Proving Non-opacity

Mohsen Lesani Jens Palsberg
University of California, Los Angeles
{lesani, palsberg}@ucla.edu

Abstract

Guerraoui and Kapalka defined opacity as a safety criterion for transactional memory algorithms in 2008. Researchers have shown how to prove opacity, while little is known about pitfalls that can lead to non-opacity. In this paper, we identify two problems that lead to non-opacity and we prove an impossibility result. We first show that the well-known TM algorithms DSTM and McRT don't satisfy opacity. DSTM suffers from a write-skew anomaly, while McRT suffers from a write-exposure anomaly. We then prove that for direct-update TM algorithms, opacity is incompatible with a liveness criterion called local progress, even for fault-free systems. Our result implies that if TM algorithm designers want both opacity and local progress, they should avoid direct-update algorithms.

1. Introduction

Transactional memory. Atomic statements can simplify concurrent programming that involves shared memory. Transactional memory (TM) [21, 33] interleaves the bodies of atomic statements as much as possible, while guaranteeing noninterleaving semantics. Thus, the noninterleaving in the semantics can coexist with a high degree of parallelism in the implementation. TM aborts an operation that cannot complete without violating the semantics. The use of TM provides atomicity, deadlock freedom, and composability [18], and increases programmer productivity compared to use of locks [28, 30]. Researchers have developed formal semantics [1, 24, 27] and a wide variety of implementations of the TM interface in both software [6, 7, 19, 20, 31] and hardware [2, 16]. Intel supports transactional synchronization primitives in its new processor microarchitecture codenamed Haswell.

Safety. A TM interface consists of the operations `read`, `write`, and `commit`. The task of a TM algorithm is to implement those three operations. What is a correct TM algorithm? The traditional safety criterion for database transactions is strict serializability [29]. For TM algorithms, strict serializability [32] requires that committed transactions together have an equivalent sequential execution, that is, an execution that could also happen if the transactions execute noninterleaved. However, to ensure semantic correctness, active and aborted transactions should execute correctly too. This observation has led researchers to define the stronger safety criteria opacity [11], VWC [23], and TMS1 [8]. We will focus on opacity, which is the strongest safety criterion and requires *all* transactions together to have an equivalent sequential execution.

Verification. Researchers have shown how to verify the safety of TM algorithms. In pioneering work, Tasiran [34] proved serializability for a class of TM algorithms. Cohen et al. [4, 5] were the first to use a model checker to verify strict serializability of TM algorithms for a bounded number of threads and memory locations. Later, Guerraoui and Kapalka [15] proved opacity of two-phase locking with a graph-based approach that is related to an earlier approach to serializability. Guerraoui et al. [12–14] used a model checker to verify opacity of TM algorithms that use an unbounded number of threads and memory locations. One of their key innovations was a reduction theorem that says that an algorithm is correct for any number of threads and memory locations if and only if it is correct for two threads and two memory locations. Their theorem relies on four assumptions about TM algorithms. In follow-up work, Emmi et al. [9] showed how to use a theorem prover to generate the invariants that are needed to prove strict serializability. Their proofs work for TM algorithms that use an unbounded number of threads and memory locations, and don't rely on a reduction theorem. Later, Lesani et al. [25] presented a verification framework for transactional memory based on IO automata and simulation.

The problem: Which pitfalls lead to non-opacity?

Our results: We identify two problems that lead to non-opacity, we find problems with DSTM and McRT, and we prove an impossibility result.

We show that the well-known TM algorithms DSTM and McRT don't satisfy opacity. These results may be surprising because previous work has proved that DSTM and McRT satisfy opacity [9, 13, 14]. However, there is no conflict and no mystery: the previous work focused on abstractions of DSTM and McRT, while we work with specifications that are much closer to original formulations of DSTM and McRT. Thus, we experience a common phenomenon: once we refine a specification, we may lose some properties.

Let us recall common terminology. A TM algorithm is a *deferred-update* algorithm if every transaction that writes a value must commit before other transactions can read that value. All other TM algorithms are *direct-update* algorithms. DSTM is a deferred-update algorithm while McRT is a direct-update algorithm.

DSTM suffers from a write-skew anomaly, while McRT suffers from a write-exposure anomaly. The write-skew anomaly is an incorrectness pattern that is known in the database community [10]. The write-exposure anomaly happens when a direct-update TM algorithm exposes written values to other transactions before the transaction commits.

We present fixes to both DSTM and McRT that we conjecture make the fixed algorithms satisfy opacity. Interestingly, we note that writers can limit the progress of readers in the fixed McRT algorithm. This is an instance of a general pattern: we prove that for direct-update TM algorithms, opacity is incompatible with a liveness criterion called local progress [3], even for fault-free systems. Our result implies that if TM algorithm designers want both opacity and local progress, they should avoid direct-update algorithms.

We hope that our observations can help TM algorithm designers to avoid the write-skew and write-exposure pitfalls, and to be aware that if local progress is a goal, then deferred-update algorithms may be the only option.

The rest of the paper. In Section 2, we formalize opacity. In Sections 3–4, we prove that core versions of DSTM and McRT don't satisfy opacity. In Section 5, we prove that for direct-update TM algorithms, opacity and local progress are incompatible.

$$
\begin{aligned}
Writes(H) &= \{W \mid \exists T \in H, i \in I, v \in V \colon W = write_T(i,v) \Subset H\} \\
TSequential &= \{S \in THistory \mid \leq_S \text{ is a total order of } Trans\} \\
Committed(H) &= \{T \in Trans \mid ret_T(C) \Subset H\} \\
Aborted(H) &= \{T \in Trans \mid ret_T(A) \Subset H\}, \\
Completed(H) &= Committed(H) \cup Aborted(H), \\
Live(H) &= \{T \in H \mid T \notin Completed(H)\} \\
TComplete &= \{H \in THistory \mid \forall T \in H \colon T \in Completed(H)\} \\
Pending(H) &= \{T \in Live(H) \mid T \text{ has a pending event in } H\} \\
CommitPending(H) &= \{T \in Pending(H) \mid T \text{ has a pending } inv_T(commit_T) \text{ event in } H\} \\
TExtension(H) &= \{H' \in THistory \mid H \text{ is a prefix of } H' \ \wedge \ \forall T \in H' \Rightarrow T \in H \ \wedge \\
&\qquad Live(H) \setminus CommitPending(H) \subseteq Aborted(H') \ \wedge \\
&\qquad CommitPending(H) \subseteq Completed(H')\} \\
Visible(S,T) &= filter\left(S, \lambda T'.(T' = T) \vee \left((T' <_S T) \wedge T' \in Committed(S)\right)\right) \\
NoWriteBetween_S(W,R) &= \forall W' \in Writes(S) \colon W' \preceq_S W \vee R \prec_S W' \\
SeqSpec(i) &= \{S \in Sequential \mid \forall T \in S \colon \forall v \in V \colon \\
&\qquad \forall R = read_T(i){:}v \Subset S \colon \exists T' \in S \colon \\
&\qquad \exists W = write_{T'}(i,v) \Subset S \colon \\
&\qquad W \prec_S R \wedge NoWriteBetween_S(W,R)\} \\
TSeqSpec &= \{S \in TSequential \cap TComplete \mid \forall T \in S \colon \forall i \in I \colon \\
&\qquad (Visible(S,T) \mid i) \in SeqSpec(i)\} \\
FinalStateOpaque &= \{H \in THistory \mid \exists H' \in TExtension(H) \colon \exists S \in TSequential \colon \\
&\qquad H' \equiv S \wedge \ \leq_{H'} \subseteq \leq_S \ \wedge S \in TSeqSpec\}
\end{aligned}
$$

Figure 1. $FinalStateOpaque$

2. Opacity

This section formalizes the notion of opacity [11]. The goal is to enable formal proofs of our theorems.

2.1 Execution Histories

We now recall standard definitions of execution histories [22].

Strings. If s_1 and s_2 are strings, we write $s_1 \Subset s_2$ iff s_1 is a subsequence of s_2. For example, $bd \Subset abcde$. Let s be an isogram, that is, s contains no repeating occurrence of the alphabet. For any $s_1, s_2 \Subset s$, we write $s_1 \lhd_s s_2$ iff the last element of s_1 occurs before the first element of s_2 in s. For example $ab \lhd_{abcde} de$. We use $s(i)$ to denote the i^{th} element of s.

Objects and events. Let O denote the set of objects, let n_o denote the set of methods of object o, let $Trans$ denote the set of thread (transaction) identifiers $\{T_1, \ldots, T_n\}$, and let V denote the set of values. The set of method calls is $MC = \{o.n_T(v_1, \ldots, v_n) \mid o \in O, n \in n_o, T \in Trans, v_1, \ldots, v_n \in V\}$. In a method call $mc = o.n_T(v_1, \ldots, v_n)$, we have that o is the receiver object, n is the method called, T is the identifier of the caller thread, and $v_1, \ldots, v_n$ are argument values. To differentiate repeated method calls by the same thread, it is assumed that calls from each thread have sequence numbers. Thus, the thread id and the sequence number make a unique id for each method call. In the interest of brevity, we only write the thread id in the notation and elide the sequence number. We explicitly declare the uniqueness of the method calls when it is not evident from the context. From the point of view of the caller thread, the execution of a method call has two events: an invocation event and, later, a response event. An invocation event represents the time that the caller thread initiates the call, and a response event represents the time the call returns to the caller thread. The set of invocation events is $Inv = \{inv_T(mc) \mid T \in Trans, mc \in MC\}$. The set of response events is $Res = \{ret_T(v) \mid T \in Trans, v \in V \cup \{A, C\}\}$. ($A$ and C are used later to denote abortion and commitment of transactions.) The set of events is $Ev = Inv \cup Res$. We will use the term completed method call to denote a sequence of an invocation event followed by the matching response event (with the same transaction identifier and sequence number). The set of completed method calls is $M = \{inv_T(mc), ret_T(v) \mid mc \in MC, v \in V \cup \{A, C\}\}$. We use $mc{:}v$ to denote the completed method call $inv_T(mc), ret_T(v)$. For $m \in M$, we let inv_m denote the invocation event of m, and we let ret_m denote the response event of m. The function $Tid : Ev \to Trans$ is defined as follows: $Tid(inv_T(mc)) = T$ and $Tid(ret_T(v)) = T$.

Operations on event sequences. Let E and E' be event sequences. We use $E \cdot E'$ to denote the concatenation of E and E'. For a thread T, we use $E|T$ to denote the subsequence of all events of T in E. For an object o, we use $E|o$ to denote the subsequence of all events of o in E. We write $E \equiv E'$ iff $\forall T \colon E|T = E'|T$, and say that E and E' are equivalent. We say that thread T is in E, written $T \in E$ iff $\exists j \colon Tid(E(j)) = T$. $Sequential$ is the set of sequences of completed method calls possibly followed by an invocation event.

Execution history. An execution history X is a sequence of events such that $\forall T \in X \colon X|T \in Sequential$. Let $History$ denote the set of execution histories. Note that the implicit sequence number of method calls makes the events of a history distinct from each other and hence every history is an isogram. An invocation event e issued by a transaction T is *pending* in an execution history X iff $X|T$ contains no response event that matches and follows e. For an execution history X, we let $Extension(X)$ denote the set of execution histories constructed from X by appending responses

for some pending invocations in X. We let $Complete(X)$ denote the subsequence of X that consists of all completed method calls in X.

Real-time relations. For an execution history X, we define the real-time relations $\prec_X$, $\preceq_X$, $\sim_X$, $\precsim_X$ on M as follows: First, $m_1 \prec_X m_2$ iff $ret_{m_1} \lhd_X inv_{m_2}$. $m_1 \preceq_X m_2$ iff $m_1 \prec_X m_2 \vee m_1 = m_2$. Second, $m_1 \sim_X m_2$ iff $m_1 \not\prec_X m_2 \ \wedge \ m_2 \not\prec_X m_1$. Third, $m_1 \precsim_X m_2$ iff $m_1 \prec_X m_2 \ \vee \ m_1 \sim_X m_2$.

Sequential specifications of objects. In Appendix A [26], we specify the classes of objects that we use in this paper. We let $SafeReg[V]$ denote the class of safe registers on the set of values V, $AtomicReg[V]$ denote the class of atomic registers on the set of values V, $CASReg[V]$ denote the class of CAS registers on the set of values V, $SCounter$ denote the class of strong counters, $Lock$ denote the class of locks, $TryLock$ denote the class of try-locks, $Set[V]$ denote the class of subsets of set V, $Map[K, V]$ denote the class of maps from the set of keys K to the set of values V. Note that Set and Map types provide no safety. For brevity, we write r as a syntactic sugar for $r.read$. Also $r := v$ is used as a syntactic sugar for $r.write(v)$. The sequential specification of an object o, denoted by $SeqSpec(o)$, is a set of prefix-closed, sequential histories of o that declares the set of correct sequential histories of o.

Linearizability. An execution history X is linearizable for an object o iff $\exists X' \in Extension(X|o)\colon \ \exists L \in Sequential\colon L \equiv Complete(X') \ \wedge \ L \in SeqSpec(o) \ \wedge \ \prec_{X|o} \subseteq \prec_L$, and we say that such an L is a linearization and $\prec_L$ is a linearization order of $X|o$.

2.2 Shared Memory and Transaction Histories

We now define shared memory and transaction histories.

Shared Memory. The shared memory is a singleton object mem that encapsulates the set of locations Loc where each location loc_i, $i \in I$, $I = \{1, \ldots, m\}$ stores a value $v \in V$. The object mem has three methods $read_T(i)$, $write_T(i, v)$ and $commit_T$. The method call $read_T(i)$ returns the value of loc_i or A (if the transaction is aborted). The method $write_T(i, v)$ writes v to loc_i and returns ok or returns A. The method $commit_T$ tries to commit transaction T and returns C (if the transaction is successfully committed) or returns A (if it is aborted). The object mem is implicit in method calls on mem, that is, $read_T(i)$ abbreviates $mem.read_T(i)$. For an execution history X, the subsequence $X|mem$ is the sequence of all events of X on mem.

Transaction History. A transaction history H is $Init \cdot H'$, where $Init$ is the transaction $write_{T_0}(1, v_0), \ldots, write_{T_0}(m, v_0)$, $commit_{T_0}$:C that initializes every location to v_0, and for all $T \in H'\colon H'|T$ is a prefix of $O.F$ where O is a sequence of reads $read_T(i)$:v and writes $write_T(i, v)$ (for some $i \in I$, and $v \in V$) and F is one of the following sequences: (1) $inv_T(read_T(i))$, $ret_T(A)$ (for some $i \in I$) , (2) $inv_T(write_T(i, v))$, $ret_T(A)$ (for some $i \in I$, and $v \in V$), (3) $inv_T(commit_T)$, $ret_T(C)$, or (4) $inv_T(commit_T)$, $ret_T(A)$. Let $THistory$ denote the set of transaction histories. The projection of H on i, written $H|i$, denotes the subsequence of history H that contains exactly the events on location i. For a transaction history H, we define the real-time relations $<_H$ and $\leq_H$ on $Trans$ as follows. First, $\forall T, T' \in Trans\colon \ T <_H T'$ iff $H|T \lhd_H H|T'$. Second, $T \leq_H T'$ iff $T <_H T' \vee T = T'$.

2.3 A Formal Definition of Opacity

A TM algorithm is defined to be opaque if every transaction history of every source program running on top of that TM algorithm is opaque. A history is defined to be opaque if every prefix of it is final-state-opaque. Next, we present the definition of final-state-opacity.

$FinalStateOpaque$ is defined in Figure 1. We use T prefix before some of the terms to avoid confusion with the terms that we defined above for execution histories of objects. We say that a transaction history is sequential if it is a sequence of transactions. A transaction T is committed or aborted in a transaction history H if there is respectively a commit or abort response event for T in H. A completed transaction is either committed or aborted. A live transaction is a transaction that is not completed. A transaction history is complete if all its transactions are completed. A pending transaction has a pending event and a commit-pending transaction has a commit pending event. An extension of a transaction history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions. If $H \in THistory$ and p is a predicate on transaction identifiers, we define $filter(H, p)$ to be the subsequence of H that contains those events e in H for which $p(Tid(e))$ is true. The visible history for a transaction T in a sequential transaction history S, $Visible(S, T)$, is the sequence of committed transactions before T in S and T itself. The sequential specification of a location i, $SeqSpec(i)$, is the set of sequential histories of read and write method calls on i where every read returns the value given as the argument to the latest preceding write (regardless of thread identifiers). It is essentially the sequential specification of a register. Transactional sequential specification is the set of complete sequential transaction histories S that for every transaction T and location i, $Visible(S, T)|i$ is a member of the sequential specification of i. A transaction history H is final-state-opaque if there is an equivalent sequential transaction history S that is real-time-preserving and a member of transactional sequential specification. In other words, every correct concurrent execution is indistinguishable from a correct sequential execution.

Note that opacity and linearizability are at two different levels. In fact, linearizability is the correctness condition for the base concurrent objects. TM algorithms rely on the guarantees of several of these objects to guarantee the correctness conditions for memory transactions. At the time that the paper [20] was published, no correctness criteria had been proposed for TM. Thus, that paper used the term linearizability as the correctness criterion for TM.

3. Core DSTM doesn't satisfy Opacity

We will analyze a core version of DSTM that we call Core DSTM.

The context. We believe that Core DSTM matches the *paper* on DSTM [20]. While we prove that Core DSTM doesn't satisfy opacity, we have learned from personal communication with Victor Luchangco, one of the DSTM authors, that the *implementation* of DSTM implements more than what was said in the paper and most likely satisfies opacity. We will briefly summarize the additional feature of the implementation after our analysis.

The idea. DSTM is a deferred-update TM algorithm, which means that updates are delayed until a transaction commits [17]. Each transaction maintains a log of locations and tentative values; the write operations add to the log. When a transaction commits, it updates the locations with the values stored in the log. When a transaction aborts, it discards the log.

The algorithm. Figure 2(a) shows the Core DSTM algorithm. For a detailed walk-through of the algorithm, either see the DSTM paper [20] or Appendix B in the full version of this paper [26]. Here we will merely summarize the data structures and then explain a particular execution that proves non-opacity.

The data structures. We first introduce some notation. Let $Ref[T]$ be the class of references to objects of class T. The dereference operator is denoted by !. Let Loc be the class of objects with three fields: $writer$, $oldValue$ and $newValue$. The field $writer$ is a safe register on $Trans$. The two fields $oldValue$ and $newValue$ are safe registers on V.

```
R01 :  def read_T(i)
R02 :    if (state_T = A)
R03 :      return A
R04 :    loc := !start_i
R05 :    v := stableValue_T(loc)
R06 :    if (loc.writer ≠ T)
R07 :      rset_T ⊕ (i, v)
R08 :    if (¬validate_T())
R09 :      return A
R10 :    return v
C01 :  def commit_T
C02 :    if (¬validate_T())
C03 :      return A
C04 :    if (state_T.cas(R, C))
C05 :      return C
C06 :    else
C07 :      return A
CV1 :  def stableValue_T(loc)
CV2 :    T' := loc.writer
CV3 :    if (T' ≠ T ∧ state_T' = R)
CV4 :      state_T'.cas(R, A)
CV5 :    if (state_T' = A)
CV6 :      return loc.oldValue
CV7 :    else
CV8 :      return loc.newValue
```

```
W01 :  def write_T(i, v)
W02 :    if (state_T = A)
W03 :      return A
W04 :    start := start_i
W05 :    loc := !start
W06 :    if (loc.writer = T)
W07 :      loc.newValue := v
W08 :      return ok
W09 :    v' := stableValue_T(loc)
W10 :    start' := new Loc(T, v', v)
W11 :    if (start_i.cas(start, start'))
W12 :      return ok
W13 :    else
W14 :      return A
V01 :  def validate_T()
V02 :    foreach ((i, v) ∈ rset_T)
V03 :      loc := !start_i
V04 :      T' := loc.writer
V05 :      if (state_T' = C)
V06 :        v' := loc.newValue
V07 :      else
V08 :        v' := loc.oldValue
V09 :      if (v ≠ v')
V10 :        return false
V11 :    return state_T = R
```

The required orders are enforced by the data and control dependencies.

(a) The Obstruction-free DSTM algorithm

T_1	T_2
$read_{T_1}(i_1)$:v_0	$read_{T_2}(i_1)$:v_0
$read_{T_1}(i_2)$:v_0	$read_{T_2}(i_2)$:v_0
$write_{T_1}(i_1, -v_0)$	$write_{T_2}(i_2, -v_0)$
$commit_{T_1}$.C01–C03	$commit_{T_2}$.C01–C03
$commit_{T_1}$.C04–C07	$commit_{T_2}$.C04–C07

(b) DSTM Counterexample

Figure 2. DSTM Algorithm

Core DSTM uses the following shared objects. For each $T \in Trans$, $state_T$ is a CAS register on $\{\mathbb{R}, \mathbb{A}, \mathbb{C}\}$ (that is, running, aborted or committed) with the default value $\mathbb{R}$. For each $i \in I$, $start_i$ is a CAS register on references to locator objects. The default value of each $start_i$ is a locator with $writer$ set to T_0. Each $T \in Trans$ has the transaction-local read set $rset_T$, which is a set of vectors (i, v), for $i \in I$ and $v \in V$, and is $\emptyset$ initially.

The problem. Core DSTM can produce the transaction history H_1, as we will show below:

$$\begin{aligned} H_1 \quad = \quad & Init \cdot \ read_{T_1}(i_1){:}v_0 \cdot \ read_{T_2}(i_1){:}v_0 \cdot \\ & read_{T_1}(i_2){:}v_0 \cdot \ read_{T_2}(i_2){:}v_0 \cdot \\ & write_{T_1}(i_1, -v_0) \cdot \ write_{T_2}(i_2, -v_0) \cdot \\ & inv_{T_1}(commit_{T_1}) \cdot \ inv_{T_2}(commit_{T_2}) \cdot \\ & ret_{T_1}(\mathbb{C}) \cdot \ ret_{T_2}(\mathbb{C}) \end{aligned}$$

After initialization, H_1 has four completed read operations, then two completed write operations, and finally an interleaving of two commit operations. Near the end of this section we prove that H_1 isn't opaque, which shows that Core DSTM isn't opaque. Intuitively, H_1 is not opaque because Core DSTM suffers from a write-skew anomaly: if we order T_1 before T_2, then the values read by T_2 violate opacity; and if we order T_2 before T_1, then the values read by T_1 violate opacity. Note that since H_1 is not opaque and all the transactions in H_1 are committed, H_1 is not even serializable.

Write-skew. Let us recall the essence of the write-skew anomaly. Consider a transaction that updates a set of locations and commits. To have serializability, other transactions should be prevented from observing the values of some of these locations before the update and some of the rest of the locations after the update. The property that all reads of a transaction observe a consistent snapshot is called snapshot isolation. DSTM provides snapshot isolation by validating the read set (at $R08$) before the read method returns. Snapshot isolation is a necessary but not a sufficient condition for serializability: algorithms that only support snapshot isolation are known to be prone to the write-skew anomaly, as shown by Fekete et al. [10]. A correct TM algorithm should both provide snapshot isolation and prevent the write-skew anomaly.

Let us consider H_1 more closely. Assume that a person has two bank accounts that are stored at loc_{i_1} and loc_{i_2} with the initial balances $v_0 = 10000$. Assume also that the regulations of the bank require the sum of a person's accounts to be positive or zero. Thus, a transaction that updates the value of one of the accounts with the previous value of the account minus the sum of the two accounts is authorized because the transaction makes the sum of the two accounts zero. In H_1, the initial value of i_1 is v_0, and the initial value of i_2 is v_0. The transaction T_1 attempts to update i_1 with $v_0 - (v_0 + v_0) = -v_0$. Similarly, the transaction T_2 attempts to update i_2 with $-v_0$. Both transactions are authorized, they begin execution in a consistent state, they both commit, and yet they result in an inconsistent state. This example is an instance of the general write-skew anomaly where $ReadSet(T_1) \cap WriteSet(T_2) \neq \emptyset$ and $ReadSet(T_2) \cap WriteSet(T_1) \neq \emptyset$, and both T_1 and T_2 commit.

The problematic execution. Figure 2(b) shows a concurrent execution of transactions T_1 and T_2 with Core DSTM that produces H_1. Each transaction executes from top to bottom and the horizontal lines denote barriers, that is, the operations above the line are finished before the operations below the line are started.

Both T_1 and T_2 read the initial values of i_1 and i_2 (the initial snapshot of the memory). Next, T_1 and T_2 write the new value $-v_0$ to locations i_1 and i_2 respectively. Then, each of them invokes commit and finishes the validation phase ($C01 - C03$) before the other one effectively commits (executes the cas method call at $C05$). During the validation, the two transactions still see v_0 as the stable value of the two locations; thus, both of them can pass

the validation phase. Finally, both of them succeed at cas and the result is a memory with $loc_{i_1} = -v_0$ and $loc_{i_2} = -v_0$, which is not an authorized state.

Note that the counterexample happens when the two commit method calls interleave between $C03$ and $C04$.

Theorem 1. $H_1 \notin FinalStateOpaque$.

Proof. We will prove the theorem by contradiction. Suppose $H_1 \in FinalStateOpaque$. H_1 is a complete history, thus $TExtension(H_1) = \{H_1\}$. By definition of $FinalStateOpaque$, we have that there exists a history S such that (1) $S \in TSequential$, (2) $H_1 \equiv S$, (3) $\leq_{H_1} \subseteq \leq_S$ and (4) $S \in TSeqSpec$. From the definition of H_1 above, we have that $T_0 \prec_{H_1} T_1$ and $T_0 \prec_{H_1} T_2$. Thus, from [3] we have that (5) $T_0 \prec_S T_1 \wedge T_0 \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \vee T_2 \prec_S T_1$. From [2], [5] and [6], we have that S is either of the following two histories

- Case $S = H_1|T_0 \cdot H_1|T_1 \cdot H_1|T_2$.
  We have that
  $Visible(S, T_2)|i_1 =$
  $write_{T_0}(i_1, v_0), read_{T_1}(i_1){:}v_0, write_{T_1}(i_1, -v_0),$
  $read_{T_2}(i_1){:}v_0$
  Thus, $Visible(S, T_2)|i_1 \notin SeqSpec(i_1)$. Thus, $S \notin TSeqSpec$, a contradiction to [4].
- Case $S = H_1|T_0 \cdot H_1|T_2 \cdot H_1|T_1$.
  We have that
  $Visible(S, T_1)|i_2 =$
  $write_{T_0}(i_2, v_0), read_{T_2}(i_2){:}v_0, write_{T_2}(i_2, -v_0),$
  $read_{T_1}(i_2){:}v_0$
  Thus, $Visible(S, T_1)|i_2 \notin SeqSpec(i_2)$. Thus, $S \notin TSeqSpec$, a contradiction to [4].

□

The fix. We learned from Victor Luchangco that the *implementation* of DSTM aborts the $writer$ transactions of the locations in the read set $rset_T$ during validation of the commit method call. We can model this fix by adding the following lines between $C01$ and $C02$:

foreach $(i \in dom(rset_T))$
  $loc := !start_i$
  $T' := loc.writer$
  $state_{T'}.cas(\mathbb{R}, \mathbb{A})$

Those lines prevent H_1 because each transaction will abort the other transaction and thus both of them abort.

Another fix to the algorithm is to let $R07$ store the locator reference (instead of the value) in the read set, and to change the validation for the commit procedure to the following lines:

def $validate_T()$
  foreach $((i, ref) \in rset_T)$
    $start := start_i$
    if $(start \neq ref)$
      return $false$
  return $state_T = \mathbb{R}$

Those lines prevent H_1 because both transactions observe that the locator is changed, fail the validation at $C02$ and abort.

4. Core McRT doesn't satisfy Opacity

We will analyze a core version of McRT that we call Core McRT.

The context. McRT [31] predates the definition of opacity [11] and wasn't intended to satisfy such a property, as far as we know. Rather, McRT is serializable, by design. Still, we prove that Core McRT doesn't satisfy opacity.

The idea. McRT is a direct-update TM algorithm, which means that transactions directly modify memory locations [17]. Each transaction maintains an undo-log of values that it has overwritten. If the transaction aborts, it restores the old values from the log. If the transaction commits, it discards the log.

The algorithm. Figure 3(a) shows the Core McRT algorithm. For a detailed walk-through of the algorithm, either see the McRT paper [31] or Appendix B in the full version of this paper [26]. Here we will merely summarize the data structures and then explain a particular execution that proves non-opacity.

The data structures. Core McRT uses the following shared objects. For each $i \in I$, r_i is a safe register on V, ver_i is an atomic register on $Ver = \{0, \ldots Ver_{max}\}$ that is initially 0, and l_i is a try-lock that is initially released. Core McRT uses the following transaction-local objects. For each $T \in Trans$, the read set $rset_T$ is a map from I to Ver which is $\emptyset$ initially, and the undo set $uset_T$ is a map from I to V which is $\emptyset$ initially.

In the original implementation, ver_i and l_i are stored in a single word. In our specification, we make the distinction explicit and specify the order of accesses to these registers. In addition, the original implementation overwrites the version bits with the transaction descriptor during the lock acquisition. Therefore, the versions had to be cached not only during the read method call but also during the write method call. Our specification stores only versions in the version registers and avoids caching of those registers during the write method call.

The problem. Core McRT can produce the transaction history H_2, as we will show below:

$$\begin{aligned} H_2 \quad = \quad & Init \cdot \quad read_{T_2}(i_1){:}v_0 \cdot \quad inv_{T_1}(read_{T_1}(i_2)) \cdot \\ & write_{T_2}(i_2, v_1) \cdot \quad ret_{T_1}(v_1) \cdot \quad write_{T_1}(i_1, v_1) \cdot \\ & inv_{T_1}(commit_{T_1}) \cdot \quad inv_{T_2}(commit_{T_2}) \cdot \\ & ret_{T_1}(\mathbb{A}) \cdot \quad ret_{T_2}(\mathbb{A}) \end{aligned}$$

After initialization, H_2 has a completed read operation, then an interleaving of a read operation and a completed write operation, then another completed write operation, and finally an interleaving of two commit operations. Near the end of this section we prove that H_2 isn't opaque, which shows that Core McRT isn't opaque. Intuitively, H_2 is not opaque because Core McRT suffers from the write-exposure anomaly: a written value is exposed to readers before the writer commits. Thus, active or aborting transactions can read inconsistent values, although they cannot eventually commit. For example, in H_2 location i_2 has initial value v_0 and no *committed* transaction writes a different value to i_2, and yet the invocation $inv_{T_1}(read_{T_1}(i_2))$ returns the value v_1.

The problematic execution. Figure 3(b) shows a concurrent execution of transactions T_1 and T_2 with Core McRT that produces H_2. As in the DSTM example, each transaction executes from top to bottom and the horizontal lines denote barriers, that is, the operations above the line are finished before the operations below the line are started.

The execution interleaves $write_{T_2}(i_2, v_1)$ between statements $read_{T_1}(i_2).R01 - R04$ and $read_{T_1}(i_2).R05 - R10$ such that the old value of l_2 and the new value of r_2 are read. Also, $commit_{T_2}.C01 - C04$ is executed before $commit_{T_1}.C05 - C06$ such that T_2 finds l_1 locked and aborts.

At $read_{T_1}(i_2).R01 - R04$, l_2 is read before it is locked in $write_{T_2}(i_2, v_1)$. Thus, $read_{T_1}(i_2)$ is not aborted at $R06$. At $R09$, v_1 is read from r_2 (v_1 is the value that $write_{T_2}(i_2, v_1).W08$ has written to r_2) and v_1 is returned. On the invocation of the commit, T_1 and T_2 have already called $write_{T_1}(i_1, v_1)$ and $write_{T_2}(i_2, v_1)$ and acquired locks l_1 and l_2 at $W03$ respectively. Additionally, T_1 and T_2 have already called $read_{T_1}(i_2)$ and $read_{T_2}(i_1)$ and thus $i_2 \in dom(rset_{T_1})$ and $i_1 \in dom(rset_{T_2})$. Thus, T_1 and T_2 read

$R01:$	**def** $read_T(i)$	$C01:$	**def** $commit_T()$
$R02:$	**if** $(i \notin dom(uset_T))$	$C02:$	**foreach** $((i \mapsto rver) \in rset_T)$
$R03:$	$ver := ver_i$	$C03:$	$locked := l_i$
$R04:$	$locked := l_i$	$C04:$	$ver := ver_i$
$R05:$	**if** $(locked)$	$C05:$	**if** $(locked \vee rver \neq ver)$
$R06:$	**return** $abort_T()$	$C06:$	**return** $abort_T()$
$R07:$	**if** $(i \notin dom(rset_T))$	$C07:$	**foreach** $(i \in dom(uset_T))$
$R08:$	$rset_T \oplus (i \mapsto ver)$	$C08:$	$ver_i := ver_i + 1$
$R09:$	$v := r_i$	$C09:$	$l_i.unlock$
$R10:$	**return** v	$C10:$	**return** $\mathbb{C}$
$W01:$	**def** $write_T(i, v)$	$A01:$	**def** $abort_T()$
$W02:$	**if** $(i \notin dom(uset_T))$	$A02:$	**foreach** $((i \mapsto v) \in uset_T)$
$W03:$	$locked := l_i.tryLock$	$A03:$	$r_i := v$
$W04:$	**if** $(\neg locked)$	$A04:$	$l_i.unlock()$
$W05:$	**return** $abort_T()$	$A05:$	**return** $\mathbb{A}$
$W06:$	$oldV := r_i$		
$W07:$	$uset_T \oplus (i \mapsto oldV)$		
$W08:$	$r_i := v$		
$W09:$	**return** ok		

In addition to the orders imposed by the data and control dependencies and lock synchronization, the following orders are required:
$R03 \prec_X R04,\ C03 \prec_X C04$

(a) The McRT Algorithm

T_1	T_2
$read_{T_1}(i_2).R01$–$R04$	$read_{T_2}(i_1)$
	$write_{T_2}(i_2, v_1)$
$read_{T_1}(i_2).R05$–$R10$ $write_{T_1}(i_1, v_1)$	
$commit_{T_1}.C01$–$C04$	$commit_{T_2}.C01$–$C04$
$commit_{T_1}.C05$–$C06$	$commit_{T_2}.C05$–$C06$

(b) McRT Counterexample

Figure 3. McRT Algorithm

l_2 and l_1 at $C03$ respectively. Each of the two transactions read the value of a lock that is held by the other transaction. Thus, both transactions finally abort at $C05 - C06$.

Theorem 2. *$H_2 \notin FinalStateOpaque$.*

Proof. We will prove the theorem by contradiction. Suppose $H_2 \in FinalStateOpaque$. H_2 is a complete history, thus $TExtension(H_2) = \{H_2\}$. By definition of $FinalStateOpaque$, we have that there exists a history S such that (1) $S \in TSequential$, (2) $H_2 \equiv S$, (3) $\leq_{H_2} \subseteq \leq_S$ and (4) $S \in TSeqSpec$. From the definition of H_2 above, we have that $T_0 \prec_{H_2} T_1$ and $T_0 \prec_{H_2} T_2$. Thus from [3] we have that (5) $T_0 \prec_S T_1 \wedge T_0 \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \vee T_2 \prec_S T_1$. From [2], [5] and [6], we have that S is either $H_2|T_0 \cdot H_2|T_1 \cdot H_2|T_2$ or $H_2|T_0 \cdot H_2|T_2 \cdot H_2|T_1$. In both of these cases, we have that $Visible(S, T_1)|i_2 = write_{T_0}(i_2, v_0), read_{T_1}(i_2){:}v_1$. Thus, as $v_0 \neq v_1$, we have that $Visible(S, T_1)|i_2 \notin SeqSpec(i_2)$. Thus, $S \notin TSeqSpec$, a contradiction to [4]. □

The fix. The validation in the commit method ensures that only transactions that have read consistent values can commit; this is the key to why Core McRT is serializable. A possible fix to make McRT opaque is to let also the read method do validation, that is, to insert a copy of lines $C03 - C06$ between line $R09$ and line $R10$. Note though that in the fixed algorithm, a sequence of writer transactions can make a reader transaction abort an arbitrary number of times. This observation motivated our study of progress for direct-update TM algorithms such as McRT.

5. Local Progress and Opacity

We will prove that for direct-update TM algorithms, opacity and local progress are incompatible, even for fault-free systems.

Local progress. We first recall the notion of local progress [3]. Intuitively, a TM algorithm ensures local progress if every transaction that repeatedly tries to commit eventually commits successfully. A *process* is a sequential thread that executes transactions with the same identifier. A process T is *crashing* in an infinite history H if $H|T$ is a finite sequence of operations. In other words, a process is crashing if from some point in time, it stops sending invocation events. A process T is *pending* in infinite history H if H has only a finite number of commit response $ret_T(\mathbb{C})$ events. A process *makes progress* in an infinite history, if it is not pending in it. A process T is *parasitic* in the infinite history H if $H|T$ is infinite and in history $H|T$, there are only a finite number of commit invocation $inv_T(commit)$ or abort response $ret_T(\mathbb{A})$ events. In other words, a parasitic process is a process that from some point in time keeps executing operations without being aborted and without attempting to commit. A process is *correct* in an infinite history if it is not parasitic and not crashing in the history. A process that is not correct is *faulty*. An infinite history satisfies *local progress*, if every correct process in it makes progress. A TM algorithm ensures *local progress*, if every infinite history of it satisfies local progress and every finite history of it can be extended to an infinite history of it that satisfies local progress. A system is *fault-prone* if at least one process can be crashing or parasitic. A system is *fault-free* if it is not *fault-prone*.

The seminal result. Theorem 3 is the seminal result on the incompatibility of opacity and local progress.

Theorem 3. (Bushkov, Guerraoui, and Kapalka [3]) *For a fault-prone system, no TM algorithm ensures both opacity and local progress.*

Considering a fault-prone system, the proof uses strategies that result in either a crashing or parasitic process.

Fault-prone versus fault-free. The large class of fault-prone systems presents a formidable challenge for designers of TM algorithms who want some form of progress. A crashing or parasitic process may never relinquish the ownership of a resource that another process must acquire before it can make progress. Bushkov, Guerraoui, and Kapalka [3] consider a liveness property called *solo progress* that guarantees that a process that eventually runs alone will make progress. They conjecture that obstruction-free TM algorithms (as defined in [20]) ensure solo progress in parasitic-free

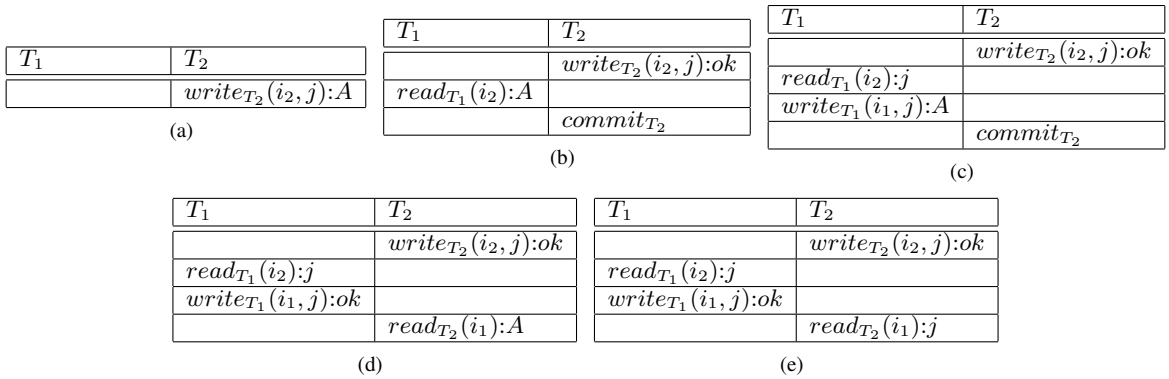

Figure 4. Impossibility of Opacity and Local-progress for Direct-update TM Algorithms

systems, and that lock-based TM algorithms ensure solo progress in systems that are both parasitic-free and crash-free. Those conjectures embody the following idea and practical advice.

> **Bushkov, Guerraoui, and Kapalka's advice [3]:** If designers of TM algorithms want opacity and progress, they must consider either weaker progress properties or fault-free systems.

TM algorithms for fault-free systems can rely on that no processes are crashing or parasitic.

Local progress for fault-free systems. Following the advice embodied in the paper by Bushkov, Guerraoui, and Kapalka [3], we study liveness in the setting of fault-free systems. Our main result is that an entire class of TM algorithms cannot ensure both opacity and local progress for fault-free systems.

We need two definitions before we can state our result formally. A TM algorithm is a *deferred-update* algorithm if every transaction that writes a value must commit before other transactions can read that value. All other TM algorithms are *direct-update* algorithms. For example, DSTM is a deferred-update algorithm while McRT is a direct-update algorithm.

Our main result is Theorem 4 which says that direct-update TM algorithms cannot ensure both opacity and local progress for fault-free systems. Thus we can refine Bushkov, Guerraoui, and Kapalka's advice.

> **Our advice:** If designers of TM algorithms want opacity and *local progress*, they might have success with *deferred-update* TM algorithms that work for fault-free systems.

The proof of Theorem 4 is different from the proof of Theorem 3 because the proof of Theorem 4 cannot use crashing or parasitic processes.

Theorem 4. *For a fault-free system, no direct-update TM algorithm ensures both opacity and local progress.*

Proof. Assume otherwise, that is, there is a direct-update algorithm that ensures opacity and local progress. We exhibit a winning strategy for the environment that acts as an adversary to the algorithm and results in either a non-opaque history or an infinite history which does not satisfy local progress. The strategy is as follows. The client iteratively executes the following sequence of operations. Iteration number j is as follows

1. Invoke $write_{T_2}(i_2, j)$.
   If the response is $\mathbb{A}$,
   leave this iteration and start the next iteration
   Otherwise,
   go to the next step.
2. Invoke $read_{T_1}(i_2)$.
   If the response is $\mathbb{A}$,
   invoke $commit_{T_2}$ and regardless of the response,
   leave this iteration and start the next iteration.
   Otherwise,
   go to the next step.
3. Invoke $write_{T_1}(i_1, j)$.
   If the response is $\mathbb{A}$,
   invoke $commit_{T_2}$ and regardless of the response,
   leave this iteration and start the next iteration.
   Otherwise,
   go to the next step.
4. Invoke $read_{T_2}(i_1)$.
   Regardless of the response, stop iterating.

In each iteration, the algorithm results in one of the executions depicted in Figure 4. Note that as the algorithm ensures local progress, by definition, every finite history of it can be extended to an infinite history of it. Thus, the three operations of the algorithm should be obstruction-free. Thus, as the operations of the above strategy are called without interleaving from other operations, every operation should return. Also note that as the algorithm is direct-update, the reads return the newly written value j. We consider two cases:

- The execution stops:
  We consider two subcases:
  - The last iteration results in the execution depicted in Figure 4(d):
    By Lemma 1, the history doesn't satisfy opacity, a contradiction.
  - The last iteration results in the execution depicted in Figure 4(e):
    By Lemma 2, the history doesn't satisfy opacity, a contradiction.
- The execution does not stop:
  The repeated iterations are depicted in Figure 4(a)-(c). We consider two subcases:
  - From some point in time, only Figure 4(a) is repeated:
    T_2 has an infinite number of operations, thus is not crashing. T_2 gets an infinite number of abort response events, thus is not parasitic. Therefore T_2 is a correct process. But T_2 does not receive an infinite number of commit response events

thus does not make progress. Therefore, the history does not satisfy local progress, a contradiction.
Note that in this case, T_1 has a finite number of operations. As mentioned above, if we assume that empty transactions have no effect on the responses from the algorithm, we can construct a similar history where after each iteration of $write_{T_2}(i_2, j)$, the commit operation $commit_{T_1}$ is executed that commits an empty transaction. The constructed history is indistinguishable for T_2 and T_1 is a correct process in the history.

- Figure 4(b) or (c) happen infinitely often in the history:
  T_1 has an infinite number of operations, thus is not crashing. T_1 gets an infinite number of abort response events, thus is not parasitic. Therefore T_1 is a correct process. But T_1 does not receive an infinite number of commit response events thus does not make progress. Therefore, the history does not satisfy local progress, a contradiction.

□

Lemma 1. *$H_3 \notin FinalStateOpaque$ where*

$$H_3 = H_0 \cdot write_{T_2}(i_2, j) \cdot read_{T_1}(i_2){:}j \cdot write_{T_1}(i_1, j) \cdot read_{T_2}(i_1){:}\mathbb{A}$$

and H_0 is a history that does not contain a write operation that writes value j.

Proof. We prove the lemma based on the following idea. To justify the read of value j by T_1, a committed transaction should have written the value j. The only transaction that writes value j is T_2 but it is aborted.

We will prove the theorem by contradiction. Suppose $H_3 \in FinalStateOpaque$. $TExtension(H_3) = \{H_3', H_3''\}$ where $H_3' = H_3 \cdot commit_{T_1}{:}\mathbb{A}$ and $H_3'' = H_3 \cdot commit_{T_1}{:}\mathbb{C}$. By definition of $FinalStateOpaque$, we have that there exists $H \in TExtension(H_3)$ such that there exists a history S such that (1) $S \in TSequential$, (2) $H \equiv S$, (3) $\leq_H \subseteq \leq_S$ and (4) $S \in TSeqSpec$. We consider two cases:

- Case $H = H_3'$.
  From the definition of H_3' and H_3, we have $\forall T \in H_0 \colon T \prec_{H_3'} T_1 \wedge T \prec_{H_3'} T_2$. Thus, by [3], we have (5) $\forall T \in H_0 \colon T \prec_S T_1 \wedge T \prec_S T_2$. From [1], we have that (6) $T_1 \prec_S T_2 \vee T_2 \prec_S T_1$. Thus, from [5], [6] and [2], we have that S is either of the following two histories: $S = S_0 \cdot H_3'|T_2 \cdot H_3'|T_1$ or $S = S_0 \cdot H_3'|T_1 \cdot H_3'|T_2$ where S_0 is a serialization of H_0. For both of these histories, we have that $Visible(S, T_1)|i_2 = S_0|i_2 \cdot read_{T_1}(i_2){:}j$ where no transaction in $S_0|i_2$ writes value j. Thus, $Visible(S, T_1)|i_2 \notin SeqSpec(i_2)$. Thus, $S \notin TSeqSpec$, a contradiction to [4].
- Case $H = H_3''$.
  Similar to the previous case, we will have that $Visible(S, T_1)|i_2 = S_0|i_2 \cdot read_{T_1}(i_2){:}j$.

□

Lemma 2. *$H_4 \notin FinalStateOpaque$ where*

$$H_4 = H_0 \cdot write_{T_2}(i_2, j) \cdot read_{T_1}(i_2){:}j \cdot write_{T_1}(i_1, j) \cdot read_{T_2}(i_1){:}j$$

and H_0 is a history that does not contain a write operation that writes value j.

Proof. We prove the lemma based on the following idea. The two transactions T_1, T_2 have read value j from i_2 and i_1 respectively. The only transaction that writes value j to i_2 and i_1 is T_2 and T_1 respectively. Thus, to justify the two read operations, each of the transactions should have been ordered before the other one in the justifying sequential history, which is impossible.

We will prove the theorem by contradiction. Suppose $H_4 \in FinalStateOpaque. TExtension(H_4) = \{H_4', H_4'', H_4''', H_4''''\}$ where
$H_4' = H_4 \cdot commit_{T_1}{:}\mathbb{C} \cdot commit_{T_2}{:}\mathbb{C}$,
$H_4'' = H_4 \cdot commit_{T_1}{:}\mathbb{C} \cdot commit_{T_2}{:}\mathbb{A}$,
$H_4''' = H_4 \cdot commit_{T_1}{:}\mathbb{A} \cdot commit_{T_2}{:}\mathbb{C}$,
$H_4'''' = H_4 \cdot commit_{T_1}{:}\mathbb{A} \cdot commit_{T_2}{:}\mathbb{A}$.
By definition of $FinalStateOpaque$, we have that there exists $H \in TExtension(H_4)$ such that there exists a history S such that (1) $S \in TSequential$, (2) $H \equiv S$, (3) $\leq_H \subseteq \leq_S$ and (4) $S \in TSeqSpec$. We consider four cases:

- Case $H = H_4'$.
  Similar to Lemma 1, we have that S is either of the following two histories: $S_1 = S_0 \cdot H_4'|T_1 \cdot H_4'|T_2$ or $S_2 = S_0 \cdot H_4'|T_2 \cdot H_4'|T_1$ where S_0 is a serialization of H_0. We consider two cases
  - $S = S_1$:
    We have that $Visible(S, T_1)|i_2 = S_0|i_2 \cdot read_{T_1}(i_2){:}j$. where no transaction in $S_0|i_2$ writes value j.
    Thus, $Visible(S, T_1)|i_2 \notin SeqSpec(i_2)$.
    Thus, $S \notin TSeqSpec$, a contradiction to [4].
  - $S = S_2$:
    We have that $Visible(S, T_2)|i_1 = S_0|i_1 \cdot read_{T_2}(i_1){:}j$. where no transaction in $S_0|i_1$ writes value j.
    Thus, $Visible(S, T_2)|i_1 \notin SeqSpec(i_1)$.
    Thus, $S \notin TSeqSpec$, a contradiction to [4].
- Case $H = H_4''$.
  Similar to the previous case, we will have that $Visible(S, T_1)|i_2 = S_0|i_2 \cdot read_{T_1}(i_2){:}j$.
- Case $H = H_4'''$.
  Similar to the previous case, we will have that $Visible(S, T_2)|i_1 = S_0|i_1 \cdot read_{T_2}(i_1){:}j$.
- Case $H = H_4''''$.
  Similar to the previous case, we will have that $Visible(S, T_2)|i_1 = S_0|i_1 \cdot read_{T_2}(i_1){:}j$.

□

6. Conclusion

We have identified two problems that lead to non-opacity and we have proved an impossibility result. We hope that our observations can help TM algorithm designers to avoid the write-skew and write-exposure pitfalls, and to be aware that if local progress is a goal, then deferred-update algorithms may be the only option.

Our proofs of non-opacity for Core DSTM and Core McRT show that care has to be taken when defining abstractions of TM algorithms. Even if an algorithm satisfies opacity at a high level of abstraction, it may fail to satisfy opacity at a lower level of abstraction.

References

[1] M. Abadi, A. Birrell, T. Harris, and M. Isard. Semantics of transactional memory and automatic mutual exclusion. In *POPL*, pages 63–74, 2008.

[2] C. Scott Ananian, Krste Asanovic, Bradley C. Kuszmaul, Charles E. Leiserson, and Sean Lie. Unbounded transactional memory. In *HPCA*, 2005.

[3] Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 9–18, New York, NY, USA, 2012. ACM.

[4] Ariel Cohen, John W. O'Leary, Amir Pnueli, Mark R. Tuttle, and Lenore D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, 2007.

[5] Ariel Cohen, Amir Pnueli, and Lenore D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV*, 2008.

[6] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC, (LNCS 4167)*, 2006.

[7] Dave Dice and Nir Shavit. TLRW: Return of the read-write lock. In *SPAA*, 2010.

[8] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.

[9] Michael Emmi, Rupak Majumdar, and Roman Manevich. Parameterized verification of transactional memories. In *Proceedings of PLDI'10, ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 134–145, June 2010.

[10] Alan Fekete, Dimitrios Liarokapis, Elizabeth O'Neil, Patrick O'Neil, and Dennis Shasha. Making snapshot isolation serializable. *ACM Trans. Database Syst.*, 30(2), June 2005.

[11] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[12] Rachid Guerraoui, Thomas A. Henzinger, Barbara Jobstmann, and Vasu Singh. Model checking transactional memories. In *ACM SIGPLAN Conference on Programming Languages Design and Implementation*, pages 372–382, 2008.

[13] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Software transactional memory on relaxed memory models. In *Proceedings of CAV'09, Seventh International Conference on Computer Aided Verification*, pages 321–336, 2009.

[14] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Model checking transactional memories. *Distributed Computing*, 2010.

[15] Rachid Guerraoui and Michal Kapalka. *Principles of Transactional Memory*. Morgan and Claypool Publishers, 2010.

[16] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.

[17] Tim Harris, James Larus, and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool Publishers, second edition, 2010.

[18] Tim Harris, Simon Marlow, Simon Peyton, and Jones Maurice Herlihy. Composable memory transactions. In *PPOPP'05*, pages 48–60. ACM Press, 2005.

[19] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[20] M. Herlihy, V. Luchangco, M. Moir, and III W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.

[21] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, pages 289–300, 1993.

[22] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[23] Damien Imbs, José Ramon de Mendivil, and Michel Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 280–281, New York, NY, USA, 2009. ACM.

[24] E. Koskinen, M. Parkinson, and M. Herlihy. Coarse-grained transactions. In *POPL*, pages 19–30, 2010.

[25] Mohsen Lesani, Victor Luchangco, and Mark Moir. A framework for formally verifying software transactional memory algorithms. In *CONCUR*, 2012.

[26] Mohsen Lesani and Jens Palsberg. Non-opacity (technical report with the appendices). www.cs.ucla.edu/ lesani/downloads/submission/transact/Transact.pdf.

[27] K. F. Moore and D. Grossman. High-level small-step operational semantics for transactions. In *POPL*, pages 51–62, 2008.

[28] V. Pankratius, A.-R. Adl-Tabatabai, , and F. Otto. Does transactional memory keep its promises? results from an empirical study. Technical Report 2009–12, Institute for Program Structures and Data Organization (IPD), University of Karlsruhe, September 2009.

[29] Christos H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[30] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? *SIGPLAN Notices*, 45(5), January 2010.

[31] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.

[32] M. L. Scott. Sequential specification of transactional memory semantics. In *TRANSACT*, 2006.

[33] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, 1995.

[34] Serdar Tasiran. A compositional method for verifying software transactional memory implementations. Technical Report MSR-TR-2008-56, Microsoft Research, apr 2008.