Making Lock-free Data Structures Verifiable with Artificial Transactions

Xinhao Yuan, David Williams-King, Junfeng Yang and Simha Sethumadhavan
{xinhaoyuan,dwk,junfeng,simha}@cs.columbia.edu
Columbia University

Abstract

Among all classes of parallel programming abstractions, lock-free data structures are considered one of the most scalable and efficient because of 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 address this fundamental problem between performance and verifiability of lock-free data structures. 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 structures 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.*

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 [32, 24]. The reason is that 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 deadlocks, live locks, convoying, and priority inversions that plagued lock-based code [25].

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 [4]. Almost every high-level programming language/system has a (semi-)standard lock-free library, including C++'s boost::lockfree [3], and portions of C#'s System.Collections.Concurrent namespace [2] and Java's java.util.concurrent package [5].

Despite their importance, lock-free data structures remain extremely hard to get right, even for the experts, as evidenced by subtle bugs in a substantial amount of lock-free code published at good magazines and referred journals [7]. A key reason is that current executions of the operations produce a vast set of possible shared memory access interleavings, or *schedules*, whose number grows exponentially with the number of shared memory accesses. Each schedule may lead to a different, sometimes correct vs buggy, result, so all schedules must be validated for correctness, a daunting task for developers.

While much effort is dedicated to building effective tools to check parallel programming, few handle lock-free data structures. For instance, implementation-level model checkers [22, 42, 34, 49, 48, 35, 28, 44, 45] enumerate through different orders of high-level synchronizations (e.g., lock operations) for bugs. While they may be adapted to check schedules involving fine-grained shared-memory accesses, the number of these schedules would simply explode, far beyond what can be exhaustively checked—an instance of the well-known *state space explosion* problem. State space reduction techniques [21, 20, 36, 23] alleviate this problem by classifying schedules into equivalent classes so that only one schedule of each class needs to be checked for a desired property. However, their effectiveness is limited [17] because it is extremely hard to find equivalence for schedules of general programs. StableMT systems [17, 46, 47] shrink the set of schedules by over-constraining the order of high-level synchronizations, such as enforcing a round-robin ordering for all lock operations. They do not work well with lock-free data structures because enforcing an order of all shared memory accesses incurs prohibitive overhead.

This paper presents TXIT, a system that simplifies the verification of lock-free data structures. It 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 by TXIT to run atomically. These transactions are added post facto after developers have writ-

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

ten the code, hence we call them 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, TXIT continues to enforce these transactions for correctness. To reduce the overhead of enforcing transactions, TXIT leverages hardware support. 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.

A key challenge TXIT faces is correctness: making arbitrary groups of instructions atomic may introduce deadlocks or live locks, as demonstrated in prior work [13]. Fortunately, we show that TXIT does not suffer from this problem by proving that adding transactions preserves lock-freedom. We further explain the correctness and completeness of TXIT (§3).

A second challenge TXIT faces is the tradeoff of performance vs verifiability (i.e., the number of schedules). The granularity of artificial transactions determines the performance and verifiability of a lock-free data structure. Larger transactions yield fewer schedules, making the data structure much easier to verify. However, larger transactions also 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. We present an analytical model for better understanding this tradeoff, and a heuristic search engine for empirically finding a high-performing transaction placement plan given a testing budget (§4).

We implemented TXIT for C/C++ lock-free data structures. It leverages the LLVM compiler [29] to instrument programs and insert artificial transactions, the Pyevolve genetic programming engine [9] to search for an optimal transaction placement plan, the dBug model checker [40] 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 (§5).

Evaluation on six popular lock-free data structures (§6) 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 due to performance pathologies in Haswell TSX. For instance, transactional reads are (1) up to 1.63× slower than non-transactional ones and (2) always *slower* than transactional writes. We suggest two ideas to improve Haswell TSX.

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

        element *pop(stack *s) {
          element *top;
          do {
pop.1:      top = s->top;
pop.2:      element *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 its test case (b).

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 reasoning of TXIT's correctness, the analytical model for understanding the performance vs verifiability tradeoff, 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 and 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 in writing and verifying such data structures; how TXIT makes it easy to verify the example; and the recommend usage of TXIT.

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.

Figure 2 shows a schedule triggering the bug. After thread 1 gets the stack top and set `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

```
      [thread 1]                        [thread 2]
t1.1: x = pop(s) {
  pop.1: top = s->top;
  pop.2: last = top->last;
         // now last points to B
                                t2.1: x = pop(s) { ... } // pop A
                                t2.2: y = pop(s) { ... } // pop B
                                t2.3: push(s,x) { ... } // push A
  pop.3: while (...) // loop ends
}
t1.2: y = pop(s) { ... } // B
                                t2.4: free(y); // B
t1.3: free(y); // B again. ERROR
```

Figure 2: A schedule causing a double-free error.

the next element, it gets B, causing a double free. This bug is the classic ABA bug [1, 31], which is common in lock-free data structure implementations.

Finding this bug is hard because even the simple test case has an enormous number of schedules, estimated by dBug to be 9×10^{22}. Even with state-of-the-art state space reduction techniques such as dynamic partial order reduction (DPOR) [20], 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. In addition, since `pop` is now atomic, the transformation eliminates the schedule shown in Figure 2.

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).

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. To reduce the overhead of enforcing transactions, TXIT leverages hardware transactional memory.

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

```
Boolean flagA = false, flagB = false;

// thread 1                 // thread 2
while(!flagA) {}            flagA = true;
flagB = true;               while(!flagB) {}
```

Figure 3: A two-thread barrier program which executes correctly without transactions (assuming a fair scheduler), but deadlocks if both threads are made transactional.

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 at last to get high assurance in production environments.

3. Why Adding Transactions Is Valid

We have seen that grouping together multiple instructions into artificial transactions greatly reduces the number of possible schedules. However, it may be unsafe to add transactions, as shown in prior work [13]. For instance, Figure 3 shows one example taken from prior work where adding transactions causes a deadlock. When executed without transactions, this example cannot deadlock (barring an unfair scheduler which only executes one thread, or a memory model that prevents writes from being seen). If both threads are spinning, the second thread must have set *flagA* to true, so thread 1 will eventually make progress as soon as it gets a time step and sees this write. However, when all of the code shown for thread 1 is placed into one transaction, and all of the code for thread 2 into another, the system deadlocks. There is no schedule that allows either thread to proceed, because neither thread can execute its statements atomically without statements from the other thread being interleaved.

This section addresses this issue. We start with preliminaries. We use P to denote a program, and P' the program after transactions are added. We use I to denote an input, S a schedule of instructions resulting from running P or P' on an input. For ease of discussion, we call deadlocks or livelocks *liveness bugs* because they effectively make the executions dead, and all other bugs *safety bugs*.

We assume a reasonable transaction runtime that does not unnecessarily lose liveness. For instance, given two transactions that conflict when run in parallel but can both complete when run in serial, the transaction runtime should not keep aborting the transactions and then concurrently retrying them (so they have to be aborted again). This assumption is realistic, matching virtually all practical transaction runtime systems, such as the one implemented in TXIT (§5.3).

A schedule of P' may contain instructions from aborted transactions. Because of transactional atomicity, the instructions can be removed from the schedule without leading to a different output. Thus, we consider only the schedules with no aborted transactions.

We now address the correctness issue by arguing two claims.

> **Correctness Claim:** adding transactions to an arbitrary program does not introduce safety bugs. Alternatively, any safety bug in P' is a safety bug in P.

Adding transactions restricts schedules because it forces instructions in each transaction to execute atomically. However, each schedule of P' is still a schedule of P. In addition, adding transactions does not alter the operational semantics of an instruction, so running P' on input I with schedule S produces the same output as running P on I with S. If a tool finds a safety bug by running P' on I with S, this bug must be in P, too, because P on I with S produces the same output. However, the reverse is not true: P may have a safety bug, but the transactions in P' kill the corresponding buggy schedule, effectively hiding the bug. Correctness is preserved because TXIT enforces the same transactions in production environments.

> **Liveness Claim:** adding transactions to a lock-free data structure does not introduce liveness bugs. More precisely, adding transactions preserves lock-freedom.

It is possible that adding transactions to an arbitrary program removes all schedules that terminate, introducing liveness bugs as shown in Figure 3. However, we prove that this problem does not exist for lock-free data structures.

Formally, a *lock-free* program has the property that, regardless of scheduling, it always makes system-wide progress. Suppose after adding transactions, P' is no longer lock-free, i.e., there exists a schedule S that causes P' to make no progress. This liveness issue cannot be caused by the transaction runtime based on our assumption, so it must be caused by some (spin or block) waiting instruction in P' inherited from P. Thus, the same schedule S can also cause P to make no progress, contradicting with the condition that P is lock-free. An alternative way to understand this claim is, if a tool finds a liveness bug in P', then either the bug also exists in P or P is not lock-free.

Note that the above argument can be easily extended to show that adding transactions preserves other similar properties, such as *wait-freedom* (each operation completes in a bounded number of steps for any schedule; stronger than lock-freedom) and *obstruction-freedom* (any thread will make progress if run in isolation; weaker than lock-freedom).

TXIT can work with different verification tools, but it may inherit a tool's limitations on correctness and completeness. In our current implementation, TXIT leverages dBug to systematically test schedules. To check a lock-free data structure which essentially is a library, dBug requires a test case program and an input to exercise the library. Thus, it may miss bugs that cannot be triggered by the test case or input. To mitigate this limitation, developers can write representative test cases which appear straightforward for lock-free data structures because their interfaces are simple. One way to completely remove this limitation is to integrate TXIT with a static verifier.

4. Performance vs Verifiability Tradeoff

The granularity of the transactions affects verifiability (i.e., the number of schedules) and performance. Larger transactions yield fewer schedules but potentially higher overhead because they increase the chance of conflict. It is thus crucial for TXIT to select a good plan to insert transactions that make a sensible performance vs verifiability tradeoff. This section presents an analytical model to better understand this tradeoff (§4.1) and a search engine for finding an optimal transaction placement plan (§4.2).

4.1. Analytic Model

We assume an ideal program with t threads, each running a total of n instructions. We assume a transaction placement plan with fixed-size transactions of size m.

The verifiability is captured by the total number of schedules. For this ideal program, each thread has $\frac{n}{m}$ transactions, and transactions of different threads can run in different orders, so the total number of schedules is $(t(\frac{n}{m}))!/(\frac{n}{m})!^t$, a number between $t^{\frac{n}{m}}$ and $t^{t\frac{n}{m}}$. The larger the m, the more verifiable the program is. For clarity, we represent verifiability with m in the remainder of this subsection.

The performance is determined by two costs. The first cost, we call *conflict cost*, comes from concurrent executions of transactions. We define the *dependency* of a transaction as all shared memory locations it touches during its execution. Here two transactions are *dependent* if and only if there exist a shared memory location that is touched by both transactions, and at least one transaction modified it. Running two dependent transaction in parallel is called a *conflict* and must be serialized by some concurrency control, which causes at least one of the transactions to be aborted. Larger transactions usually contain more data dependencies. Moreover, larger transactions run for a longer period of time, increasing the timing window in which a conflict may occur. In both cases, larger transactions may result in a higher abort rate, degrading the parallelism of the transactions and resulting in lower system-wide throughput.

To understand the cost of conflicts, we introduce p as the uniformly independent probability for a pair of instructions to be dependent. Given a transaction of m instructions, all of its instructions must run without conflicts for the transaction to commit. Since each other thread is concurrently running a transaction of size m, each instruction has $m(t-1)$ other instructions it may conflict with. Thus the probability for each instruction to have no conflict is $(1-p)^{m(t-1)}$. The proba-

bility $s(m)$ for a transaction of size m to have no conflict is thus

$$s(m) = \left[(1-p)^{m(t-1)}\right]^m = (1-p)^{(t-1)m^2} = s_p^{m^2}$$

where $s_p = (1-p)^{(t-1)}$ which does not depend on m.

The second cost, we call *operational cost*, comes from the hardware implementation of transactions. To start and end a transaction, the hardware often needs to flush its pipeline and any modified cache lines. We represent this fixed per-transaction overhead with parameter c (we assume transactions have a working set of cache lines, so the number of modified cache lines and the overhead of flushing them will be relatively fixed). When running an instruction within a transaction, the hardware may have to do more bookkeeping, such as logging an overwritten value [33]. We represent this per-instruction overhead with a slowdown factor r which often approaches 1 for efficient hardware transaction implementations. The running time of a single transaction of size m without conflict is thus $c + rm$.

Given the two costs, we now compute the total running time of our ideal program. Considering conflicts, the running time of a transaction is $\frac{1}{s(m)}(c+rm)$. Each thread has n instructions and $\frac{n}{m}$ transactions. The expected overhead of transactional over non-transactional execution is thus

$$\begin{aligned} \mathrm{E}[overhead] &= \frac{1}{n} \cdot \frac{n}{m} \cdot \frac{1}{s(m)}(c+rm) \\ &= s_p^{-m^2}\left(\frac{c}{m} + r\right) \end{aligned}$$

We show several sample curves in Figure 4 to give a representative view of the model. We set $s_p = 0.99995$ and $r = 1$ but vary $c = \{5, 10, 15, 20\}$ because in our experience c matters the most. However, the shape of the curves do not change much for different constants. In general, large m incurs more overhead because of conflicts. However, if m becomes sufficiently small compared to the fixed per-transaction cost c, the overhead also increases because no matter the size of a transaction, it must still pay the fixed cost c. A curve with a non-zero c has a single turning point showing the optimum transaction size for performance, whereas the optimum transaction size for the $c = 0$ curve is 1.

4.2. Searching for an Optimal Placement Plan

In practice, not only the transaction size, but also the locations in which transactions are placed affects performance and verifiability. To illustrate, consider Figure 5. In (a), two synchronizations are wrapped along with a computation block that operates on thread-local data. In (b), the computation blocks are isolated as single transactions. Though (b) has more transactions, it has fewer pairs of dependent transactions, producing a smaller schedule space. On the other hand, in (a), the transactions containing the computation blocks are large, increasing the chance of aborts, so (a) may have worse performance than (b).

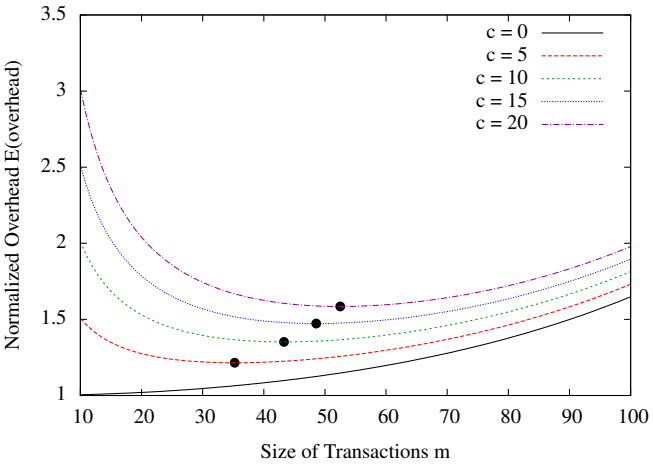


Figure 4: Performance curves for different c. When $c > 0$, each function shows a single turning point (solid points) to get the best performance.

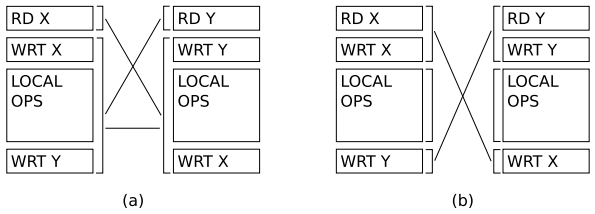


Figure 5: Two transaction placement plans for the same code. X and Y are both shared variables. Brackets on the side denote transactions. Lines between brackets denote pairs of dependent transactions.

Given the verifiability constraint on the number of schedules, the essential problem is to find appropriate code locations to place transactions for the good performance. We formalize the solution space for this search as follows. We first set up the initial plan where each operation of a lock-free data structure is a single atomic transaction. We then refine the plan by inserting *sync points* which split existing transactions into smaller ones.[2] Thus, the search algorithm can composite all possible placement plans by inserting sync points in the data structure code.

To reduce the solution space, we identify locations of shared memory accesses that may interfere with other threads by running the test case before the search. We insert sync points only before or after these locations. Thus, given n locations of shared memory accesses, the search space is essentially all bit-vectors with length $2n$, where a 1 bit indicates a sync point. In our experiments, the vectors are from 10 to 100 bits.

[2]Functions are inlined so that inserting a sync point to a callee function does not accidentally split transactions in another caller; see §5.2.

We guide the search toward a solution with (1) the best possible performance and (2) the number of schedules smaller than the testing budget s_{budget}. For (1), we estimate the number of schedules $s_{\text{est}}(x)$ using dBug. If $s_{\text{est}}(x) > s_{\text{budget}}$, we discard x immediately. For (2), we run the test case to measure the actual running time.

We solve this search problem with a genetic algorithm [18] because the solution space can be large. This technique explores the solution space by evolving the current solution (called a generation) with mutation and crossover. Given the full set of bit-vectors as the solution space, the genetic algorithm mutates a solution by randomly flipping one bit in the solution, and crossovers two solutions by exchanging their segments at the same location. We generate the initial solution by set each bit with 1 with a low probability (0.05 in our experiments). We used the Pyevolve [9] framework.

5. 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.

5.1. Architecture

TXIT takes a lock-free data structure and a 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 6.

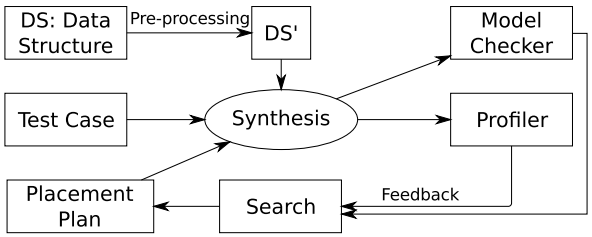


Figure 6: The architecture of TXIT. Placement plans are synthesized into a proposed program, which is profiled and checked, and which feeds into new placement plans.

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 the results as feedback. After a number of iterations, the system outputs the best placement plan it found.

Note that the system does not fully verify the source program. It finds a transaction placement plan and performs verification on the reduced set of schedules that arises. The goal is to get the best performance with a set of schedules that is still verifiable.

5.2. Pre-processing

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

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

Another transformation that needs to be performed is to eliminate all calls to the interface functions from within the library. This makes it easy to identify the boundary of the data structure and the test case (and avoid accidentally creating nested transactions). Doing so is trivial: for each exported function `f` in the library, we replace it with a dummy `f_ext` that takes the same arguments, and which passes on all its arguments to the actual function. In this way the boundary processing (entering/exiting transaction mode) can be placed in the begining and end of each dummy function.

After function level transformations, 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 them during model checking.

5.3. Intel Haswell TSX Runtime

Our work utilizes transactional memory as the architecture support to enforce transactions. Transactional Memory has been researched for many years. The newest Haswell microarchitecture from Intel brings the first hardware transactional memory—called the TSX extension—onto consumer-level processors, which makes arbitrary transaction enforcement feasible and efficient.

To utilize the TSX, we wrapped the instruction level interface into to routine, `tx_begin` and `tx_end`. The tricky thing is how TSX deals with conflicts. TSX detects the conflicts in its cache coherence protocol. By our speculation, when a transaction is executing, TSX monitors the cache line states for all local cache lines that have been touched by the transaction. Once a local cache is requested to be invalidated or degraded (e.g. from "exclusive" state to "shared" state), TSX will discover the conflict against transactions in other threads, and abort the local transaction to resolve the conflict. This makes the local transaction exits transactional mode, and jump to a fallback branch prepared before the transaction.

The way how TSX resolves conflict does not guarantee the progress of transactions. One could let failed transactions retry until success, which could lead to live lock. A simple example is shown in figure 7.

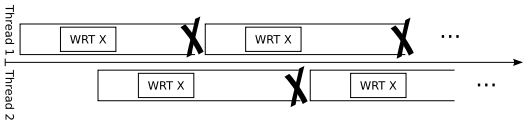


Figure 7: An example of livelock caused by retrying after aborts. Crosses denote the aborts by conflicting operations.

The optimization guidelines of TSX [26] require the programs to always provide a non-transactional fallback path for each transaction and must not simply let the transaction retries. Since the lock-free code are not aware of transaction enforcement, we need to provide our own fallback path. we involved exponential back-off strategy to resolve the contention in decentralized way. The major reason for using back-off strategy rather than fall back to global critical sections is the scalability. Transactions can conflict on splitted set of objects instead of a single set.

For example, consider a queue with multiple producers and consumers. At a point of time, producers may conflict on the tail of the queue, while consumers conflict on the head. In this situation a global critical section will still serialize all operations, while the back-off strategy will adapt the transactions into two set of serialized operations, increasing the parallism.

There are some situations which fundamentally cannot be handled by retrying. 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 such situations, where our runtime will fallback to a global critical section.

5.4. Model Checker Enhancement

To perform model checking on the program under a given transaction placement, we leverage the model checker dBug. dBug intercepts the synchronization functions (such as `pthread_mutex_lock`) to track 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. More specifically, we extend the dBug 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. 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.

To collect the read/write set of the transactions, we hook each memory access to report the current operation to the runtime with type (read/write/read-and-modify) and address.

TXIT leverages dBug to evaluate verifiability of a placement plan. This is done by extracting the schedule space estimation from dBug. dBug computes this by getting the probability sum of all explored states based on the assumption that each branch has the same probability to be taken at every node, then finds the total state space size by number of states explored divided by their probability sum.

6. 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?
- Does TXIT perform well with current hardware transactional memory? If not, why?

6.1. Evaluation Setup

Our evaluation machine is 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 imprecise 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.

Library	Data Structures Selected
boost::lockfree [3]	stack (BLFS), queue (BLFQ)
folly (Facebook) [4]	producer-consumer-queue (FPCQ)
liblfds [6]	stack (LFDSS), queue (LFDSQ)
nbds [8]	skiplist (NBDSSL)

Table 1: Evaluated lock-free data structures.

The 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 contains a bug: 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 multiple consumers and producers stack and queue, the test cases spawn three threads, where each thread pushes two elements onto the stack/queue and then pops them out. For single consumer and producer queue (FPCQ), the test case spawns a producer thread and a consumer thread, where each of them 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, allowing threads to be fully interleaved in the insertion process.

6.2. Reduction on the Number of Schedules

Here we evaluate how artificial transactions reduce the number of the schedules without applying any search. 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 8 shows the result with y-axis in log scale, demonstrating huge reduction in the number of schedules as the transaction size grows.

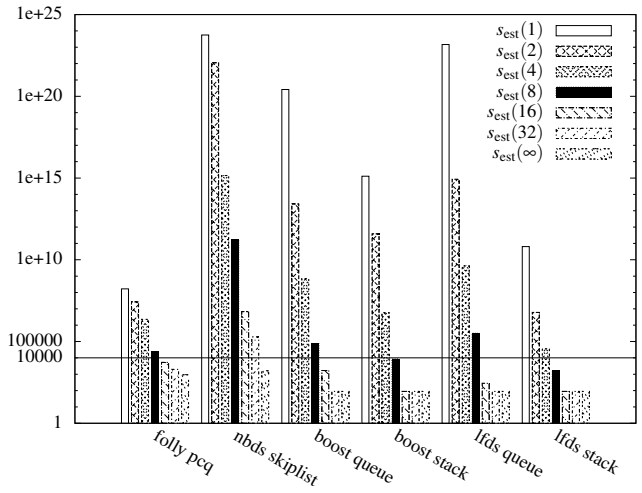


Figure 8: Number of schedules (in log scale) vs transaction size. The horizontal line is at 10^4, and any result below this line is exact.

6.3. 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} expressed as the number of schedules that developers afford to test, TXIT's heuristic search engine explores the possible transaction placement plans, evaluates the plan according to the search criteria in §4.2, 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 (§4). We 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: 70 populations 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 is made 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.

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, the smaller the 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.

Running the parallel test cases on these placement plans shows that our searching system are able to improve the performance of the baseline enforcement, by the percentage of 10.7% to 36.8%. Compared to the original performance without transactions, the numbers show the factors from 155.3% to 403.4% across all placement plans from search.

To understand what costs contributed to the overhead, we rerun these plans but using test cases that spawns only one thread; results from these experiments measure the overhead of the transactions without any conflicts (operational cost in §4.1).

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, the smaller the better.

Table 3 shows the results. Operational costs are 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 Haswell the operational cost of Haswell transactional memory is relatively large compared to the size of the transactions we insert. Next subsection examines this cost in greater detail.

6.4. 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 this scale to study the behavior of Haswell transactions. We discovered two performance pathologies that we believe are the culprit for the high operational cost. First, Haswell transactions are slow to start and end (roughly 70 cycles). Second, if a transaction writes or even *reads* a L1 data cache line modified pre-transaction, Haswell flushes this cache line to the L2 cache, each flushing costing roughly 12 cycles. These pathologies together make even transactional load instructions (1) up to 1.63× slower than non-transactional and (2) *slower* than transactional store instructions. The remaining of this section shows our detailed microbenchmarking results.

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 access 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 9:

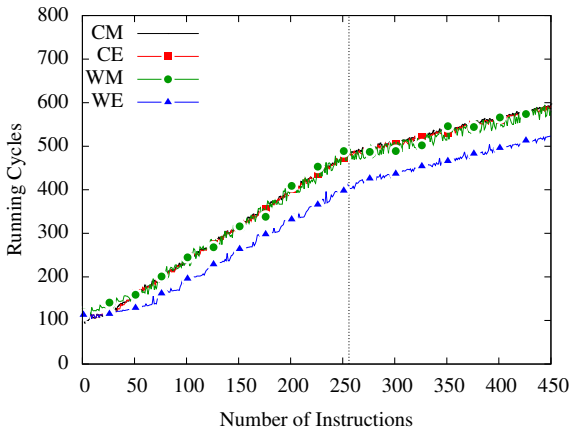
```
  1 MOVL %eax, 0x000(%rdi)  ---- start of the first pass
  2 MOVL %eax, 0x008(%rdi)
  3 MOVL %eax, 0x010(%rdi)
    ...
256 MOVL %eax, 0x1F8(%rdi)  ---- end of the first pass
257 MOVL %eax, 0x000(%rdi)  ---- repeat (second pass)
258 MOVL %eax, 0x008(%rdi)
    ...
```

Figure 9: Microbenchmark for evaluating TSX overhead.

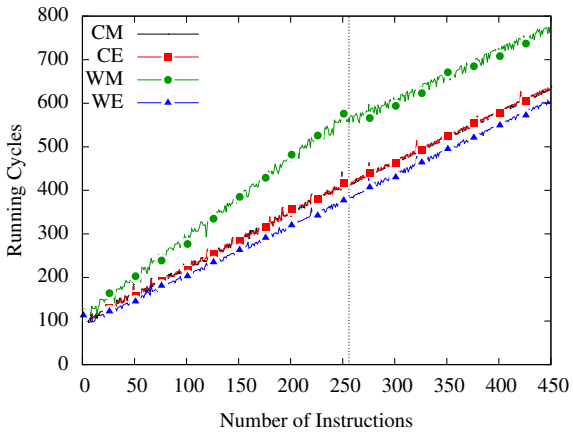
When running the benchmark function in a transaction, initially the read/write set of the transaction is empty. During the first pass of the sequence the read/write set will be filled up. By comparing the first pass to other pass we can observe the cost of bringing new cache lines into read/write set.

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, because 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



(a)



(b)

Figure 10: Running cycles for (a) read instructions and (b) write instructions, with transactions.

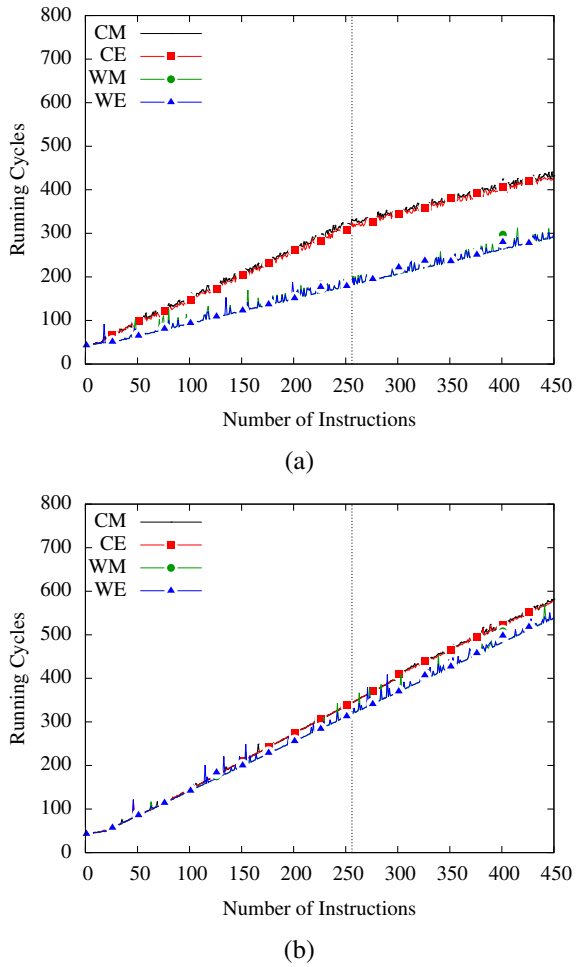
Cache Setting		CE	CM	WE	WM
TX Mode?	Mem Op				
Yes	Write	1.20	1.17	1.07	1.70
No	Write	1.17	1.17	1.07	1.07
Yes	Read	1.45	1.44	1.13	1.42
No	Read	1.10	1.13	0.54	0.54

Table 4: Transactional load and store instruction costs under different cache conditions.

two phases, where the first phase touches all cache line in the read/write set, incurring extra cost, and after that the performance stays stable. In the experiments, the first phase lasts from line `1` to `256`, which can also be observed in Figure 10. For comparison, Figure 11 shows the running cycles of the microbenchmark without transactions.

Based on data contained in these figures, we calculate the per-instruction overhead for the first phase. Table 4 shows the results. Transactional load and store instructions are quite costly in TSX in almost all cases:

- Transactional read operations are 27% to 163% slower than



(a)

(b)

Figure 11: Running cycles for (a) read instructions and (b) write instructions, without transactions.

normal mode, especially when cache is warm but modified. Based on performance counters, we believe this unexpected high cost on WarmModified comes from saving original data when bring new cache lines into read/write set, which causes cache write-backs even when cache hits.

- When cache is warm but modified, write operations in transactions also have significant overhead (59%).
- TSX shows an average of 70 cycles of constant 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 two sources:

- **Memory Barrier.** According to Intel's manual [27], a successful executed transaction has the same memory ordering semantic as "lock" prefixed instructions. We suspect that the actual ordering semantic will be stronger because of the transaction isolation requirement. When TSX is used heavily, memory throughput will be reduced because of the effect of memory operation serializing.
- **Cache Impact.** In order 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 transaction 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 expect these issues to be fixed in the next generation of TSX.

Suggestions to Reduce TSX overhead. One suggestions is to add a *Transactional Write Back Buffer* in hardware for buffering all pre-transactional data and gradually writing it back into L2, eliminating the L2 write-back latency. For a successful transaction, the buffer can be immediately discarded even if there is modified data because this data has been overwritten by the transaction. Aborted transactions need to wait for the buffer to be fully flushed before jumping the abort handler, which ideally incur a single L2 latency. Since transactions succeed much more often than not, we can avoid the L2 latency in most cases.

Another suggestion that may ameliorate the operational cost is to implement a *Transaction Checkpointing Mechanism*, which essentially combines a transaction end and an immediately following transaction start into one instruction. Thus, instead of paying the overhead of two memory barriers, we pay only one. The architecture may even pipeline the successive checkpoints, speeding up consecutive transactions that TXIT uses.

7. Related Work

Verification of Concurrent Programs. As concurrent programs have become more widespread, techniques have been developed to perform verification on them. Checkfence [14] 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. [41] focus on improving the way that model-checkers can represent multithreaded programs as SAT formulae. Line-Up [15] 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. These tools 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 may become more powerful with TXIT.

State Space Reduction. To address the state space explosion problem, a number of reduction techniques have been proposed. Partial order reduction [21, 20] exploits the commutativity of transitions, eliminating redundant schedules

that produce the same state. Interface reduction [23] partitions the system into components and interfaces, eliminating state coupling. Symmetric reduction [36] exploits the structural symmetries of states in the system. These techniques are all orthogonal to ours, since they assume all schedules are possible and attempt to eliminate redundant ones. On the other hand, TXIT reduces the possible set of schedules without requiring equivalence. Instead, it outputs a transformed program which is still correct (because the original was lock-free) and for which it is possible to verify.

Closely related to TXIT are StableMT systems [17, 46, 47], which restrict the possible schedules that can be executed, and deterministic multithreading (DMT) systems [12, 37, 19], which establish a single schedule for each input. However, these systems often work by over-constraining the order of high-level synchronizations, which performs well for lock-based code but would be prohibitively expensive when applied to lock-free code, because ordering would need to be performed at the granularity of shared memory accesses.

Similarly, the schedule specialization framework proposed by Jingyue et al [43] restrains program execution to follow a set of allowable schedules (collected from real execution traces). Then static analyses can be performed on those allowed schedules, in the knowledge that those are the only schedules that will occur. Again, this schedule-restriction mechanism works at the level of high-level synchronizations, while we approach the problem of lock-free data structures.

Artificial Transaction Enforcement. Because of the useful guarantees that transactional memory provides, many systems leverage it to improve program reliability. BulkSC [16] proposes an implementation of a sequentially consistency memory model over the underlying relaxed memory model. This is achieved by having the hardware dynamically group instructions into "chunks", which are executed at native speed 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 [30] 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. However, these systems cannot directly aid verification tools because the transactions are added to the dynamic execution streams of instructions, and may change for each execution subject to unpredictable runtime factors. Moreover, these systems lack the correctness guarantees TXIT provides, and they are not designed to solve the fundamental tradeoff of performance and verifiability.

Speculative Lock Elision. A technique related to TXIT is speculative lock elision. When transactional memory is used to elide locks in a lock-based data structure, the system speculatively skips the locking process, increasing performance [38, 39, 11]. It improves the performance of both coarse-grained and fine-grained locking, but not the difficulty of verification—the lock elision transformation is transparent and does not change locking semantics, so the set of possible schedules remains the same.

To visualize how these techniques compare, we show the performance and verifiability characteristics for each solution in Figure 12. Coarse-grained locking, fine-grained locking, and lock-free data structures gradually increase both in performance and the difficulty of verification. This trend is due to the increasingly fine granularity of synchronizations used by these techniques; the implicit synchronizations of lock-free data structures make them run even faster but are hardest of all to verify. While the speculative lock elision (dashed arrows) allows a one-way transformation to improve the performance of lock-based data structures, our approach (the solid line) makes lock-free data structures become verifiable, gradually improving performance with increasing verification capability. We can provide several points along this line in the solution space using different transaction granularities.

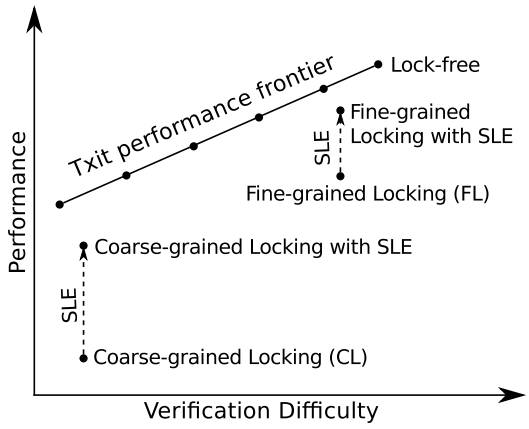


Figure 12: How speculative lock elision (SLE), applied to both coarse-grained locking and fine-grained locking, compares with our method.

8. 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. We have shown that adding transactions to arbitrary programs introduce no safety bugs, and adding transactions to lock-free data structures introduce no liveness bugs. The granularity of artificial transactions affects the performance and verifiability as larger transactions yield fewer schedules and better verifiability, but they reduce performance because they increase the

probability of transaction aborts. We have shown an analytical model for understanding this fundamental tradeoff and a practical search engine for finding a transaction placement plan that optimizes performance given a verifiability budget. In our evaluation, we have demonstrated that TXIT reduces the set of schedules enough that tools can exhaustively check. In understanding the performance TXIT, we have also uncovered several performance pathologies in TSX, knowing which will help other (potential) TSX users.

References

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

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

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

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

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

[6] liblfds. http://liblfds.org.

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

[8] nbds. http://nbds.googlecode.com.

[9] Pyevolve. http://pyevolve.sourceforge.net/.

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

[11] Yehuda Afek, Amir Levy, and Adam Morrison. Programming with hardware lock elision. In *ACM SIGPLAN Notices*, volume 48, pages 295–296. ACM, 2013.

[12] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient system-enforced deterministic parallelism. *Communications of the ACM*, 55(5):111–119, 2012.

[13] Colin Blundell, E Christopher Lewis, and Milo MK Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–17, 2006.

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

[15] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In *ACM Sigplan Notices*, volume 45, pages 330–340. ACM, 2010.

[16] Luis Ceze, James Tuck, Pablo Montesinos, and Josep 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.

[17] 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 24th ACM Symposium on Operating Systems Principles (SOSP '13)*, November 2013.

[18] Lawrence Davis et al. *Handbook of genetic algorithms*, volume 115. Van Nostrand Reinhold New York, 1991.

[19] Joseph Devietti, Brandon Lucia, Luis Ceze, and Mark Oskin. Dmp: deterministic shared memory multiprocessing. In *ACM SIGARCH Computer Architecture News*, volume 37, pages 85–96. ACM, 2009.

[20] 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.

[21] Patrice 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.

[22] Patrice 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.

[23] Huayang Guo, Ming Wu, Lidong Zhou, Gang Hu, Junfeng Yang, and Lintao Zhang. Practical software model checking via dynamic interface reduction. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, pages 265–278, New York, NY, USA, 2011. ACM.

[24] Danny Hendler, Nir Shavit, and Lena 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.

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

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

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

[28] 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.

[29] Chris Lattner and Vikram 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.

[30] Brandon Lucia, Joseph Devietti, Karin Strauss, and Luis Ceze. Atom-aid: Detecting and surviving atomicity violations. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*, ISCA '08, pages 277–288, Washington, DC, USA, 2008. IEEE Computer Society.

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

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

[33] Kevin E Moore, Jayaram Bobba, Michelle J Moravan, Mark D Hill, and David A Wood. Logtm: log-based transactional memory. In *HPCA*, volume 6, pages 254–265, 2006.

[34] Madanlal Musuvathi, David Y. W. Park, Andy Chou, Dawson R. Engler, and David L. Dill. Cmc: A pragmatic approach to model checking real code. *SIGOPS Oper. Syst. Rev.*, 36(SI):75–88, December 2002.

[35] Madanlal Musuvathi and Shaz 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.

[36] C. Norris IP and David L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1-2):41–75, 1996.

[37] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: efficient deterministic multithreading in software. *ACM Sigplan Notices*, 44(3):97–108, 2009.

[38] Ravi Rajwar and James R Goodman. Transactional lock-free execution of lock-based programs. *ACM SIGOPS Operating Systems Review*, 36(5):5–17, 2002.

[39] Amitabha Roy, Steven Hand, and Tim Harris. A runtime system for software lock elision. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 261–274. ACM, 2009.

[40] 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.

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

[42] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[43] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and precise analysis of parallel programs through schedule specialization. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 205–216, New York, NY, USA, 2012. ACM.

[44] Maysam Yabandeh, Nikola Knezevic, Dejan Kostic, and Viktor Kuncak. Crystalball: Predicting and preventing inconsistencies in deployed distributed systems. In *NSDI*, volume 9, pages 229–244, 2009.

[45] 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.

[46] Junfeng Yang, Heming Cui, and Jingyue Wu. Determinism is overrated: What really makes multithreaded programs hard to get right and what can be done about it. In *Proceedings of the 5th USENIX Workshop on Hot Topics in Parallelism (HotPar13)*, page 75. Citeseer, 2013.

[47] Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. Making parallel programs reliable with stable multithreading. *Communications of the ACM*, 57(3):58–69, 2014.

[48] Junfeng Yang, Can Sar, and Dawson 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.

[49] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. *ACM Trans. Comput. Syst.*, 24(4):393–423, November 2006.