases: the first phase touches all cache line in the read/write set, incurring extra cost; after that all data are warmed in the cache, ameliorating the performance. In the experiments, the first phase lasts from line `1` to `256`. We calculate the per-instruction overhead for the first phase. For comparison, we conducted the same experiments under non-transaction mode. Table 4 shows the results. Transactional load and store instructions are quite costly in TSX in almost all cases:

Cache Setting		CE	CM	WE	WM
TX Mode?	Mem Op				
Yes	Read	1.45	1.44	1.13	1.42
No	Read	1.10	1.13	0.54	0.54
Yes	Write	1.20	1.17	1.07	1.70
No	Write	1.17	1.17	1.07	1.07

Table 4. Comparison of load and store instruction cycles under different conditions.

- Transactional reads are 27% to 163% slower than reads in normal mode, especially under WarmModified condition.
- Under WarmModified condition, transactional writes also have significant overhead (59%).
- Additionally, TSX shows a 70-cycle average overhead per transaction in all experiments. This may come from the memory barrier effect on the boundaries of transactions.

Summary of TSX overhead Based on our microbenchmark results, we believe the performance pathologies of TSX come from the following sources:

- **Cache Impact.** To isolate the memory accesses of transactions, TSX keeps the write set in local cache, which means the original value must be saved somewhere else. When instructions in a transactional region access a local cache line which has been previously modified but not touched in the current transaction, the CPU needs to backup the value to lower level cache in order to save the original value. This is similar to evicting dirty cache lines, but here the CPU is "cleaning" the dirty cache line. This only happens in transactional mode, and make a L1 cache hit effectively a write back to L2. We suggest that TSX has a *specialized write back buffer* to resolve this issue.
- **Memory Barrier.** According to Intel's manual [21], a successfully executed transaction has the same memory ordering semantic as "lock" prefixed instructions. When TSX is used heavily in fine-grained, memory throughput will be reduced because of the effect of memory operation serializing.
- **Time Window Inflation.** Because of the hardware overhead, the time window of a transactional region is greatly inflated. This makes the transaction more likely to abort because of conflicts, and thus waste more time on retrying. Under the best effort protocol of TSX, which always fails the transaction that accesses the data first, time window inflation makes the transaction much harder to proceed under contention.

We expect these issues to be fixed in the next generation of TSX.

5. Related Work

Concurrent Program Verification. As concurrent programs have become more widespread, techniques have been developed to perform verification on them. Checkfence [12] converts C source code into a SAT formula, which is given to a standard SAT solver to prove correctness under relaxed memory models. Sinha et al. [32] focus on improving the way that model-checkers can represent multithreaded programs as SAT formulae. Line-Up [13] can establish linearizability for concurrent methods. All of these techniques focus on converting a program (or an abstraction thereof) into a format such as a SAT formula that can be analyzed by a model-checker. All these tools suffer from the state space exploration problem, and a number of reduction techniques have been proposed. Partial order reduction [15, 16] exploits the commutativity of transitions, eliminating redundant schedules that produce the same state. Interface reduction [18] partitions the system into components and interfaces, eliminating state coupling. Symmetric reduction [30] exploits the structural symmetries of states in the system. All of these works are orthogonal to TXIT, and may be plugged into TXIT to check lock-free data structures after artificial transactions are added. Since TXIT dramatically reduces the set of schedules, these tools and techniques may become more powerful with TXIT.

Artificial Transaction Enforcement. Because of the useful guarantees that transactional memory provides, many systems leverage it to improve program reliability. BulkSC [14] proposes an implementation of a sequentially consistency memory model over the underlying relaxed memory model, by dynamically grouping instructions into "chunks", and executing chunks in the relaxed memory model. The "chunks" effectively provide artificial transactions which appear as a stronger memory model to the program. A system based on BulkSC called Atomic-Aid [25] proposes an architecture to hide atomic violations that have the potential to expose data races, by setting up "chunks" through dynamic analysis results. Both of these systems improve the program reliability through enforcing transactions, but they lack the correctness guarantees, since transactions are added to the dynamic execution streams of instructions and can be changed, where bugs may still be triggered. They are not designed to solve the fundamental trade-off between performance and verifiability.

6. Conclusion

We have presented TXIT, a system for making it easy to verify lock-free data structures, one of the most scalable and efficient among all classes of parallel programming abstractions. The key idea is to insert artificial transactions to reduce the set of schedules, while enforcing the transactions in production environment for correctness. Leveraging recent advances in hardware transactional memory support – specifically Intel Haswell TSX – TXIT achieves acceptable performance. The granularity of artificial transactions affects the performance and verifiability: larger transactions yield fewer schedules and better verifiability, but reduce performance because of the increased probability of transaction aborts. In our evaluation, we have demonstrated that TXIT reduces the set of schedules enough that tools can exhaustively check them all. In understanding the performance of TXIT, we have also uncovered several performance pathologies in TSX, knowledge of which will help other (potential) TSX users.

Acknowledgments

This work was supported in part by AFRL FA8650-11-C-7190 and FA8750-10-2-0253; ONR N00014-12-1-0166; NSF CCF-1162021, CNS-1054906; an NSF CAREER award; an AFOSR YIP award; and two Sloan Research Fellowships.

References

[1] Boost::Lockfree. `http://www.boost.org/doc/libs/1_55_0/doc/html/lockfree.html`.

[2] Folly C++ library. `http://github.com/facebook/folly`.

[3] Java's lock-free concurrency. `http://www.pwendell.com/2012/08/13/java-lock-free-deepdive.html`.

[4] liblfds. `http://liblfds.org`.

[5] Lock-free code: A false sense of security. http://www.drdobbs.com/cpp/lock-free-code-a-false-sense-of-security/210600279.

[6] Are all of the new concurrent collections lock-free? `http://blogs.msdn.com/b/pfxteam/archive/2010/01/26/9953725.aspx`.

[7] nbds. `http://nbds.googlecode.com`.

[8] Pyevolve. `http://pyevolve.sourceforge.net/`.

[9] Aba problem on wikipedia. `http://en.wikipedia.org/wiki/ABA_problem`.

[10] Transactional synchronization in haswell. `https://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell`, 2012.

[11] C. Blundell, E. C. Lewis, and M. M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–17, 2006.

[12] S. Burckhardt, R. Alur, and M. M. Martin. Checkfence: checking consistency of concurrent data types on relaxed memory models. In *ACM SIGPLAN Notices*, volume 42, pages 12–21. ACM, 2007.

[13] S. Burckhardt, C. Dern, M. Musuvathi, and R. Tan. Line-up: a complete and automatic linearizability checker. In *ACM Sigplan Notices*, volume 45, pages 330–340. ACM, 2010.

[14] L. Ceze, J. Tuck, P. Montesinos, and J. Torrellas. Bulksc: Bulk enforcement of sequential consistency. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, pages 278–289, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-706-3. . URL `http://doi.acm.org/10.1145/1250662.1250697`.

[15] C. Flanagan and P. 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. . URL `http://doi.acm.org/10.1145/1040305.1040315`.

[16] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996. ISBN 3540607617.

[17] P. Godefroid. Model checking for programming languages using verisoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 174–186, New York, NY, USA, 1997. ACM. ISBN 0-89791-853-3. . URL `http://doi.acm.org/10.1145/263699.263717`.

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

[19] D. Hendler, N. Shavit, and L. Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 206–215. ACM, 2004.

[20] M. Herlihy and J. E. B. Moss. *Transactional memory: Architectural support for lock-free data structures*, volume 21. ACM, 1993.

[21] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, June 2014.

[22] Intel. Intel 64 and IA-32 Architectures Optimization Reference Manual, March 2014.

[23] C. Killian, J. W. Anderson, R. Jhala, and A. Vahdat. Life, death, and the critical transition: Finding liveness bugs in systems code. In *Proceedings of the 4th USENIX Conference on Networked Systems Design &#38; Implementation*, NSDI'07, pages 18–18, Berkeley, CA, USA, 2007. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1973430.1973448`.

[24] C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Code Generation and Optimization, 2004. CGO 2004. International Symposium on*, pages 75–86. IEEE, 2004.

[25] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th An-*

nual International Symposium on Computer Architecture, ISCA '08, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3174-8. . URL `http://dx.doi.org/10.1109/ISCA.2008.4`.

[26] M. M. Michael. Aba prevention using single-word instructions. *IBM Research Division, RC23089 (W0401-136), Tech. Rep*, 2004.

[27] M. M. Michael. Scalable lock-free dynamic memory allocation. *ACM Sigplan Notices*, 39(6):35–46, 2004.

[28] M. Musuvathi and S. Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 2007 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. . URL `http://doi.acm.org/10.1145/1250734.1250785`.

[29] M. Musuvathi, D. Y. W. Park, A. Chou, D. R. Engler, and D. L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, Dec. 2002. ISSN 0163-5980. . URL `http://doi.acm.org/10.1145/844128.844136`.

[30] C. Norris IP and D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996. ISSN 0925-9856. . URL `http://dx.doi.org/10.1007/BF00625968`.

[31] J. Simsa, R. Bryant, and G. 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`.

[32] N. Sinha and C. Wang. Staged concurrent program analysis. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 47–56. ACM, 2010.

[33] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003. ISSN 0928-8910. . URL `http://dx.doi.org/10.1023/A%3A1022920129859`.

[34] M. Yabandeh, N. Knezevic, D. Kostic, and V. Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, volume 9, pages 229–244, 2009.

[35] J. Yang, C. Sar, and D. Engler. Explode: A lightweight, general system for finding serious storage system errors. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 10–10, Berkeley, CA, USA, 2006. USENIX Association. URL `http://dl.acm.org/citation.cfm?id=1267308.1267318`.

[36] J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, Nov. 2006. ISSN 0734-2071. . URL `http://doi.acm.org/10.1145/1189256.1189259`.

[37] 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 *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`.