Implementing and Evaluating a Model Checker for Transactional Memory Systems

Woongki Baek, Nathan Bronson, Christos Kozyrakis, Kunle Olukotun
Computer Systems Laboratory
Stanford University
{wkbaek, nbronson, kozyraki, kunle}@stanford.edu

***Abstract*—Transactional Memory (TM) is a promising technique that addresses the difficulty of parallel programming. Since TM takes responsibility for all concurrency control, TM systems are highly vulnerable to subtle correctness errors. Due to the difficulty of fully proving the correctness of TM systems, many of them are used without any formal correctness guarantees.**

This paper presents ChkTM, a flexible model checking environment to verify the correctness of various TM systems. ChkTM aims to model TM systems close to the implementation level to reveal as many potential bugs as possible. For example, ChkTM accurately models the version control mechanism in timestamp-based software TMs (STMs). In addition, ChkTM can flexibly model TM systems that use additional hardware components or support nested parallelism.

Using ChkTM, we model several TM systems including a widely-used industrial STM (TL2), a hybrid TM (SigTM) that uses hardware signatures, and an STM (NesTM) that supports nested parallel transactions. We then demonstrate how ChkTM can be used to find a previously unreported correctness bug in the current implementation of eager-versioning TL2. We also verify the serializability of TL2 and SigTM and strong isolation guarantees of SigTM. Finally, we quantitatively analyze ChkTM to understand the practical issues and motivate further research in model checking TM systems.

I. Introduction

Transactional Memory (TM) [16] has emerged as a promising technique that simplifies parallel programming. TM addresses the difficulty of parallel programming by allowing programmers to simply declare certain code blocks as *transactions*. TM guarantees that user-defined transactions execute in an *atomic* and *isolated* way with respect to other code blocks. Many TM systems have been proposed using hardware [14, 22], software [9, 15, 25], and hybrid techniques [3, 6].

TM takes responsibility for managing all accesses to shared state, so extreme attention must be paid to its performance. As a result, subtle but fast implementations are favored over simpler ones, even though that makes the resulting systems difficult to prove correct. TM's central position, however, means that the severity of any bug is magnified.

To address this problem, a few recent works have attempted to formally verify the correctness of TM systems using model checking techniques. In [7], Cohen et al. proposed a formal method to verify the correctness of TM systems similar to TCC [14] and LogTM [22] using the TLA+ model checker [19]. In [13], Guerraoui et al. proved an important reduction theorem that states the verification problem can be reduced to the most general problem with two threads and two shared variables when an evaluated TM system satisfies a set of certain conditions. They then verified the correctness of abstract models of several software TMs (STMs) such as TL2 [9] and DSTM [15]. Finally, O'Leary et al. [24] verified the correctness of Intel's McRT STM [25] using the Spin model checker [17].

However, there still exist research issues that require further investigations. First, TM systems should be modeled close to the implementation level, to reveal as many potential bugs as possible. For example, the TL2 model in [13] does not model the version control mechanism using timestamps, which requires a hand proof that their abstract model is equivalent to the actual implementation. Second, model checking should be extended to a wide range of TM systems that use additional hardware components (e.g., hybrid TMs) or support nested parallelism. Third, both transactional and non-transactional memory accesses should be modeled to investigate subtle correctness issues with weak isolation and ordering [27]. Finally, an in-depth, quantitative analysis should be performed to understand practical issues such as the sensitivity of the state space to various system parameters and to motivate further research in model checking TM systems.

This paper presents *ChkTM*, a flexible model checking environment for verifying the correctness of multiple TM systems. ChkTM aims to model TM systems close to the implementation level. For example, the version control mechanism of timestamp-based STMs can be accurately modeled in ChkTM using our *timestamp canonicalization* technique. Furthermore, ChkTM can flexibly model TM systems that use additional hardware components or support nested parallelism. In addition, transactional and non-transactional memory operations are also modeled in ChkTM.

The specific contributions of this work are:

- We propose a flexible model checking environment for TM (ChkTM) that can be used to verify the correctness of various TM systems. ChkTM consists of three components: (1) an architectural state space explorer, (2) TM model specifications, and (3) a test program generator.
- In ChkTM, we model several TM systems including a widely-used high-performance STM (TL2) [9], a hybrid TM (SigTM) [6] that accelerates an STM using hardware

Assume that initially x==y==0

```
// T1          // T2
atomic {       ...
 t1=x;         ...
 ...           x=1;
 t2=x;         ...
}              ...
```

Can t1!=t2?
(a) Non-repeatable reads

```
// T1              // T2
atomic {           ...
 if(y==0)          ...
  x=1;             ...
 ...               if(x==1)
 /*abort*/ }        y=2;
```

Can x==0?
(b) Speculative dirty reads

Fig. 1: Violations of strong isolation.

signatures, and an STM (NesTM) that supports nested parallel transactions [2].

- We introduce a timestamp canonicalization technique that accurately models the version control mechanism in TL2 and NesTM.
- We describe a case study in which we actually found a subtle, unreported correctness bug in the current implementation of eager-versioning TL2. We reported this bug to the TL2 developers.
- Using ChkTM, we verify that TL2 and SigTM guarantee the serializability of every possible execution of every possible program with two threads, each executing one transaction that performs at most three transactional memory operations. We also verify that SigTM provides strong isolation for the test programs described in [27].
- We perform an in-depth, quantitative analysis on ChkTM to understand the practical issues in model checking TM. We first investigate the sensitivity of the state space to system parameters such as the number of concurrent threads. Second, we study the scalability of multi-threaded ChkTM to reduce the latency of the verification. Third, we investigate the tradeoff between the performance and correctness of the verification when various approximation techniques are applied to the TM model. Our quantitative analysis also motivates further research on a reduction theorem [13] and dynamic partial order reduction techniques [12] for verifying TM systems with nested parallelism.

The rest of the paper is organized as follows. Sections II and III review the correctness criteria and the TM systems evaluated in this paper. Section IV describes the design and implementation of ChkTM. Section V presents the main correctness results and an in-depth, quantitative analysis on ChkTM. Section VI reviews related work and Section VII concludes the paper.

II. Correctness Criteria for TM

Serializability: The main correctness criterion we use in this paper is *conflict serializability* [28]. We will discuss how ChkTM verifies the serializability of every possible execution of TM programs in Section IV-A.

Strong isolation: A TM system provides *strong isolation* if transactions are isolated both from other transactions and non-transactional memory accesses [20]. Implementing strong isolation in an STM has an unattractive performance impact on all non-transactional code. Although acceptable performance has been reported using whole-program static analysis [27] or dynamic recompilation [4, 26], few STMs provide strong isolation.

In contrast, hardware TMs (HTMs) and some hybrid TMs [3, 6] (including SigTM investigated in this paper) provide strong isolation with low overheads using hardware support. Some STMs provide partial isolation guarantees that can be implemented more efficiently that strong isolation [8]. Of the TMs we examine in this paper, only SigTM provides any isolation for non-transactional accesses. There is no need to evaluate the partial guarantees for SigTM because it can implement strong isolation efficiently.

Figure 1 presents two violations of strong isolation, as discussed in [27]. Figure 1(a) illustrates a *non-repeatable read* (NR). The expected program behavior is that T1's two reads to x should observe the same value. However, since T2 updates x without using a write barrier, T1 cannot detect the conflict on x. Therefore, the two reads to x may return different values. Both lazy and eager STMs are vulnerable to NR.

Figure 1(b) illustrates a *speculative dirty read* (SDR) where a non-transactional read may observe a value speculatively written by a transaction. The expected program outcome is x==1. However, since eager STM speculatively updates shared memory in place on write (i.e., x=1), T2 may observe the value speculatively written by T1 and set y to 2. When T1 re-executes the transaction after it aborts, it will not update x because the value of y has been already updated to 2 by T2. In contrast, lazy STM is invulnerable to SDR because the updates made by a transaction are invisible (i.e., buffered) outside the transaction until it commits. We refer readers to [27] for an in-depth discussion on weak isolation behaviors in lazy and eager STMs.

III. Evaluated TM Systems

A. TL2

Lazy TL2: This STM maintains a software write buffer to perform lazy versioning [9]. It uses a global version clock to establish serializability. Using a hashing function, each memory object is associated with a version-owner lock (voLock) that either acts as a lock or stores a version number that indicates the value of global version clock at the time when the memory object was written by a committing transaction. When a transaction reads a memory object, it first checks its write buffer. If not found in the write buffer, it checks the version number in the associated voLock to check any conflict. If no conflict is detected, the address of the memory object is inserted in the read set of the transaction. When a transaction writes to a memory object, it buffers the update into its write buffer. On commit, a transaction performs the following two steps: (1) acquiring all the voLocks for the memory objects in its write set and (2) validating all the memory objects in its read set. If any of the two steps fails, the transaction aborts. Otherwise, the transaction writes back the buffered updates, increments the version number in the acquired voLocks, and finally releases the voLocks. Lazy TL2 and all other STMs

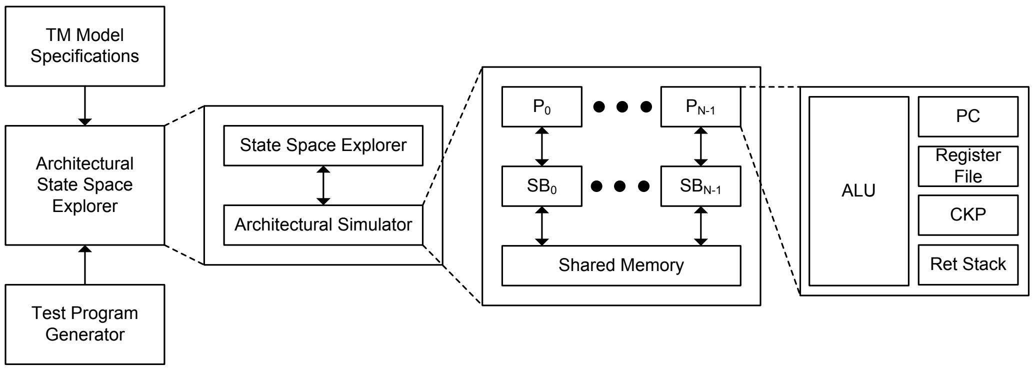


Fig. 2: The overall architecture of ChkTM.

evaluated in this paper (i.e., eager TL2 and NesTM) provide weak isolation of transactions.

Eager TL2: In contrast to lazy TL2, eager TL2 [5, 10] maintains a software undo log to perform eager (in place) versioning. The read barrier is similar to the one in lazy TL2, but does not need to check the write buffer. When a transaction writes to a memory object, it attempts to acquire the associated voLock. If this acquisition fails, the transaction aborts. Otherwise, the transaction updates the memory object in place and inserts the previous value into its undo log. On commit, a transaction validates all the memory objects in its read set. If it fails, the transaction aborts by rolling back the speculatively written memory objects and releasing the acquired voLocks. Otherwise, the transaction increments the version number in the acquired voLocks and releases them.

B. NesTM

NesTM extends eager TL2 to support closed-nested parallel transactions [2]. Following the model in [23], NesTM allows a transaction to execute only when it does not have any active children (i.e., a transaction cannot concurrently run with its children). In contrast to eager TL2, voLocks in NesTM are extended to encode the version and ownership information at the same time.

When a transaction reads a memory object, it checks the owner of the memory object. If the owner is not itself or its ancestor, the transaction aborts. Otherwise it checks the version number to check any conflict. When a transaction writes to a memory object, it attempts to acquire the associated voLock only when the owner of the memory object is itself or its ancestor. If the voLock acquisition fails, the transaction aborts. Once the transaction acquires the voLock, it validates its ancestors. If this validation fails, the transaction aborts. Otherwise, the transaction updates the memory object in place. On commit, a transaction validates all the memory objects in its read set. If this validation fails, the transaction aborts by restoring the values of memory objects and associated voLocks in its write set to the previously observed values. Otherwise, the transaction successfully commits by incrementing version number in the acquired voLocks and transferring ownership of the memory objects to its parent.

C. SigTM

Lazy SigTM: This hybrid TM accelerates a lazy-versioning STM using *hardware signatures* [6]. Hardware signatures conservatively represent the read and write sets of transactions. In addition, hardware signatures continuously snoop the coherence messages to provide conflict detection and strong isolation. Lazy SigTM uses a software write buffer to perform lazy versioning. Read and write barriers in lazy SigTM operate similarly to the ones in lazy TL2, except that software read and write sets are replaced with hardware signatures. On commit, a transaction performs write-set validation by acquiring exclusive ownership of the cache lines holding the memory objects in its write buffer. If the write-set validation fails, the transaction aborts. Otherwise, the transaction successfully commits by writing back all the buffered updates.

Eager SigTM: In contrast to lazy SigTM, eager SigTM [5] uses a software undo log to perform eager versioning. The read barrier simply inserts the address into the read signature and reads the memory value. The write barrier immediately attempts to acquire exclusive ownership of the cache line holding the memory object. If this acquisition fails, the transaction aborts. Otherwise, it updates the memory object in place and inserts the previous value into its undo log. On commit, the transaction simply resets hardware signatures. Like lazy SigTM, eager SigTM provides strong isolation.

IV. Design and Implementation of ChkTM

Figure 2 presents the overall architecture of ChkTM that consists of the following three components: (1) an architectural state space explorer, (2) TM model specifications, and (3) a test program generator. All the components are implemented in the Scala programming language [1]. We decided to use Scala because its domain-specific language features allowed us to implement ChkTM in a very concise way. We will discuss the three components of ChkTM in a bottom-up approach.

A. Architectural State Space Explorer

As illustrated in Figure 2, the architectural state space explorer (ASE) consists of two main components: (1) an architectural simulator and (2) a state space explorer. The *architectural simulator* models the internal architectural state

```
1:  procedure EXPLORE
2:    $frontiers \leftarrow InitialState$
3:    while $frontiers.isEmpty$ = false do
4:      $nextFrontiers \leftarrow \emptyset$
5:      for all $currState$ in $frontiers$ do
6:        $hasSucc \leftarrow$ false
7:        for all $trans$ in $AllPossibleTransitions$ do
8:          $hasSucc \leftarrow$ true
9:          $nextState \leftarrow trans(currState)$
10:         $transitionPair \leftarrow (currState, nextState)$
11:         if $hist$.contains($transitionPair$)=false then
12:           $hist$.put($transitionPair$)
13:           $nextFrontiers$.put($nextState$)
14:       if $hasSucc$ = false then
15:         $terminals$.put($currState$)
16:     $frontiers \leftarrow nextFrontiers$
```

Fig. 3: Pseudocode for the state space explorer.

Assume that initially x==y==0

```
// T1         // T2
atomic {      atomic {
 ld x          st x,200
 st y,101 }    ld y }

VOR(T1)={(x,0)}
VOR(T2)={(y,101)}
x=200, y=101

VOR(T1)={(x,200)}
VOR(T2)={(y,0)}
x=200, y=101
```

(a) Test code and valid results

```
T1: ld x
T2: st x,200
T2: ld y
T1: st y,101

VOR(T1)={(x,0)}
VOR(T2)={(y,0)}
x=200, y=101
```

(b) An invalid execution

Fig. 4: Detecting a serializability violation using the values observed by transactional reads (VOR).

Assume that initially x==0

```
// T1         // T2
atomic {      atomic {
 st x,100      st x,200 }
 st x,101 }

VOW(T1)={(x,0),(x,100)}
VOW(T2)={(x,101)}
x=200

VOW(T1)={(x,200),(x,100)}
VOW(T2)={(x,0)}
x=101
```

(a) Test code and valid results

```
T1: st x,100
T2: st x,200
T1: st x,101

VOW(T1)={(x,0),(x,200)}
VOW(T2)={(x,100)}
x=101
```

(b) An invalid execution

Fig. 5: Detecting a conflict-serializability violation using the values overwritten by transactional writes (VOW).

of a simple shared memory multiprocessor system consisting of processors, store buffers, and shared global memory. Each processor in the simulator models a RISC processor with an ALU, a program counter, a register file, a register checkpoint, and a return stack. Every update to shared memory is made via a bounded store buffer (SB) that may retire stores in any order, which provides a memory consistency model similar to SPARC's Total Store Order (TSO) [29]. An explicit memory fence instruction and a compare-and-swap instruction cause SB flushes. When the SB size is set to 0, the simulator models sequential consistency. We use only sequential consistency in this work and leave the verification of TM systems under relaxed consistency models as future work. Finally, all the system parameters such as the number of processors and shared memory size are fully configurable.

The second component of ASE is the *state space explorer*. In the state space explorer, states are represented as persistent trees, and state transitions are pure functions that produce a new state without altering the old. Figure 3 illustrates how the state space explorer performs a breadth-first search (BFS)[1] to determine all the possible final outcomes when executing a small program using a TM model.

To reduce the state space, we implemented a simple optimization in which instructions are merged with their successor if the instruction's dynamic execution reads and writes to only processor-private values. Since these intermediate values cannot affect the outcome, this optimization does not affect the correctness of model checking.

To verify the serializability of a TM system, ChkTM first performs *coarse-grain state space exploration* (CSE) where only a single processor is active at any time and where the active processor cannot be changed while a transaction is active. CSE directly enumerates all serial schedules. The terminal states produced by CSE are used as valid terminal states. We augment the architectural state to record the values observed by transactional reads (VOR) and the values overwritten by transactional writes (VOW). VOR and VOW are retained only for committed transactions. In addition, ChkTM ensures that every store in a test program writes a unique value. This establishes one-to-one mapping between values recorded in VOR and VOW, and the conflicting read or write operations. After CSE is performed, ChkTM records the VOR, VOW, and final memory state corresponding to every serial execution. These are the valid terminal states.

Next, ChkTM performs *fine-grain state space exploration* (FSE) which explores every possible memory access interleaving. When a terminal state is reached during FSE, ChkTM checks that the VOR, VOW, and final memory state of the terminal state are identical to one of the valid terminal states produced by CSE. Figure 4 illustrates how ChkTM detects a serializability violation using VOR and final memory state. Figure 4(a) shows a simple test program and a set of valid terminal states produced by CSE. Figure 4(b) illustrates an invalid execution trace and an invalid terminal state produced during FSE. ChkTM reports a serializability violation by detecting the terminal state reached during FSE does not match to any valid terminal state in Figure 4(a).

A serializability test using only the VOR and final memory state guarantees view serializability. To enforce conflict serializability, we add VOW. The key observation behind using VOW is that any view serializable, but not conflict serializable schedule contains a blind write [28]. VOW allows us to check that the ordering of all the write operations of a terminal state

[1]Since the state space explorer is based on BFS, it is highly parallelizable. To implement the multi-threaded state space explorer, we parallelized the main loop (i.e., line 5 in Figure 3) by dynamically assigning a chunk of iterations to each thread at a time. We quantify its scalability in Section V.

```
long TxLoad(Self, addr)
{
  if (Self.WS.contains(addr)){

    val = Self.WS.lookup(addr);

    Self.RS.insert(addr);


    return val; }
  cv = getVo(addr);
  val = *addr;
  if (isLocked(cv) || extractTS(cv)>Self.rv ||
      cv!=getVo(addr)){
    TxAbort(Self);
  }
  Self.RS.insert(addr);


  return val;
}
```

(a) Pseudocode

```
// TxLoad, BX: addr, AX: return value
instrs = instrs ++ (List(
 (0 -> new Instr {
   nextPC=TxLoad+(if (ws.contains(bx)) 10 else 40) }),
 (10 -> new Instr {
   assign(AXKey(cpu), ws.apply(bx).wsVal) }),
 (20 -> new Instr {
   rs = rs + bx
   reads = (bx, ax) :: reads }),
 (30 -> new Ret),
 (40 -> new Instr { vx = read(VoLockKeys(bx)) }),
 (50 -> new Instr { ax = read(AppMemKeys(bx)) }),
 (60 -> new Instr {
   nextPC = (if ((extractOwner(vx) != -1) ||
   (extractTS(vx)>rv) || (vx!=read(VoLockKeys(bx))))
   TxAbort else (TxLoad + 70)) }),
 (70 -> new Instr {
   rs = rs + bx
   reads = (bx, ax) :: reads }),
 (80 -> new Ret),
).map(e => (e._1 + TxLoad, e._2)))
```

(b) Modeled code

Fig. 6: A comparison between the C-language styled pseudocode and the Scala-language styled model in ChkTM for `TxLoad` in lazy TL2.

reached during FSE is identical to the one produced by a serial execution in CSE. Figure 5 demonstrates an example of how ChkTM detects a conflict serializability violation (but still view serializable) of an execution using VOW.

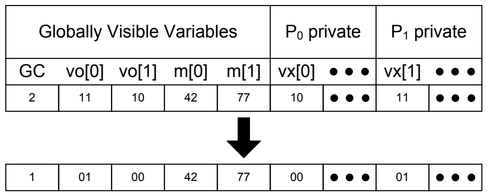
Finally, similar to the serializability verification, ChkTM checks strong isolation as follows. Given a test program, ChkTM produces all the possible expected outcomes discussed in Section II by performing CSE. ChkTM then performs FSE to explore every possible execution of the test program. If any unexpected outcome is detected during FSE, ChkTM reports a strong isolation violation.

B. TM Model Specifications

1) Modeling TL2: To model TL2, the baseline ASE is augmented with additional state variables including a global version clock, voLocks, read and write sets of transactions. Global version clock and voLocks are modeled as globally visible variables because they are accessed by multiple processors in the system. In contrast, read and write sets of each transaction are modeled as processor-private variables.

Then, the algorithms of lazy and eager TL2 are specified using a domain-specific language embedded in Scala. Figure 6 compares the C language-styled pseudocode and the model specified in Scala for `TxLoad` in lazy TL2. Figure 6 clearly demonstrates that ChkTM aims to model an evaluated TM system at the implementation level to reveal as many potential bugs as possible. Other TM barriers are also modeled similarly.

As noted in [24], it is challenging to accurately model timestamp-based STMs including TL2. This is because an infinite number of states correspond to serializable executions with different timestamp values. To address the state space explosion problem, we introduce *timestamp canonicalization*. The key observation behind this technique is that the relative ordering among timestamp values is important to order transactional events, but the exact timestamp values are unimportant. In this spirit, after each step, ChkTM canonicalizes all



Globally Visible Variables					P_0 private		P_1 private	
GC	vo[0]	vo[1]	m[0]	m[1]	vx[0]	● ● ●	vx[1]	● ● ●
2	11	10	42	77	10	● ● ●	11	● ● ●

1	01	00	42	77	00	● ● ●	01	● ● ●

Fig. 7: An example of timestamp canonicalization.

timestamps that are present anywhere in the architectural state. There are three steps: (1) compute the set of timestamp values present anywhere in the architectural state, (2) sort them, and (3) replace each value with its ordinal position in the sorted set. The canonicalized variables are global version clock (GC), voLocks, and any registers that are holding a timestamp value. Figure 7 gives an example. Note that the least significant digit (in decimal) of voLock and VX register values is not canonicalized because it encodes the ownership information.

2) Modeling NesTM: In addition to the state variables introduced to model TL2, the NesTM model uses additional state variables including a doomed bit, a rollback counter, and a commit lock per each processor. Furthermore, additional instructions such as `fork` and `join` are provided to initiate and terminate the execution of nested threads. Unlike (non-nested) TL2, the read and write sets of each processor are also modeled as globally visible variables because they can be accessed by child transactions running on different processors [2]. Finally, since the rollback counter may also cause a state space explosion, ChkTM performs *rollback-counter canonicalization*, in addition to timestamp canonicalization.

3) Modeling SigTM: To model SigTM, the baseline ASE is augmented with additional state variables to represent per-processor hardware signatures (and their state information such as `lookUp` and `WSigNack`), and transactional state information such as a doomed bit. All of them are modeled as

```
atomic {
 if (exec) {
  if (read) { TxLoad(genAddr); }
  else { TxStore(genAddr,genUniqVal); }}
 // repeat the process as necessary
 ... }
```

Fig. 8: Skeleton transaction code used for tests.

Assume that initially x==y==0

```
// T1             // T2
atomic {          atomic {
  st x,1            st y,2
  ld y }            ld x }
```

Can VOR(T1)=={(y,2)} and VOR(T2)=={(x,1)}?

Fig. 9: A simple test program used to test (buggy) eager TL2.

```
   //T1:TxLoad                      //T2:TxStore,TxAbort
0: cv=getVo(addr)                   ...
1: ...                              lock(addr)
2: ...                              *addr=2
3: val=*addr // 2!                  ...
4: ...                              *addr=0
5: ...                              unlock(addr)
6: if(...cv!=getVo(addr)){          ...
7:  ... }                           ...
```

Fig. 10: An unserializable execution detected by ChkTM in (buggy) eager TL2.

```
1: procedure TxAbort
2:   $rs$.reset()
3:   for all $addr$ in $ws$ do
4:     Memory[$addr$] $\leftarrow$ $ws$.lookup($addr$)
5:   for all $addr$ in $ws$ do
6:     unlock($addr$)        ▷ Timestamp value should have been incremented.
7:   $ws$.reset()
8:   doContentionManagement()
9:   restoreCheckpoint()
```

Fig. 11: A bug (in line 6) found in TxAbort in eager TL2.

globally visible state variables because they can be accessed by other processors. Unlike TL2 and NesTM, timestamp variables are unnecessary to model SigTM. Additional memory operations are also modeled including fetchExclusive and non-transactional load and store instructions.

Currently, ChkTM models SigTM using a zero-latency interconnection network. We expect that prior work on model checking cache coherence protocols [18] can be used to extend the current hardware model in ChkTM. We leave this as future work.

C. Test Program Generator

To verify the serializability of TM systems, the test program generator of ChkTM uses the skeleton code shown in Figure 8. A transaction executes a specified number of transactional memory operations. For each memory operation, there are three parameters to consider: (1) whether it is performed or not, (2) the type of memory operation (i.e., read or write), and (3) the accessed memory location. All these parameters can be generated either randomly (for random tests) or systematically (for exhaustive, complete tests). Also note that the value written by each write operation is unique with respect to all other written values. Finally, to verify the strong isolation guarantees of TM, we manually coded each problematic scenario described in [27].

D. Case Study: Invalid-read Problem

To showcase ChkTM's effectiveness, we present a case study in which we found an unreported *invalid-read* bug in the available implementation of eager TL2. To verify the correctness of eager TL2, we modeled it and performed exhaustive tests using ChkTM. During the tests, ChkTM reported a serializability violation with the test program in Figure 9 and generated problematic executions including the one shown in Figure 10.

To investigate the cause of the problem, we analyze the invalid execution in Figure 10. T1 executes the code in TxLoad, while T2 executes the code in TxStore and TxAbort. At step 0, T1 samples the current value of the voLock for addr which is the address of y. At steps 1 and 2, T2 acquires the voLock and writes a value of 2 to addr during the execution of TxStore. At step 3, T1 reads the speculatively written (i.e., 2) value from addr. At steps 4 and 5, during the execution of TxAbort, T2 restores the memory and voLock values for addr to the previously observed values. At step 6, the if statement tests whether the value of the voLock read at step 0 matches the current value of the voLock (i.e., cv!=getVo(addr)?). Since T2 has already restored the value of voLock to the previous value (i.e., the same value as cv), the test at step 6 fails (i.e., cv==getVo(addr)). Therefore, T1 assumes the memory value read at step 3 is still valid and successfully commits in the end. This is an execution scenario in which the invalid outcome in Figure 9 is produced.

By analyzing the invalid execution, we identified that there is a subtle correctness bug in line 6 in TxAbort's code shown in Figure 11. To avoid the invalid-read problem, an aborting transaction must increment the timestamp values of the acquired voLocks instead of merely restoring them to the previous values. We reported this bug to the TL2 developers.

In addition, readers should note that it may be very difficult to find this kind of subtle bugs by performing random tests using the implemented TM code on real machines. One can increase the possibility of revealing such bugs from random tests by inserting randomized delays at various locations in the TM code (e.g., between the operations at steps 0 and 3 in Figure 10). However, it requires programmer's non-trivial intuitions on where potential bugs would be. In contrast, ChkTM can reveal any potential bug without requiring programmer's intuitions because it explores every possible execution of a TM program, even unlikely ones.

E. Discussion

State space reduction: While we implemented a simple optimization that merges locally visible steps to reduce the state space in ChkTM, it is an interesting, open question to implement more aggressive optimizations, such as partial

TM	Tot. time (s)	Avg. time (s)	Avg. # of states
TL2-L	7971	0.51	2115
TL2-E	13453	0.86	10362
SigTM-L	10494	0.67	4931
SigTM-E	8227	0.53	2279

TABLE I: Total, average execution time, and average number of explored states required to verify the serializability of TM systems.

Anomaly	TL2-L	TL2-E	SigTM-L	SigTM-E
NR	Y	Y	N	N
ILU	Y	Y	N	N
IDR	N	Y	N	N
SLU	N	Y	N	N
SDR	N	Y	N	N
OW	Y	N	N	N
BW	Y	N	N	N

TABLE II: Summary of weak isolation anomalies in TL2 and SigTM.

order reduction (POR) techniques [12]. As noted in [24], traditional POR techniques cannot be directly applied due to their conservativeness when iterations or global variables are used in the TM model. In particular, we note that it will be interesting to extend dynamic POR techniques [11] in the context of model checking TM systems.

Modeling other TM systems: Because ChkTM addresses the state space explosion problem due to the use of timestamps (or any kind of counters) by timestamp canonicalization, it is straightforward to model other STMs in ChkTM. To model other hybrid or hardware TM systems, ChkTM should be augmented with additional state variables to model hardware components used in the evaluated TM systems. An open question is how to model asynchronous communications with hardware components without causing a state space explosion.

Verifying liveness: ChkTM could be extended to verify the liveness of TM systems by detecting cycles in the state transition graph. The existence of a cycle indicates that there is an execution scenario in which a TM program can never terminate (e.g., transactions infinitely abort and restart). Note that all the evaluated TM systems in this paper are known to admit livelock, but attempt to probabilistically provide liveness using randomized backoff schemes.

V. Evaluation

In this section, we first discuss our main correctness results for TL2 and SigTM, and describe our progress in verifying NesTM. We then present an in-depth, quantitative analysis on ChkTM to understand the practical issues in model checking TM systems. More specifically, we investigate (1) the sensitivity of the execution time and the number of the explored states to the parameters such as test program size and number of threads, (2) the scalability of the multi-threaded ChkTM, and (3) the tradeoff between the performance and correctness of the verification when approximation techniques are applied to the TM model.

We performed experiments on computers with two quad-core 2.33GHz Intel®Xeon®CPUs and 32GB of shared memory. We used Linux `x86_64` kernel 2.6.18 and the 64-bit Server VM in Sun's Java^{TM}SE Runtime Environment, build 1.6.0-14-b08. We used an 8GB heap for model checking TL2 and SigTM and a 30GB heap for model checking NesTM. Finally, we used the Scala compiler version 2.7.5 to compile the ChkTM code.

A. Correctness Results

Serializability: To verify the serializability of TL2 and SigTM, we first generated all the possible test programs that satisfy the following conditions[2]: (1) two threads in each program, (2) one transaction per thread, (3) at most three transactional memory operations (i.e., read or write) per transaction, (4) no non-transactional memory operations in the program (i.e., purely transactional). In addition, we assumed that the underlying system has two shared memory words and provides sequential consistency.

We then ran all the generated test programs on the TL2 and SigTM models in ChkTM and verified that no errors were reported. Therefore, we make the following statement:

TL2 and SigTM (both lazy and eager) guarantee the serializability of every possible execution of every possible program that runs two threads, each of which executes one transaction that performs no more than three transactional memory operations.

Table I summarizes the execution time and number of explored states required to verify each TM system. It takes less than a second to run each test and 2–4 hours to exhaustively verify each TM system with all the possible test programs. Table I shows that the explored state space in verifying lazy TL2 is smaller than the ones with SigTM. Since read sets and write buffers of lazy TL2 can be modeled as processor-private variables, our optimization that merges locally visible steps is effective in reducing the state space. In contrast, it is less effective for SigTM because read and write signatures in SigTM are modeled as globally visible variables. Table I also demonstrates that the explored state space in model checking eager TL2 is significantly larger than other TM systems. Since the timestamp values are incremented even when a transaction aborts to avoid the invalid-read problem, more unique states associated with the incremented timestamp values are generated and explored in verifying eager TL2.

Strong isolation: To investigate the strong isolation guarantees of TL2 and SigTM, we manually coded the test programs presented in [27]. We tested seven of the nine anomalies, including non-repeatable reads (NR), intermediate lost updates (ILU), intermediate dirty reads (IDR), speculative lost updates (SLU), speculative dirty reads (SDR), overlapped writes (OW), and buffered writes (BW). We omitted granularity-related anomalies as they can only be solved at a level higher than a word-based STM. Table II summarizes the results. For every possible execution of all the test programs, SigTM (both lazy and eager) did not produce any unexpected final outcome. In contrast, ChkTM successfully detected weak isolation anomalies for TL2. In summary, we were able to mechanically verify

[2]The use of this configuration was inspired by the approach discussed in [24].

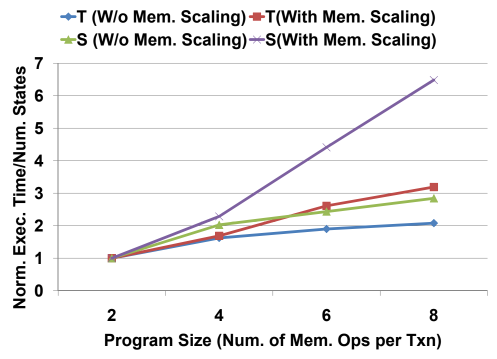


Fig. 12: Sensitivity of the execution time and number of explored states to the test program and shared memory sizes.

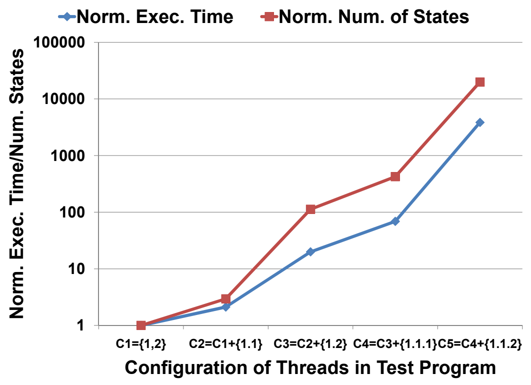


Fig. 13: Sensitivity of the execution time and number of explored states to the number of threads.

the arguments in [27] using the memory operations (both transactional and non-transactional) modeled in ChkTM.

B. Quantitative Analysis

Sensitivity to the system parameters: We investigate the sensitivity of the execution time and number of explored states in verifying TM to various system parameters. The parameters on which we focus are program and shared memory sizes, and number of threads. Figure 12 illustrates the results for the program and shared memory sizes. Program size represents the number of memory operations performed by each transaction. Shared memory size denotes the number of shared memory words in the system. The first result data set is collected by just scaling the program size while the shared memory size is fixed at two (i.e., two shared memory words in the system). In contrast, the second result data set is collected by incrementing the shared memory size by 1 when the program size is incremented by 2. In the baseline configuration, program and shared memory sizes are set to 2. All the results are normalized to the baseline result. Finally, the result at each configuration is collected by running 100 randomly generated test programs on the lazy TL2 model.

Figure 12 shows that the execution time and number of states increase rather slowly when only the program size increases. With a small shared memory size, each operation conflicts with other operations with a high probability. This limits the number of possible interleavings as each operation has dependencies with other operations due to the conflicts. Therefore, the state space grows relatively marginally even when each transaction performs more operations. In contrast, with a larger shared memory size, each operation can execute more independently with respect to other operations due to lower contention. Therefore, more interleavings can be produced and the state space grows much faster when both program and shared memory sizes increase.

We also investigate the sensitivity of the execution time and number of states to the number and configuration of threads when checking NesTM. Figure 13 presents the sensitivity results to different number and configuration of threads. Specifically, C1 is the configuration in which two top-level transactions (T1 and T2) run. C2 is generated by adding T1.1 (a child of T1) to C1. C3 is generated by adding T1.2 (a child of T1 and sibling of T1.1) to C2. C4 and C5 are repeatedly generated in a similar manner. Each transaction executes only two reads, to reduce the state space.

Figure 13 shows that the execution time and number of states required to verify NesTM increases explosively when the number of threads rises (note that y-axis in Figure 13 has a log scale). One interesting observation is that the increase is largest when a new sibling thread is added (e.g., from C2 to C3, from C4 to C5). This is due to the fact that there is no ordering dependency between sibling transactions, thus more interleavings can be produced. In contrast, when a new (single) child thread is added (e.g., from C1 to C2, from C3 to C4), the increase is relatively smaller. This is because the newly added child has an ordering dependency with its parent, thus the number of possible interleavings is limited. For instance, T1.1 can run only after T1 forks it, and T1 is suspended while T1.1 is active.

The result in Figure 13 clearly motivates the need for a reduction theorem applicable to nested parallel TM. While two threads and two variables are proved sufficient for non-nested TM in [13], no such guarantee of completeness is available for our checks of NesTM. Since the state space explosively grows with the increasing number of threads and nesting levels, model checking may be feasible only for small configurations. For example, when larger test programs are used with the thread configuration C5, it is currently not possible to verify NesTM using ChkTM even on computers with 32GB physical memory due to a state space explosion. Finally, dynamic state space reduction techniques should be also investigated to make it feasible to model check the nested TM systems on commodity machines.

Scalability of the multi-threaded ChkTM: To investigate the scalability of the multi-threaded ChkTM, we use a test program with the thread configuration C5 on the NesTM model. To avoid a state space explosion, each transaction in the test program performs only two reads. We varied the number of concurrent threads from 1 to 8, and measured the speedup by dividing the execution time with multiple threads by the one with 1 thread. As shown in Figure 14, the multi-threaded

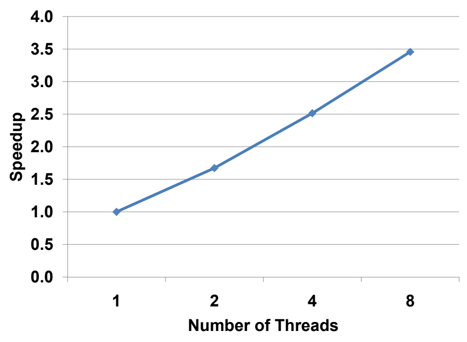


Fig. 14: Speedup of the multi-threaded ChkTM.

Aprx.	Loc.	Description
#1	L	Atomically read voLock and mem. value
#2	C	Atomically release voLocks
#3	A	Atomically release voLocks
#4	S	Atomically insert a WS entry and write mem. value
#5	A	Atomically rollback mem. values
#6	C,A,S	All except for Approx. 1

TABLE III: Approximations applied to NesTM models. L, S, C, and A denote `TxLoad`, `TxStore`, `TxCommit`, and `TxAbort`.

ChkTM shows reasonable scalability when model checking NesTM (e.g., speedup of 3.5× with 8 threads).

Tradeoff between performance and correctness: One commonly-used technique to reduce the state space is to apply approximations by merging several steps of the modeled algorithm into one. Obviously, a careless use of such approximations may affect the correctness of the verification. To investigate the tradeoff between the performance and correctness of the verification, we apply a set of approximations when verifying NesTM. Table III summarizes such approximations. For example, Approximation 1 merges the steps of reading the voLock and the value of a memory object (i.e., the operations at step 0 and 3 in Figure 10 are assumed to happen atomically). For another example, Approximation 2 releases all the acquired voLocks on commit in a single step (i.e., the loop is atomically executed).

Table IV summarizes the average execution time and the number of explored states for verifying the baseline and approximated NesTM models. The results were collected using 10 randomly generated test programs in which the thread configuration C3 is used and each transaction executes at most two memory operations. Table IV shows that Approximation 2 is effective in reducing the state space. On commit, a transaction tends to have more entries in its write set because it has executed all of its memory operations. Therefore, Approximation 2 can effectively reduce the state space by merging more steps in the algorithm. In contrast, approximations applied to the abort barrier are not as effective. On abort, a transaction tends to have less entries in its write set because it often fails to execute all of its memory operations. Therefore the approximations applied to the abort barrier are less effective as they merge fewer steps. In summary, Approximation 6 (all the approximations except for Approximation 1) reduces the execution time and number of states by 29.1% and 35.6%, respectively.

Ver	Time (s)	# of states	Fidelity
Base	11.70	619,864	Y
Aprx. 1	10.51	533,996	N
Aprx. 2	8.82	415,530	Y
Aprx. 3	11.79	617,851	Y
Aprx. 4	11.46	602,791	Y
Aprx. 5	11.52	618,011	Y
Aprx. 6	8.29	399,058	Y

TABLE IV: Average execution time, number of explored states, and the fidelity of the baseline and approximated NesTM models.

Table IV also summarizes the fidelity of the baseline and approximated NesTM models when the invalid-read bug discussed in Section IV-D is intentionally injected. With the injected bug, the model with Approximation 1 does not report an error (i.e., false negative). This is because merging the aforementioned steps makes it impossible to produce any invalid execution similar to the one shown in Figure 10. To our surprise, other approximated models do not show any false negative (at least for the invalid-read bug with a set of test programs we used). If it can be proven that a particular approximation does not affect the correctness of model checking, it can be used as an effective tool to reduce the state space.

VI. Related Work

While there is a large amount of previous work on TM, relatively little work has been done on formally verifying TM systems. Cohen et al. [7] proposed a formal method to verify the correctness of a few TM systems similar to TCC [14] and LogTM [22] using the TLA+ model checker [19]. Guerraoui et al. [13] proved an important reduction theorem that states the TM verification problem can be reduced to the most general problem with two threads and two shared variables, when an evaluated TM satisfies a set of certain conditions. In addition, they verified the correctness of abstract models of several STMs including DSTM and TL2. While insightful, these prior theoretical works modeled the evaluated TM systems rather abstractly. For example, the TL2 model in [13] does not model the version control mechanism using timestamps, which requires a hand proof that their abstract model is equivalent to the actual implementation. In contrast, ChkTM aims to model the evaluated TM systems close to the implementation level, to aggressively find potential bugs.

In [24], O'Leary et al. verified the correctness of Intel's McRT STM using the Spin model checker [17]. Our work is similar to theirs in the sense that both works attempt to model TM systems at the implementation level. Our work, however, significantly differs in the following three aspects. First, we extend the use cases of model checking TM by investigating a wider range of TM systems, including an industrial, high-performance STM (TL2), a hybrid TM (SigTM) that uses hardware signatures, and an STM (NesTM) that supports nested parallel transactions. Second, ChkTM models both transactional and non-transactional memory operations to enable our study on subtle correctness issues with weak isolation and ordering. Finally, we provide an in-depth, quantitative analysis on ChkTM to understand the practical issues and motivate further research in model checking TM systems.

Finally, Manovit et al. proposed an axiomatic formulation to model the formal specification of a TM system and used it with random testing to find bugs in the evaluated TM system [21]. Our work differs in the sense that ChkTM aims to formally verify the correctness of TM systems by model checking all possible executions.

VII. Conclusion

This paper presents ChkTM, a flexible model checking environment for verifying the correctness of various TM systems. ChkTM aims to model TM systems at the implementation level to reveal as many potential bugs as possible. Using ChkTM, we found a subtle, unreported correctness bug in the current implementation of eager TL2. We also verified the serializability of TL2 and SigTM and strong isolation guarantees of SigTM. We quantitatively analyzed ChkTM to understand the practical issues in model checking TM systems. Finally, our quantitative study motivates further research such as investigating a reduction theorem and dynamic partial order reduction techniques for verifying TM systems with nested parallelism without causing a state space explosion.

Acknowledgements

We would like to thank the anonymous reviewers for their feedback. We also want to thank Sun Microsystems for making the TL2 code available. Woongki Baek was supported by a Samsung Scholarship and an STMicroelectronics Stanford Graduate Fellowship. This work was supported by NSF Awards number 0546060, the Stanford Pervasive Parallelism Lab, and the Gigascale Systems Research Center (GSRC).

References

[1] The Scala Programming Language. http://www.scala-lang.org.

[2] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and Evaluating Nested Parallel Transactions in Software Transactional Memory. In Submission.

[3] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly-atomic hybrid transactional memory. In *Proceedings of the 35th Annual International Symposium on Computer Architecture*. Jun 2008.

[4] N. G. Bronson, C. Kozyrakis, and K. Olukotun. Feedback-directed barrier optimization in a strongly isolated STM. In *POPL '09: Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 213–225, New York, NY, USA, 2009. ACM.

[5] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[6] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*. June 2007.

[7] A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pages 37–44. November 2007.

[8] L. Dalessandro and M. L. Scott. Strong isolation is a weak idea. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[9] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC'06: Proceedings of the 20th International Symposium on Distributed Computing*, March 2006.

[10] D. Dice and N. Shavit. Understanding tradeoffs in software transactional memory. In *CGO '07: Proceedings of the International Symposium on Code Generation and Optimization*, March 2007.

[11] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM.

[12] 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.

[13] R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *PLDI '08: Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, pages 372–382, New York, NY, USA, 2008. ACM.

[14] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st International Symposium on Computer Architecture*, pages 102–113, June 2004.

[15] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, July 2003. ACM Press.

[16] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 289–300, 1993.

[17] G. Holzmann. *Spin model checker, the: primer and reference manual*. Addison-Wesley Professional, 2003.

[18] R. Joshi, L. Lamport, J. Matthews, S. Tasiran, M. Tuttle, and Y. Yu. Checking cache-coherence protocols with tla+. *Form. Methods Syst. Des.*, 22(2):125–131, 2003.

[19] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[20] J. Larus and R. Rajwar. *Transactional Memory*. Morgan Claypool Synthesis Series, 2006.

[21] C. Manovit, S. Hangal, H. Chafi, A. McDonald, C. Kozyrakis, and K. Olukotun. Testing implementations of transactional memory. In *PACT '06: Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, pages 134–143, New York, NY, USA, 2006. ACM.

[22] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-Based Transactional Memory. In *12th International Conference on High-Performance Computer Architecture*, February 2006.

[23] J. E. B. Moss and T. Hosking. Nested Transactional Memory: Model and Preliminary Architecture Sketches. In *OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL)*. University of Rochester, October 2005.

[24] J. O'Leary, B. Saha, and M. R. Tuttle. Model checking transactional memory with spin. *Distributed Computing Systems, International Conference on*, 0:335–342, 2009.

[25] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: A high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the 11th ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, March 2006. ACM Press.

[26] F. T. Schneider, V. Menon, T. Shpeisman, and A.-R. Adl-Tabatabai. Dynamic optimization for efficient strong atomicity. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-Oriented Programming Systems, Languages, and Applications*, New York, NY, USA, October 2008. ACM.

[27] T. Shpeisman, V. Menon, A.-R. Adl-Tabatabai, S. Balensiefer, D. Grossman, R. L. Hudson, K. Moore, and B. Saha. Enforcing isolation and ordering in STM. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation*, June 2007.

[28] A. Silberschatz, H. Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 2006.

[29] C. SPARC International, Inc. *The SPARC architecture manual (version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.