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, 32] 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 [27, 29]. Researchers have developed formal semantics [1, 23, 26] and a wide variety of implementations of the TM interface in both software [6, 7, 19, 20, 30] 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 [28]. For TM algorithms, strict serializability [31] 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 TMS1 [8] and opacity [11]. 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 [33] 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. [24] 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 liveliness 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.
We now recall standard definitions of execution histories [22]. This section formalizes the notion of opacity [11]. The goal is to enable formal proofs of our theorems.

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 $s_1 \subseteq s_2$ if $s_1$ is a subsequence of $s_2$. For example, $bd \subseteq abde$. Let $s$ be an isogram, that is, $s$ contains no repeating occurrence of the alphabet. For any $s_1, s_2 \subseteq s$, we write $s_1 \preceq s_2$ iff the last element of $s_1$ occurs before the first element of $s_2$ in $s$. For example $ab \preceq abde$. 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 $MC = \{o.m(v_1, \ldots, v_n) | o \in O, n \in n_o, T \in Trans, v_1, \ldots, v_n \in V\}$. In a method call $mc = o.m(v_1, \ldots, v_n)$, 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 $Inv = \{inv(mc) | T \in Trans, mc \in MC\}$. The set of response events is $Res = \{ret_T(v) | T \in Trans, v \in V \cup \{A, C\}\}$. (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 $M = \{inv(mc), ret_T(v) | mc \in MC, v \in V \cup \{A, C\}\}$. We use $mcv$ to denote the completed method call $inv(mc), ret_T(v)$. 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 $Tid : Ev \rightarrow Trans$ 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 $E \equiv E'$ if $\forall T : E[T] = E'[T]$, and say that $E$ and $E'$ are equivalent. We say that thread $T$ is in $E$, written $T \in E$ if $\exists j : Tid(E(j)) = T$. $Sequential$ 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 $\forall T \in X : X[T] \in Sequential$. 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 is issued by a transaction $T$ that is $pending$ in an execution history $X$ if $X[T]$ 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 $\prec_X$, $\preceq_X$, $\sim_X$, $\succeq_X$ on M as follows: First, $m_1 \prec_X m_2$ iff $r \in \text{read}(m_1)$ and $r \in \text{write}(m_2)$. Second, $m_1 \sim_X m_2$ iff $m_1 \not\prec_X m_2 \land m_2 \not\prec_X m_1$. Third, $m_1 \preceq_X m_2$ iff $m_1 \prec_X m_2 \lor m_1 \sim_X m_2$.

Sequential specifications of objects. In Appendix A [25], 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 trylocks, 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 $r \text{.read}$. Also $r := v$ is used as a transaction sugar for $r \text{.write(v)}$. 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 $\exists X' \in \text{Extension}(X[o]) \land \forall L \in \text{Sequential}: L \equiv \text{Complete}(X'[o]) \land L \in \text{SeqSpec}(o) \land \prec_X \subseteq \prec_L$, 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, $i \in I, I = \{1, \ldots, m\}$ stores a value $v \in V$. The object mem has three methods read(i), write(i, v) and commit(i). The method call read(i) returns the value of loc, or $A$ (if the transaction is aborted). The method write(i, v) writes v to loc, and returns ok or returns $A$. The method commit(i) 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(i) abbreviates mem.read(i). For an execution history X, the subsequence $X'|\text{mem}$ is the sequence of all events of X on mem.

Transaction History. A transaction history H is Init - $H'$, where Init is the transaction write(0, v0), write(0, v1), ..., write(0, vm), commit0,C that initializes every location to normalized, or $A$ (if the transaction is aborted). The method write(i, v) writes v, and for all $A \subseteq X' \in \text{Extension}(X[o])$.

2.3 A Formal Definition of Opacity

Opacity of a TM algorithm is defined in two steps. First, it is defined what it means for a transaction history to be opaque which is called final-state-opacity. Then, a TM algorithm is defined to be opaque if every transaction history of every source program running on top of that TM algorithm is final-state- opaque. 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 $H \in T\text{History}$ 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, $\text{Visible}(S, T)$, is the sequence of committed transactions before T in S and T itself.

The shared 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 $H$ that for every transaction T and location i, $\text{Visible}(S, T)[i]$ 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 $H'$ 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 Luchango, 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 [25]. Here we merely summarize the data structures and then explain a particular execution that proves non-opacity.

The data structures. We first introduce some notation. Let $\text{Ref}[T]$ be the class of references to objects of class T. The dereference operation 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.
Core DSTM uses the following shared objects. For each \( T \in \text{Trans} \), \( \text{state}_T \) is a CAS register on \( \{R, A, C\} \) (that is, running, aborted or committed) with the default value \( R \). For each \( i \in I \), \( \text{start}_i \) is a CAS register on references to locator objects. The default value of each \( \text{start}_i \) is a locator with \( \text{writer} \) set to \( T_0 \). Each \( T \in \text{Trans} \) 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:

\[
H_1 = \text{Init} \cdot \text{read}_{T_1}(i_1):v_0 \cdot \text{read}_{T_2}(i_1):v_0 \cdot \text{write}_{T_1}(i_1, -v_0) \cdot \text{write}_{T_2}(i_2, -v_0) \cdot \text{inv}_{T_1}(\text{commit}_{T_1}) \cdot \text{inv}_{T_2}(\text{commit}_{T_2}) \cdot \text{ret}_{T_1}(C) \cdot \text{ret}_{T_2}(C)
\]

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.

![Figure 2. DSTM Algorithm](image-url)

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 \( loc_1 \) and \( loc_2 \) 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 \( \text{ReadSet}(T_1) \cap \text{WriteSet}(T_2) \neq \emptyset \) and \( \text{ReadSet}(T_2) \cap \text{WriteSet}(T_1) \neq \emptyset \), 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 \( \text{cas} \) 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 \textit{cas} and the result is a memory with $\text{loc}_{i_1} = -v_0$ and $\text{loc}_{i_2} = -v_0$, which is not an authorized state.

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

**Theorem 1.** $H_1 \not\in \text{FinalStateOpaque}$.

**Proof.** We will prove the theorem by contradiction. Suppose $H_1 \in \text{FinalStateOpaque}$. $H_1$ is a complete history, thus $\text{Extension}(H_1) = \{H_1\}$. By definition of $\text{FinalStateOpaque}$, we have that there exists a history $S$ such that (1) $S \in \text{TS.Sequential}$, (2) $H_1 \equiv S$, (3) $\prec \in S$ and (4) $S \in \text{TSeqSpec}$. 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 $(5) T_0 \prec S T_1 \land T_0 \prec S T_2$. From [1], we have that $(6) T_1 \prec S T_2 \lor T_2 \prec S T_1$. 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 $\text{Visible}(S, T_2)\{i_1 = \text{write}_{T_1}(i_1; v_0), \text{read}_{T_1}(i_1; v_0), \text{write}_{T_1}(i_1; -v_0), \text{read}_{T_1}(i_1; v_0)\}$.

- Thus, $\text{Visible}(S, T_2)\{i_1 \notin \text{SeqSpec}(i_1)\}$. Thus, $S \notin \text{TSeqSpec}$, a contradiction to [4].

- Case $S = H_1 | T_0 \cdot H_1 | T_1 \cdot H_1 | T_1$.
  
  We have that $\text{Visible}(S, T_1)\{i_2 = \text{write}_{T_2}(i_2; v_0), \text{read}_{T_2}(i_2; v_0), \text{write}_{T_2}(i_2; -v_0), \text{read}_{T_2}(i_2; v_0)\}$.

- Thus, $\text{Visible}(S, T_1)\{i_2 \notin \text{SeqSpec}(i_2)\}$. Thus, $S \notin \text{TSeqSpec}$, a contradiction to [4].

\[\Box\]

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

```
foreach (i ∈ dom(\text{rset}_T))
    loc := lstart,
    T := loc.writer stateT := \text{cas}(\text{R}, \text{K})
```

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 \text{validate}_T()
    foreach ((i, ref) ∈ \text{rset}_T)
        start := start,
        if (start \neq ref)
            return false
    return \text{state}_T = \text{K}
```

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 [30] 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 [30] or Appendix B in the full version of this paper [25]. 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 $V_\text{er}$, $\text{ver}_i$ is an atomic register on $V_\text{er} = \{0, \ldots, V_\text{er.max}\}$ 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 $T \in \text{Trans}$, the read set $\text{rset}_T$ is a map from $T$ to $V_\text{er}$ which is $0$ initially, and the undo set $\text{uset}_T$ is a map from $T$ to $V_\text{er}$ which is $0$ initially.

In the original implementation, $\text{ver}_i$ 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:

```
H_2 = \text{Init} \cdot \text{read}_{T_2}(i_1; v_0) \cdot \text{inv}_{T_1}(\text{read}_{T_1}(i_2)) \cdot \text{write}_{T_2}(i_2; v_1) \cdot \text{ref}_{T_1}(i_1) \cdot \text{write}_{T_1}(i_1; v_1) \cdot \text{inv}_{T_1}(\text{commit}_{T_1}) \cdot \text{inv}_{T_2}(\text{commit}_{T_2}) \cdot \text{ref}_{T_1}(\text{K}) \cdot \text{ref}_{T_2}(\text{K})
```

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 $\text{inv}_{T_2}(\text{read}_{T_2}(i_2))$ 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 $\text{write}_{T_2}(i_2; v_1)$ between statements $\text{read}_{T_1}(i_2).R01 \rightarrow R04$ and $\text{read}_{T_1}(i_2).R05 \rightarrow R10$ such that the old value of $i_2$ and the new value of $r_2$ are read. Also, $\text{commit}_{T_1}.C01 \rightarrow C04$ is executed before $\text{commit}_{T_1}.C05 \rightarrow C06$ such that $T_2$ finds $l_i$ locked and aborts.

At $\text{read}_{T_1}(i_2).R01 \rightarrow R04$, $i_2$ is read before it is locked in $\text{write}_{T_2}(i_2; v_1)$. Thus, $\text{read}_{T_1}(i_2)$ is not aborted at R06. At R09, $v_3$ is read from $r_2$ ($v_3$ is the value that $\text{write}_{T_2}(i_2; v_1).W08$ has written to $r_2$) and $v_1$ is returned. On the invocation of the commit, $T_1$ and $T_2$ have already called $\text{write}_{T_1}(i_1; v_1)$ and $\text{write}_{T_1}(i_1; v_1)$ and acquired locks $l_i$ and $l_j$ at W03 respectively. Additionally, $T_1$ and $T_2$ have already called $\text{read}_{T_1}(i_2)$ and $\text{read}_{T_1}(i_1)$ and thus $i_2 \in \text{dom}(\text{rset}_{T_1})$ and $i_1 \in \text{dom}(\text{rset}_{T_2})$. Thus, $T_1$ and $T_2$ read
From [1], we have that (6)
definition of $H$.

We will prove the theorem by contradiction. Suppose

With the same identifier. A process $T$ is crashing in an infinite history $H$ if $H[\cdot]$ 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$\tau_T(\cdot)$ 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 $H[\cdot]$ is infinite and in history $H[\cdot]$, there are only a finite number of commit invocation inv$\tau_T(\cdot)$ or abort response ret$\tau_T(\cdot)$ events.

In other words, a parasitic process is a process that from some point in time in 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 that it 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.

Theorem 2. $H_2 \not\in FinalStateOpaque$.

Proof. We will prove the theorem by contradiction. Suppose $H_2 \in FinalStateOpaque$. $H_2$ is a complete history, thus $T_Extension(H_2) = \{H_2\}$. By definition of FinalStateOpaque, we have that there exists a history $S$ such that (1) $S \in T.Sequential$, (2) $H_2 \equiv S$, (3) $S \leq T_\Sigma S$ and (4) $S \in TSeqSpec$. From the definition of $H_2$ above, we have that $T_0 \prec_{X} T_1$ and $T_0 \prec_{X} T_2$. Thus we have that (5) $T_0 \prec_{X} T_0 \land T_0 \prec_{X} T_2$. From [1], we have that (6) $T_0 \prec_{X} T_2 \lor T_0 \prec_{X} T_1$. From [2], [5] and [6], we have that $S$ is either $H_2[T_0 \cdot H_2[T_1 \cdot H_2[T_2 \cdot H_2[T_1 \cdot H_2[T_2]]]]$. In both of these cases, we have that $Visible(S,T_1)|\Sigma = write_{T_0}(i_{2}, v_{0}), return(i_{2})(\Sigma)$. Thus, as $v_{0} \neq v_{1}$, we have that $Visible(S,T_1)|\Sigma \notin TSeqSpec$. 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 $H[\cdot]$ 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$\tau_T(\cdot)$ 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 $H[\cdot]$ is infinite and in history $H[\cdot]$, there are only a finite number of commit invocation inv$\tau_T(\cdot)$ or abort response ret$\tau_T(\cdot)$ 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 that it 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.

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 liveliness 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 systems.

(a) The McRT Algorithm

In addition to the orders imposed by the data and control dependencies and lock synchronizations, the following orders are required:

$R03 \prec_{X} R04, \ C03 \prec_{X} C04$

(b) McRT Counterexample
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 $\text{write}_{T_2}(i_2,j):A$.
   - If the response is $\text{ok}$,
     - leave this iteration and start the next iteration.
   - Otherwise, go to the next step.
2. Invoke $\text{read}_{T_1}(i_2)$.
   - If the response is $\text{ok}$,
     - invoke $\text{commit}_{T_2}$ and regardless of the response, leave this iteration and start the next iteration.
   - Otherwise, go to the next step.
3. Invoke $\text{write}_{T_1}(i_1,j)$.
   - If the response is $\text{ok}$,
     - invoke $\text{commit}_{T_2}$ and regardless of the response, leave this iteration and start the next iteration.
   - Otherwise, go to the next step.
4. Invoke $\text{read}_{T_2}(i_1)$.
   - Regardless of the response, stop iterating.

In each iteration, the algorithm results in one of the executions depicted in Figure 4. 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 $\text{write}_{T_2}(i_2,j)$, the commit operation $\text{commit}_{T_1}$ is ex-
Lemma 1. \( H_3 \notin \text{FinalStateOpaque} \) where
\[
H_3 = H_0 \cdot \text{write}_{T_2}(i_2, j) \cdot \text{read}_{T_1}(i_2, j) \cdot \text{write}_{T_1}(i_1, j) \cdot \text{read}_{T_1}(i_1, j) \cdot H_4
\]
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 \( H_3 \in \text{FinalStateOpaque}. T \text{Extension}(H_3) = \{H'_3, H''_3\} \) where \( H'_3 = H_3 \cdot \text{commit}_{T_1} \cdot H_4 \) and \( H''_3 = H_3 \cdot \text{commit}_{T_2} \cdot \text{commit}_{T_1} \cdot H_4 \). By definition of \( \text{FinalStateOpaque} \), we have that there exists \( H \in T \text{Extension}(H_3) \) such that there exists a history \( S \) such that (1) \( S \in T \text{Sequential} \), (2) \( H \equiv S \), (3) \( \leq_H \subseteq \leq_S \) and (4) \( S \in T \text{SeqSpec} \). We consider two cases:

- Case \( H = H'_3 \).
  From the definition of \( H'_3 \) and \( H_3 \), we have \( \forall T \in H_0 : T \prec_H' T_1 \wedge T \prec_H T_2 \). Thus, by [3], we have (5) \( \forall T \in H_0 : T \prec_S T_1 \wedge T \prec_S T_2 \). From [1], we have that (6) \( T_1 \prec_S T_1 \vee T_2 \prec_S T_1 \). 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 \( \text{Visible}(S, T_1)[i_2] = S_0[j_2] \cdot \text{read}_{T_2}(i_2, j) \) where no transaction in \( S_0[i_2] \) writes value \( j \). Thus, \( \text{Visible}(S, T_1)[i_2] \notin \text{SeqSpec}(i_2) \). Thus, \( S \notin T \text{SeqSpec} \), a contradiction to [4].

- Case \( H = H''_3 \).
  Similar to the previous case, we will have that \( \text{Visible}(S, T_1)[i_2] = S_0[i_2] \cdot \text{read}_{T_2}(i_2, j) \).

\( \square \)

Lemma 2. \( H_4 \notin \text{FinalStateOpaque} \) where
\[
H_4 = H_0 \cdot \text{write}_{T_2}(i_2, j) \cdot \text{read}_{T_1}(i_2, j) \cdot \text{write}_{T_1}(i_1, j) \cdot \text{read}_{T_1}(i_1, j)
\]
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 \( H_4 \in \text{FinalStateOpaque}. T \text{Extension}(H_4) = \{H'_4, H''_4, H'''_4\} \) where \( H'_4 = H_4 \cdot \text{commit}_{T_1} \cdot \text{commit}_{T_2} \). We consider two 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 \( S_2 = S_0 \cdot H''_4[T_1] \cdot H''_4[T_2] \) where \( S_0 \) is a serialization of \( H_0 \). We consider two cases:

  - Case \( S = S'_1 \).
    We have that \( \text{Visible}(S, T_1)[i_4] = S_0[i_2] \cdot \text{read}_{T_1}(i_2, j) \) where no transaction in \( S_0[i_2] \) writes value \( j \). Thus, \( \text{Visible}(S, T_1)[i_2] \notin \text{SeqSpec}(i_2) \). Thus, \( S \notin T \text{SeqSpec} \), a contradiction to [4].

- Case \( S = S'_2 \).
  We have that \( \text{Visible}(S, T_2)[i_1] = S_0[i_1] \cdot \text{read}_{T_2}(i_1, j) \) where no transaction in \( S_0[i_1] \) writes value \( j \). Thus, \( \text{Visible}(S, T_2)[i_1] \notin \text{SeqSpec}(i_1) \). Thus, \( S \notin T \text{SeqSpec} \), a contradiction to [4].

\( \square \)

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