# Decomposing Opacity

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

**Abstract.** Transactional memory (TM) algorithms are subtle and the TM correctness conditions are intricate. Decomposition of the correctness condition can bring modularity to TM algorithm design and verification. We present a decomposition of opacity called markability as a conjunction of separate intuitive invariants. We prove the equivalence of opacity and markability. The proofs of markability of TM algorithms can be aided by and mirror the algorithm design intuitions. As an example, we prove the markability and hence opacity of the TL2 algorithm. In addition, based on one of the invariants, we present lower bound results for the time complexity of TM algorithms.

## 1 Introduction

A transactional memory (TM) [24, 36] is a concurrent object that encapsulates and manages accesses to an array of memory locations. The clients of a TM are transactions, sequences of accesses to the encapsulated locations. A transactional processing system is the composition of a TM and a set of client transactions. While the clients issue the invocation events, the TM issues the response events. Researchers have proposed several TM correctness conditions including opacity [20], VWC [25], TMS1 and TMS2 [13], and DU-opacity [2] that characterize the required safety conditions on TM response events.

Considering the strength of the promised safety properties, designing a correct TM is an art. TM algorithms whether in software [9, 11, 12, 15, 23, 35], hardware [1, 7, 22, 37] or hybrid [8, 10, 26, 31, 32] are subtle and prone to bugs [30]. Thus, verification of TM algorithms by model checking [4–6, 16–18, 33], invariant generation [14] and theorem proving [28] has been a topic of recent attention. Verifying a complicated monolithic condition for a realistic specification of a TM algorithm can be a formidable problem. Can the correctness condition of TM be stated as a conjunction of simpler intuitive conditions? In other words, is there a meaningful decomposition of the correctness condition? What are the separate invariants that the TM designers should maintain? Decomposition of the correctness condition enhances the understanding of the correctness and brings modularity to the algorithm design. It showcases different aspects of correctness and helps designers concentrate on maintaining one aspect at a time. More importantly, separation has obvious benefits of modularity and scalability for verification. Furthermore, it supports studying the time complexity and performance of TM algorithms.

We decompose opacity to separate intuitive invariants. We define that an execution history is markable if there is a specific ordering relation on the set of transactions and read operations called marking such that three invariants are satisfied. We prove that markability is required and sufficient for opacity. At a high level, the first invariant called write-observation requires that each read operation returns the most current value. The second invariant called read-preservation requires that the read location is not overwritten in the interval that the location is read and the transaction takes effect. The third invariant is the well-known real-time-preservation property. We show that the marking relation for the TL2 algorithm [11] can be defined using the execution order and the linearization order of method calls on the used synchronization objects and proofs of markability can be aided by and mirror the algorithm design intuitions. We prove markability and hence opacity of TL2. Finally, inspired by the read-preservation invariant, we present lower bound results for the time complexity of a class of TM algorithms.

In the following sections, we first introduce the notion of markability and present the marking of TL2 as an example. We then formally define markability, and present the marking theorem that states the equivalence of opacity and markability. Next, we formally state the marking relation of TL2 and state that TL2 is markable and hence opaque. Finally, we present our lower bound results for the time complexity of TM algorithms.

## 2 Write-observation and Read-preservation

In this section, we explain the main ideas behind markability by focusing on complete histories with only global reads and writes. A history is complete if every transaction in it is either aborted or committed. A read $R$ by a transaction $T$ is global if $T$ has no write to the same location before $R$. A write $W$ by a transaction $T$ is global if $T$ has no write to the same location after $W$.

A transaction history is *markable* if and only if there exists a *marking* of it that is *write-observant*, *read-preserving*, and *real-time-preserving*. We explain each property in turn.

A marking of a transaction history is a relation on the union of the *transactions* and the *read operations* in the history. We can think of the marking as the union of a collection of orders: The *effect order*: The effect order is a total order of the transactions. The *access orders*: Consider an unaborted read operation $R$ on a location $i$. Let us refer to the committed transactions that have write operation(s) to location $i$ as *writers of* $i$. For each such $R$, the access order is an antisymmetric relation that orders $R$ and every writer of $i$. The effect order represents the order in which the transactions appear to take effect. The access order of a read operation $R$ from a location $i$ represents where the access to $i$ by $R$ has happened between the accesses to $i$ by the writers of $i$.

Note that marking not only recognizes the points where transactions take effect but also the points where reads take place. The effect point of a transaction captures the point where the whole transaction takes effect. But a transaction

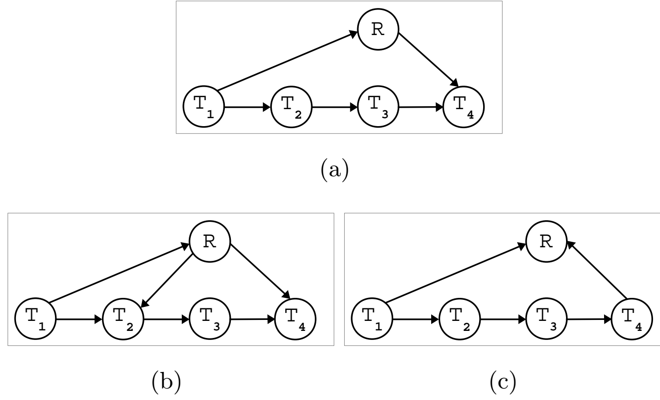


**Fig. 1.** Illustrations of Write-observation and Read-preservation

is split into multiple operations. Particularly, read operations observe values before the commit operation is even invoked. Any value that the TM algorithm returns in response to a read invocation should be justified at the point where the transaction takes effect. There is a point where each writer transaction writes the new value to the underlying shared objects. Every read operation reads the value that it returns at a certain point between the write points of the writer transactions. The access order captures this design decision. Having the access order in addition to the effect order makes it possible to decompose the consistency condition into two orthogonal invariants. Particularly, the read-preservation invariant makes sure that the read value is not overwritten in the interval between the point where a read happens and the point where the transaction takes effect. Next, we will explain write-observation and read-preservation invariants in turn.

At a high level, write-observation means that each read operation should read the most current value. Let us explain this idea in more detail. Consider an unaborted read operation $R$ from a location $i$. Let *pre-accessors* be the writers of $i$ that come before $R$ in the access order for $R$. We can use the effect order to determine the *last* pre-accessor that is, the pre-accessor that is greatest in the effect order. Write-observation requires that the value that $R$ reads be the same as the value written by the last pre-accessor.

Figure 1 illustrates the write-observation and read-preservation invariants. Each sub-figure shows a marking relation $\sqsubseteq$. In every sub-figure, the effect order is $T_1 \sqsubseteq T_2 \sqsubseteq T_3 \sqsubseteq T_4$ and the transaction $T_3$ performs the read operation $R$. In Figure 1(a), $T_1$ and $T_4$ are writers of $i$ and the access order for $R$ is $\{T_1 \sqsubseteq R, R \sqsubseteq T_4\}$. $T_1$ is the last pre-accessor for $R$. Thus, by write-observation, $R$ is expected to return the value that $T_1$ writes to $i$.

At a high level, read-preservation means that the location read by a read operation is not overwritten between the points that the read takes place and the transaction takes effect. Let us explain this idea in more detail. Consider an unaborted read operation $R$ by a transaction $T$ from a location $i$. Intuitively, read-preservation requires that no writer of $i$ comes between $R$ and $T$ in the marking

relation. More precisely, read-preservation requires that there is no writer $T'$ of $i$ that accesses $i$ *after* $R$ and takes effect *before* $T$ and there is no writer $T'$ of $i$ that takes effect *after* $T$ and accesses $i$ *before* $R$. (Note that depending on whether a transaction takes effect earlier or later in its lifetime, one of these two conditions is usually trivially true.) In other words, read-preservation requires the writers to both access $i$ and take effect on the same side of $R$ and $T$. More precisely, if a writer $T'$ accesses $i$ *before* $R$ ($T'$ is marked before $R$ in the access order), then $T'$ takes effect *before* $T$ ($T'$ is marked before $T$ in the effect order) too. Similarly, read-preservation requires that if $T'$ accesses $i$ *after* $R$, it takes effect *after* $T$ too.

The marking relation in Figure 1(a) satisfies read-preservation as there is no writer between $R$ and $T_3$. The transaction $T_1$ accesses $i$ before $R$ and takes effect before $T_3$ too. The transaction $T_4$ accesses $i$ after $R$ and takes effect after $T_3$ too. Figures 1(b) and 1(c) show markings that are not read-preserving. In Figure 1(b), $T_1$, $T_2$ and $T_4$ are writers of $i$ and the access order is $\{T_1 \sqsubseteq R, R \sqsubseteq T_2, R \sqsubseteq T_4\}$. The transaction $T_2$ is between $R$ and $T_3$. Therefore, the marking is not read-preserving. In Figure 1(c), $T_1$ and $T_4$ are writers of $i$ and the access order is $\{T_1 \sqsubseteq R, T_4 \sqsubseteq R\}$. The transaction $T_4$ is between $T_3$ and $R$. Therefore, the marking is not read-preserving.

The real-time-preservation condition requires that if all the events of a transaction $T$ happen before all the events of another transaction $T'$, then $T$ is less than $T'$ in the effect order.

Our marking theorem says that a history is opaque if and only if it is markable. So, to prove opacity, we can focus on proving markability. The algorithm designer can usually define the marking relation readily from the guarantees (such as linearization orders) of the used shared objects. In contrast to opacity, markability of the algorithm can be established by modular verification of the separate markability conditions that involve different aspects of the algorithm.

If a transaction history $H$ is markable, we can show that $H$ is opaque. We construct a justifying history by ordering the transactions in the effect order. Consider an arbitrary read $R$ from $i$ by $T$. We call the writers of $i$ that take effect before $T$, pre-effectors. Let the last pre-effector be the pre-effector that is the greatest in the effect order. We need to show that the value that $R$ returns is the value that the last pre-effector writes. We recall that we refer to the writers of $i$ that access $i$ before $R$ as pre-accessors and refer to the pre-accessor that is greatest in the effect order as the last pre-accessor. First, we argue that pre-accessors are exactly pre-effectors. If a writer of $i$ accesses before $R$, by read-preservation, it does not take effect after $T$. Thus, by totality of effect order, it takes effect before $T$. In the other direction, if a writer of $i$ takes effect before $T$, by read-preservation, it does not access after $R$. Thus, as the access order orders $R$ and every writer of $i$, $T$ accesses before $R$. Second, from write-observation, we have that $R$ returns the value written by the last pre-accessor. Thus, from the two above statements, we have that $R$ returns the value written by the last pre-effector. This is the essence of the condition needed to prove opacity.

## 3 Marking TL2

Now, we look at the marking of the TL2 algorithm [11] as an example. TL2 is specified in Figure 2. The specification first declares the type of the used synchronization objects and then defines the methods of the TM interface.

In the *init* method, each transaction $t$ reads the current snapshot version from *clock* at $I01$ and writes it to the read version register $rver[t]$ at $I02$. The read version is read at $R07$ and $C08$ to validate the read values. TL2 is a deferred-update TM algorithm. A value that a transaction $t$ writes to a location is buffered in the write set $wset[t]$ at $W01$ and is written back to register $reg[i]$ at $C16_i$ while $t$ is committing. TL2 records a version in the register $ver[i]$ for the value stored in the register $reg[i]$. The version register $ver[i]$ is updated to ascending numbers at $C17_i$ after new values are written back to $reg[i]$ at $C16_i$. The try-lock $lock[i]$ is used for exclusive access to the registers for location $i$. At commit, the lock $lock[i]$ of each location $i$ in the write set $wset[t]$ is acquired at $C01$ to $C06$. (If a lock cannot be acquired, the previously acquired locks are released at $C05$ and the transaction is aborted at $C06$.) Then, a new snapshot number is read from *clock* at $C07$. Then, for each location in the read set $rset[t]$, first $lock[i]$ and then $ver[i]$ are read at $C10_i$ and $C11_i$ and the read is validated. (If a read is not validated, the acquired locks are released at $C13$ and the transaction is aborted at $C14$.) Finally, the value buffered for each location $i$ in $wset[t]$ is written back at $C15_i$ to $C18_i$. For each pair in the write set $wset[t]$, the following three operations are executed in order. First, the buffered value is written back to $reg[i]$, then $ver[i]$ is updated, and then $lock[i]$ is released. To read a location $i$, a transaction reads $ver[i]$, $reg[i]$, $lock[i]$ and again $ver[i]$ in order at $R03$ to $R06$ and then validates the read. (If the validation fails, the transaction is aborted.) Finally, $i$ is added to the read set $rset[t]$ and the read value is returned.

Let us describe the marking relation for TL2. The *clock* object numbers the snapshots. Every transaction reads an initial snapshot number at $I01$. A committing transaction makes a new snapshot at $C07$. The effect point of a TL2 transaction is $I01$, if it is live or aborted and, is $C07$, if it is committed. The effect order of transactions is the linearization order of *clock* for their effect points. The access point of a read operation is at $R04$ where $reg[i]$ is read and the access point of a writer of $i$ is $C16_i$ where $reg[i]$ is written. Consider a read $R$ from $i$ and a writer $T'$ to $i$. If the access point of $T'$ is executed before the access point of $R$, then $T'$ is ordered before $R$ in the access order of $R$. Otherwise, $T'$ is ordered after $R$ in the access order of $R$. The access and effect points for markability of a TM are reminiscent of the linearization points for linearizability of a concurrent data structure.

One of the two conjuncts of the read-preservation property requires that for every transaction $T$ with an unaborted read operation $R$ from a location $i$, there is no writer $T'$ of $i$ such that $T'$ takes effect after $T$ and accesses $i$ before $R$. Let us see how TL2 preserves this property. We assume that there exists such a writer $T'$ and show that the validation checks embodied in TL2 detect the existence of $T'$ and abort $R$. We consider a transaction $T$ with a read operation $R$ from a location $i$ and a writer $T'$ of $i$. We assume that $T'$ takes effect after

| | |
|---|---|
| $reg$: **BasicRegister**$[\lvert I\rvert]$, | $rver$: **ThreadLocal BasicRegister**, |
| $ver$: **AtomicRegister**$[\lvert I\rvert]$, | $rset$: **ThreadLocal BasicSet**, |
| $lock$: **TryLock**$[\lvert I\rvert]$, | $wset$: **ThreadLocal BasicMap**, |
| $clock$: **SCounter**, | $lset$: **ThreadLocal BasicSet** |

```
def init_t()
[I01 ▷]  snap = clock.read()
I02 ▷    rver[t].write(snap)
I03 ▷    return ok

def read_t(i)
R01 ▷    pv = wset[t].get(i)
         if (pv ≠ ⊥)
R02 ▷        return pv

R03 ▷    s_1 = ver[i].read()
[R04 ▷]  v = reg[i].read()
R05 ▷    l = lock[i].read()
R06 ▷    s_2 = ver[i].read()
R07 ▷    sver = rver[t].read()
         if (¬(¬l ∧ s_1 = s_2 ∧ s_2 ≤ sver))
R08 ▷        return 𝔸

R09 ▷    rset[t].add(i)
R10 ▷    return v
{R03 → R04, R04 → R05, R05 → R06}

def write_t(i, v)
W01 ▷    wset[t].put(i, v)
W02 ▷    return ok

def abort_t()
A01 ▷        return 𝔸
```

```
def commit_t()
C01 ▷      foreach (i ∈ wset[t])
C02_i ▷        locked = lock[i].trylock()
               if (¬locked)
C03_i ▷            lset.add(i)
               else
C04_i ▷            foreach (j ∈ lset)
C05_ij ▷               lock[j].unlock()
C06_i ▷            return 𝔸

[C07 ▷]    wver = clock.iaf()

C08 ▷      sver = rver[t].read()
           if (wver ≠ sver + 1)
C09 ▷          foreach (i ∈ rset[t])
C10_i ▷            l = lock[i].read()
C11_i ▷            s = ver[i].read()
                   if (¬(¬l ∧ s ≤ sver))
C12_i ▷                foreach (j ∈ lset)
C13_ij ▷                   lock[j].unlock()
C14_i ▷                return 𝔸

C15 ▷      foreach ((i, v) ∈ wset[t])
[C16_i ▷]      reg[i].write(v)
C17_i ▷        ver[i].write(wver)
C18_i ▷        lock[i].unlock()

C19 ▷      return ℂ
{C01 → C07, C10 → C11, C09 → C15,
 C16 → C17, C17 → C18}
```

**Fig. 2.** $TL2$ Algorithm Specification

$T$ and $T'$ accesses $i$ before $R$. For brevity, we consider only the case that $T$ is a live or aborted (not a committed) transaction. Figure 3 depicts the two transactions. We use the binary operators $\prec_X$ to denote execution order, $\sim_X$ to denote concurrent execution and $\precsim_X$ to denote in-order or concurrent execution of method calls. We use the binary operators $\prec_{clock}$, $\prec_{ver[i]}$ and $\prec_{lock[i]}$ to denote

| $T$ | | $T'$ | |
|---|---|---|---|
| $I01$ ▷ | $snap = clock.read()$ | $C02_i$ ▷ | $lock[i].trylock()$ |
| $I02$ ▷ | $rver[t].write(snap)$ | | ... |
| | | $C07$ ▷ | $wver = clock.iaf()$ |
| | ... | | ... |
| | | $C16_i$ ▷ | $reg[i].write(v)$ |
| $R04$ ▷ | $v = reg[i].read()$ | $C17_i$ ▷ | $ver[i].write(wver)$ |
| $R05$ ▷ | $l = lock[i].read()$ | $C18_i$ ▷ | $lock[i].unlock()$ |
| $R06$ ▷ | $s_2 = ver[i].read()$ | | |
| $R07$ ▷ | $sver = rver[t].read()$ | | |
| | **if** $(\neg(\neg l \wedge s_1 = s_2 \wedge s_2 \leq sver))$ **return** $\mathbb{A}$ | | |

**Fig. 3.** TL2 Read-Preservation Example

the linearization order of $clock$, $ver[i]$ and $lock[i]$ respectively.[1] We recall that the real-time-preservation property of a linearizable object $o$ states that if a method call $m_1$ on $o$ is executed before another method call $m_2$ on $o$, then $m_1$ is linearized before $m_2$. Equivalently, if $m_1$ is linearized before $m_2$, then $m_1$ is executed before or concurrent to $m_2$. By the marking relation defined above, from the premise that $T'$ takes effect after $T$, and that $T$ is aborted and $T'$ is committed, we have (1) $I01 \prec_{clock} C07$. Similarly, by the marking relation defined above, from the premise that $T'$ accesses $i$ before $R$, we have (2) $C16_i \prec_{reg[i]} R04$. The method calls $R05$ and $C18_i$ are on the object $lock[i]$. We consider two cases for the linearization order of them and show that $R$ returns $\mathbb{A}$ in both cases. Case 1: (3) $R05 \prec_{lock[i]} C18_i$. From the execution, we have (4) $C02_i \prec_X C16_i$ and (5) $R04 \prec_X R05$. By the real-time-preservation property for $ver[i]$ on 2, we have (6) $C16_i \precsim_X R04$. By the transitivity of the execution order on 4, 6 and 5, we have $C02_i \prec_X R05$; thus, by the real-time-preservation property for $lock[i]$, we have (7) $C02_i \prec_{lock[i]} R05$. From 7 and 3, we have that $R05$ is executed when $lock[i]$ is acquired. Therefore, $R05$ returns $true$ i.e. $l = true$. Thus, the validation check fails and $R$ returns $\mathbb{A}$.

Case 2: (8) $C18_i \prec_{lock[i]} R05$. By the real-time-preservation property for $lock[i]$, from 8, we have (9) $C18_i \precsim_X R05$. From the execution, we have (10) $C17_i \prec_X C18_i$ and (11) $R05 \prec_X R06$. By the transitivity of the execution order on 10, 9 and 11, we have (12) $C17_i \prec_X R06$. By the real-time-preservation property for $ver[i]$, from 12, we have (13) $C17_i \prec_{ver[i]} R06$. It is straightforward to separately prove that (14) The register $ver[i]$ is updated only to ascending numbers. From 14 and 13, we have that $R06$ reads a value that is greater than or equal to the value that $C17_i$ writes i.e. (15) $s_2 \geq wver$. From 1, and that $iaf$

[1] We have formally proved the markability of TL2 using a novel program logic [27] that facilitates reasoning about execution and linearization orders. To keep the focus of this paper on markability, we use a simplified reasoning instead of the logic.

returns the incremented value, we have (16) $snap < wver$. The value of $sver$ is read at $R07$ from $rver$. The thread-local register $rver$ is only assigned at $I02$ to $snap$. Thus, we have (17) $snap = sver$. From 15, 16 and 17, we have $s_2 > sver$. Thus, the validation check fails and $R$ returns $\mathbb{A}$ in this case too.

Please see the appendix [29] for the proof of markability of TL2 and also the marking relations for DSTM (visible reads) [23] and NORec [9] TM algorithms.

## 4 Markability

In this section, we first present preliminary definitions about execution histories and then, present the formal definition of markability and state its equivalence to opacity.

### 4.1 Histories

**Strings.** We use $\|s\|$ to denote the size of the string $s$. 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 string (i.e. contains no repeating occurrence of the alphabet.) For any $s_1, s_2$, 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, $ab \lhd_{abcde} de$.

**Method calls and events.** An *invocation event* is of the form $inv(l \triangleright o.n_T(v))$ where $l$ is a label, $o$ is an object, $n$ is a method name, $T$ is a transaction identifier and $v$ is a value. A *response event* is of the form $ret(l \triangleright v)$ where $l$ is a label and $v$ is a value. A *completed* method call is the sequence of an invocation event and the matching response event (with the same label). We use $l \triangleright o.n_T(v){:}v$ to denote the completed method call $inv(l \triangleright o.n_T(v)) \cdot ret(l \triangleright v)$.

**Operations on event sequences.** Let $E$ and $E'$ be *event sequence*s. We use $E \cdot E'$ to denote the *concatenation* of $E$ and $E'$. For a transaction $T$, we use $E|T$ to denote the subsequence of all events of $T$ in $E$. A sequence of events is *sequential* if and only if it is a sequence of completed method calls possibly followed by an invocation event. A transaction $T$ is *sequential* in a sequence of events $E$ if $E|T$ is sequential.

**Execution history.** An *execution history* is an event sequence where invocation events have unique labels and every transaction is sequential. We say label $l$ is in $X$ and write $l \in X$ if there is an invocation event with label $l$ in $X$. We use $l$, $R$ and $W$ to denote labels. As the labels are unique in a history, the following functions on labels are defined. The functions $obj_X$, $name_X$, $trans_X$, $arg1_X$, $arg2_X$, $retv_X$ map labels to the receiving object, the method name, the transaction identifier, the first and the second arguments, and the return value associated with the labels. Similary, $iEv$ and $rEv$ functions on labels map labels to the invocation and the response events associated with the labels.

**Real-time relations.** For an execution history $X$, we define the *method call real-time* relations $\prec_X$ and $\preceq_X$ on labels as follows: First, $l_1 \prec_X l_2$ iff $rEv(l_1) \lhd_X iEv(l_2)$. Second, $l_1 \preceq_X l_2$ iff $l_1 \prec_X l_2 \ \vee \ l_1 = l_2$.

For an execution history $X$, we define the *transaction real-time* relations $\lessdot_X$ and $\preceq_X$ as follows. First, $T \lessdot_X T'$ iff $X|T \lhd_X X|T'$. Second, $T \preceq_X T'$ iff $T \lessdot_X T' \vee T = T'$.

**Transactional Memory.** The *transactional memory* is a singleton object $mem$ that encapsulates a set of locations where each location, $i \in I$, $I = \{1, \ldots, m\}$ encapsulates a value $v$. The object $mem$ has five methods $init_t()$, $read_t(i)$, $write_t(i, v)$, $commit_t()$ and $abort_t()$. The parameter $t$ is the invoking transaction identifier. The method call $init_t()$ initializes $t$ and returns $ok$. The method call $read_t(i)$ returns the value of location $i$ or aborts $t$ and returns $\mathbb{A}$. The method $write_t(i, v)$ writes $v$ to location $i$ and returns $ok$ or aborts $t$ and returns $\mathbb{A}$. The method $commit_t()$ tries to commit transaction $t$. If $t$ is successfully committed, it returns $\mathbb{C}$; otherwise, it returns $\mathbb{A}$. The method $abort_t()$ aborts $t$ and returns $\mathbb{A}$. The object $mem$ can be implicit, that is $read_t(i)$ abbreviates $mem.read_T(i)$. The reserved values $ok$, $\mathbb{A}$, $\mathbb{C}$ denote successful completion of writes and, abortion and commitment of transactions respectively.

**Transaction History.** A *transaction history* $H$ is an execution history such that $H|mem = H_{Init} \cdot H'$ with the following conditions. $H_{Init}$ is the following history that initializes every location to $v_0$. $H_{Init} = l_{0i} \rhd init_{T_0}() \cdot l_{00} \rhd write_{T_0}(1, v_0){:}ok \cdot \ldots \cdot l_{0m} \rhd write_{T_0}(m, v_0){:}ok \cdot l_{0c} \rhd commit_{T_0}{:}\mathbb{C}$. For every $T \in H'$, the history $H'|T$ is a prefix of $E.E'$. The event sequence $E$ is the initialization method call $l \rhd init_T()$ (for some $l$), and then a sequence of reads $l \rhd read_T(i){:}v$ and writes $l \rhd write_T(i, v)$ (for some $l$, $i$, and $v$). The event sequence $E'$ is one of the following sequences (for some $l$, $i$, and $v$): (1) $inv(l \rhd read_T(i))$, $ret(l \rhd \mathbb{A})$, (2) $inv(l \rhd write_T(i, v))$, $ret(l \rhd \mathbb{A})$, (3) $inv(l \rhd commit_T())$, $ret(l \rhd \mathbb{C})$, (4) $inv(l \rhd commit_T())$, $ret(l \rhd \mathbb{A})$, or (5) $inv(l \rhd abort_T())$, $ret(l \rhd \mathbb{A})$. Let $THistory$ denote the set of transaction histories. Let $Trans(H)$ denote the set of transactions of $H$. 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 TM algorithm specification $\pi$, let $\mathbb{H}(\pi)$ denote the set of complete transaction histories that result from execution of transactions with $\pi$.

### 4.2 Formal Definition of Markability

First, we present some preliminary definitions in Figure 4. (We use the prefix $T$ before some of the terms for transactions to avoid confusion with similar terms that are usually used for general concurrent objects.) 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 *pending* transaction has a pending event and a *commit-pending* transaction has a commit pending event. An *extension* of a history is obtained by committing or aborting its commit-pending transactions and aborting the other live transactions.

A *local read* is a read that is preceded by a write by the same transaction to the same location. Intuitively, a local read should read a value that is previously written by the same transaction and hence the name. A *global read* is a read that is not local. A *local write* is a write that precedes a write by the same transaction

$$
\begin{aligned}
Committed(H) &= \{T \mid \exists l \in H\colon obj_H(l) = mem \ \wedge\ trans_H(l) = T \ \wedge \\
&\qquad retv_H(l) = \mathbb{C}\} \\
Aborted(H) &= \{T \mid \exists l \in H\colon obj_H(l) = mem \ \wedge\ trans_H(l) = T \ \wedge \\
&\qquad retv_H(l) = \mathbb{A}\} \\
Completed(H) &= Committed(H) \cup Aborted(H) \\
Live(H) &= Trans(H) \setminus Completed(H) \\
CommitPending(H) &= \{T \mid T \in Live(H) \ \wedge\ \exists l \in H\colon \\
&\qquad obj_H(l) = mem \ \wedge\ obj_H(l) = mem \ \wedge\ trans_H(l) = T\} \\
TExtension(H) &= \{H' \mid H' \in THistory \ \wedge\ \exists H''\colon H' = H \cdot H'' \ \wedge \\
&\qquad Trans(H'') \subseteq Trans(H) \ \wedge\ \forall T\colon \|H''|T\| \leq 1 \ \wedge \\
&\qquad Live(H) \setminus CommitPending(H) \subseteq Aborted(H') \ \wedge \\
&\qquad CommitPending(H) \subseteq Completed(H')\} \\
TReads(H) &= \{R \mid R \in H \ \wedge\ obj_H(R) = mem \ \wedge\ name_H(R) = read \ \wedge \\
&\qquad retv_H(R) \neq \mathbb{A}\} \\
TWrites(H) &= \{W \mid W \in H \ \wedge\ obj_H(W) = mem \ \wedge\ name_H(W) = write \ \wedge \\
&\qquad retv_H(W) \neq \mathbb{A}\} \\
LocalTReads(H) &= \{R \mid R \in TReads(H) \ \wedge\ \exists W \in TWrites(H)\colon \\
&\qquad trans_H(R) = trans_H(W) \ \wedge \\
&\qquad arg1_H(R) = arg1_H(W) \ \wedge\ W \prec_H R\} \\
GlobalTReads(H) &= TReads(H) \setminus LocalTReads(H) \\
LocalTWrites(H) &= \{W \mid W \in TWrites(H) \ \wedge\ \exists W' \in TWrites(H)\colon \\
&\qquad trans_H(W) = trans_H(W') \ \wedge \\
&\qquad arg1_H(W) = arg1_H(W') \ \wedge\ W \prec_H W'\} \\
GlobalTWrites(H) &= TWrites(H) \setminus LocalTWrites(H) \\
Writers_H(i) &= \{T \mid T \in Trans(H) \ \wedge\ \exists l \in TWrites(H)\colon arg1_H(l) = i \ \wedge \\
&\qquad trans_H(l) = T \ \wedge\ T \in Committed(H)\}
\end{aligned}
$$

**Fig. 4.** Basic Definitions

to the same location. A local write is overwritten by the same transaction and hence the name. A *global write* is a write that is not local. The *writers of* $i$ are the committed transactions that write to location $i$.

Markability is defined in Figure 5. A *marking* $\sqsubseteq$ of a transaction history is the union of the following relations on the set of transactions and the global reads. The *effect order*: The set of transactions is totally ordered by the marking relation $\sqsubseteq$. In other words, the marking relation $\sqsubseteq$ is total, antisymmetric and transitive on the set of transactions. The *access orders*: For each global read $R$ from a location $i$, $R$ and every writer of $i$ are ordered by the marking relation $\sqsubseteq$. In other words, the marking relation $\sqsubseteq$ totally orders every global read $R$ from a location $i$ with respect to writers of $i$ and is antisymmetric.

The *write-observation* property is comprised of the two properties: *local write-observation* and *global write-observation*. Local write-observation requires that every local read $R$ from a location $i$ returns the value written by the last write

$$
\begin{aligned}
&Marking(H) = \{\sqsubseteq \ | \\
&\quad \forall T1, T2, T3 \in Trans(H)\colon \\
&\quad\quad (T1 \sqsubseteq T2 \ \vee \ T2 \sqsubseteq T1) \ \wedge \\
&\quad\quad (T1 \sqsubseteq T2 \ \wedge \ T2 \sqsubseteq T1) \Rightarrow (T1 = T2) \ \wedge \\
&\quad\quad (T1 \sqsubseteq T2) \ \wedge \ (T2 \sqsubseteq T3) \Rightarrow (T1 \sqsubseteq T3) \ \wedge \\
&\quad \forall R, T\colon \mathit{Let}\ i = arg1_H(R)\colon (R \in GlobalTRead(H) \ \wedge \ T \in Writers_H(i)) \Rightarrow \\
&\quad\quad (R \sqsubseteq T \ \vee \ T \sqsubseteq R) \ \wedge \\
&\quad\quad (R \sqsubseteq T \Rightarrow \neg T \sqsubseteq R) \ \wedge \ (T \sqsubseteq R \Rightarrow \neg R \sqsubseteq T)\} \\
&NoWriteBetween_H(W, R) \Leftrightarrow \\
&\quad \forall W' \in TWrites(H)\colon W' \preceq_H W \ \vee \ R \prec_H W' \\
&LocalWriteObs(H) \Leftrightarrow \\
&\quad \forall R \in LocalTReads(H)\colon \mathit{Let}\ T = trans_H(R), i = arg1_H(R), H' = H|T|i\colon \\
&\quad \exists W \in TWrites(H')\colon \\
&\quad W \prec_{H'} R \ \wedge \ NoWriteBetween_{H'}(W, R) \ \wedge \ retv_{H'}(R) = arg2_{H'}(W) \\
&NoWriterBetween_{H,i}(x, \sqsubseteq, x') \Leftrightarrow \\
&\quad \forall T \in Writers_H(i)\colon T \sqsubseteq x \ \vee \ x' \sqsubseteq T \\
&LastPreAccessor_{H,\sqsubseteq}(T', R) \Leftrightarrow \mathit{Let}\ i = arg1_H(R), T = trans_H(R)\colon \\
&\quad T' \in Writers_H(i) \ \wedge \ T' \neq T \ \wedge \ T' \sqsubset R \ \wedge \ NoWriterBetween_{H,i}(T', \sqsubseteq, R) \\
&GlobalWriteObs(H, \sqsubseteq) \Leftrightarrow \\
&\quad \forall R \in GlobalTReads(H)\colon \exists W \in GlobalTWrites(H)\colon \mathit{Let}\ T' = trans_H(W)\colon \\
&\quad LastPreAccessor_{H,\sqsubseteq}(T', R) \ \wedge \\
&\quad arg1_H(R) = arg1_H(W) \ \wedge \ retv_H(R) = arg2_H(W) \\
&WriteObs(H, \sqsubseteq) \Leftrightarrow \\
&\quad LocalWriteObs(H) \ \wedge \ GlobalWriteObs(H, \sqsubseteq) \\
&ReadPres(H, \sqsubseteq) \Leftrightarrow \\
&\quad \forall R \in GlobalTReads(H)\colon \mathit{Let}\ i = arg1_H(R), T = trans_H(R)\colon \\
&\quad NoWriterBetween_{H,i}(R, \sqsubseteq, T) \ \wedge \ NoWriterBetween_{H,i}(T, \sqsubseteq, R) \\
&RealTimePres(H, \sqsubseteq) \Leftrightarrow \\
&\quad \preceq\!\!\!\!\prec_H \ \subseteq \ \sqsubseteq \\
&FinalStateMarkable = \{H \ | \\
&\quad H \in THistory \ \wedge \ \exists H' \in TExtension(H)\colon \exists \sqsubseteq \in Marking(H')\colon \\
&\quad WriteObs(H', \sqsubseteq) \ \wedge \ ReadPres(H', \sqsubseteq) \ \wedge \ RealTimePres(H', \sqsubseteq)\}
\end{aligned}
$$

**Fig. 5.** *FinalStateMarkable*

to $i$ that is executed before $R$ by the same transaction. As we defined before, pre-accessors of $R$ are the writers of $i$ that are ordered before $R$ in the access order and the last pre-accessor of $R$ is the one that is greatest in the effect order. Global write-observation requires that the value that every global read $R$ from a location $i$ returns is the value written by the global write to $i$ by the last pre-accessor transaction of $R$.

The *Read-preservation* property requires that for every global read $R$ from location $i$ by transaction $T$, there is no writer transaction $T'$ of $i$ such that $T'$

is marked between $R$ and $T$ (i.e. $T'$ accesses $i$ after $R$ and takes effect before $T$), or similarly, $T'$ is marked between $T$ and $R$ (i.e. $T'$ takes effect after $T$ and accesses $i$ before $R$).

The *real-time-preservation* property requires that if $T$ is before $T'$ in the transaction real-time order, then $T$ takes effect before $T'$ as well.

A transaction history is *final-state-markable* if and only if there exists a marking for an extension of it that is write-observant, read-preserving, and real-time-preserving.

The marking theorem states that a transaction history is final-state-opaque if and only if it is final-state-markable. The formal definition of opacity and the proofs are available in the appendix [29].

**Theorem 1 (Marking).** $FinalStateOpaque = FinalStateMarkable.$

## 5 Opacity of TL2

Now, we define the marking relation for the TL2 algorithm in Figure 2. We use the call string label $l_1$'$l_2$ to denote the method call labeled $l_2$ that is executed in the body of the method call labeled $l_1$. We use $initOf_H(T)$ and $commitOf_H(T)$ to denote the *init* and *commit* method calls of $T$ in $H$.

**Definition 1 (Marking TL2).** *Consider an execution history $H \in \mathbb{H}(TL2)$. Let*

$$\begin{aligned} Eff(T) &= \begin{cases} initOf_H(T)\text{'}I01 & \text{if } T \in Aborted(H) \\ commitOf_H(T)\text{'}C07 & \text{if } T \in Committed(H) \end{cases} \\ readAcc(R) &= R\text{'}R04 \\ writeAcc(T, i) &= commitOf_H(T)\text{'}C16_i \end{aligned}$$

*The marking $\sqsubseteq$ for $H$ is the reflexive closure of the relation $\sqsubset$ that is defined as follows:*

$$\begin{aligned} &\{(T, T') \mid T, T' \in Trans(H) \ \wedge \ Eff(T) \prec_{clock} Eff(T')\} \cup \\ &\{(T, R) \mid Let\ i = arg1(R) \colon R \in GlobalTReads(H), T \in Writers_H(i) \wedge \\ &\qquad writeAcc(T, i) \prec_H readAcc(R)\} \cup \\ &\{(R, T) \mid Let\ i = arg1(R) \colon R \in GlobalTReads(H), T \in Writers_H(i) \wedge \\ &\qquad readAcc(R) \precsim_H writeAcc(T, i)\} \end{aligned}$$

The following theorems state the markability and the opacity of $TL2$.

**Theorem 2 (Markability of TL2).** $\forall H \in \mathbb{H}(TL2) \colon H \in FinalStateMarkable$

**Corollary 1 (Opacity of TL2).** $\forall H \in \mathbb{H}(TL2) \colon H \in FinalStateOpaque$

The appendix [29] presents the proofs. The above corollary states that every history of TL2 is final-state-opaque. As the set of histories of a TM algorithm is prefix-closed, a TM algorithm is opaque if and only if every history of it is final-state-opaque. (See [21], Observation 7.4.) Therefore, TL2 is opaque.

## 6 The Cost of Read Validation

The read-preservation invariant requires the TM algorithm to check that a read location is not overwritten between the point where the location is read and the point where the transaction takes effect. This requirement motivated us to study how read-preservation can influence the time complexity of TM operations and helped us construct client scenarios that exhibit lower bounds. We present a generalization of the seminal lower bound result presented in [20]. We first recall some definitions from previous works on the inherent complexity of TM [3, 19, 20, 34].

An aborted transaction that did not invoke an abort operation is said to be *forcefully* aborted. We say that two transactions *conflict* if they access the same location and one of them writes to the location. A TM algorithm is (weakly) *progressive* if and only if it forcefully aborts a transaction only when it conflicts with a live transaction. More precisely, it aborts a transaction only when there is a time $t$ at which it conflicts with another concurrent transaction that is live at time $t$ (not committed or aborted by time $t$). In addition to providing progress, progressive TM algorithms are expected to retry transactions less frequently and therefore, improve performance.

A TM algorithm is *invisible-reads* if and only if the read operation does not mutate (i.e. change the state of) any base object. Mutating base objects can potentially invalidate the caches and adversely affect performance. Thus, most high-performance TM algorithms are invisible-reads. A transaction is *read-only* if and only if it does invoke any write operations. We assume that the abort operation for a read-only transaction does not mutate any base shared object.

Two transactions *contend* on a base object $o$ if and only if they access $o$ and at least one of them mutates $o$. A TM algorithm is (strictly) *disjoint-access-parallel* if and only if two transactions contend on a base object only if they access a common memory location. Disjoint-access-parallelism can improve scalability as transactions that access disjoint memory locations access disjoint base objects.

A TM algorithm is *single-version* if and only if it stores a single value for each memory location in the base objects.

**Theorem 3.** *The time complexity of the commit operation of every opaque, progressive, disjoint-access-parallel and invisible-reads TM algorithm is* $\Omega(|\mathcal{R}|)$ *where* $\mathcal{R}$ *is the read set.*

This theorem shows that designers should pick at least one of the following sources of inefficiency in the design of every opaque TM algorithm: aborting non-conflicting transactions, sharing base objects between transactions that access disjoint locations, visible reads or linear-time complexity of the commit method. As an example, TL2 shares the *clock* object between all transactions and is, therefore, not disjoint-access-parallel. In addition, it has linear-time read-validation in the commit method.

**Theorem 4.** *The time complexity of the commit operation of every opaque, progressive, and invisible-reads TM algorithm that stores information about a*

*constant number of locations in each shared object is $\Omega(|\mathcal{R}|)$ where $\mathcal{R}$ is the read set.*

The above theorem generalizes Theorem 3 of [20] by dropping the single-version requirement. Note that the assumption about limited capacity of shared objects is stated before the theorem in [20] and explicitly in the theorem here. We leave the proofs to the appendix [29].

## 7 Conclusion

We presented a decomposition of opacity called markability as a conjunction of separate invariants. We proved the equivalence of opacity and markability. We showcased the applicability of markability as a proof technique for opacity by stating the marking relation and proving the markability of the TL2 algorithm. In addition, we presented a lower bound for the time complexity of TM algorithms.

## References


1. C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA*, 2005.
2. H. Attiya, S. Hans, P. Kuznetsov, and S. Ravi. Safety of deferred update in transactional memory. In *ICDCS*, 2013.
3. H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory of Computing Systems*, 49(4), 2011.
4. W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun. Implementing and evaluating a model checker for transactional memory systems. In *ICECCS*, 2010.
5. A. Cohen, J. W. O'Leary, A. Pnueli, M. R. Tuttle, and L. D. Zuck. Verifying correctness of transactional memories. In *FMCAD*, 2007.
6. A. Cohen, A. Pnueli, and L. D. Zuck. Mechanical verification of transactional memories with non-transactional memory accesses. In *CAV*, 2008.
7. I. Corporation. Intel architecture instruction set extensions programming reference, tsx. 319433-012, 2012.
8. L. Dalessandro, F. Carouge, S. White, Y. Lev, M. Moir, M. L. Scott, and M. F. Spear. Hybrid norec: A case study in the effectiveness of best effort hardware transactional memory. In *ASPLOS*, 2011.
9. L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *PPoPP*, 2010.
10. P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. *SIGPLAN Not.*, 41(11), 2006.
11. D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *DISC, (LNCS 4167)*, 2006.
12. D. Dice and N. Shavit. TLRW: Return of the read-write lock. In *SPAA*, 2010.
13. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, 2012.
14. M. Emmi, R. Majumdar, and R. Manevich. Parameterized verification of transactional memories. In *PLDI*, 2010.

15. P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *PPoPP*, 2008.
16. R. Guerraoui, T. A. Henzinger, B. Jobstmann, and V. Singh. Model checking transactional memories. In *PLDI*, 2008.
17. R. Guerraoui, T. A. Henzinger, and V. Singh. Software transactional memory on relaxed memory models. In *CAV*, 2009.
18. R. Guerraoui, T. A. Henzinger, and V. Singh. Model checking transactional memories. *Distributed Computing*, 2010.
19. R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *SPAA*, 2008.
20. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, 2008.
21. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory*. M&C, 2010.
22. L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *ISCA*, 2004.
23. M. Herlihy, V. Luchangco, M. Moir, and I. W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, 2003.
24. M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.
25. D. Imbs, J. R. de Mendivil, and M. Raynal. Brief announcement: virtual world consistency: a new condition for stm systems. In *PODC*, 2009.
26. S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *PPoPP*, 2006.
27. M. Lesani. On the correctness of transactional memory algorithms. Phd Dissertation. 2014. `http://www.cs.ucla.edu/~lesani/companion/dissertation`.
28. M. Lesani, V. Luchangco, and M. Moir. A framework for formally verifying software transactional memory algorithms. In *CONCUR*, 2012.
29. M. Lesani and J. Palsberg. Decomposing opacity, the companion page. `http://www.cs.ucla.edu/~lesani/companion/disc14`.
30. M. Lesani and J. Palsberg. Proving non-opacity. In *DISC, (LNCS 8205)*. 2013.
31. A. Matveev and N. Shavit. Reduced hardware transactions: A new approach to hybrid transactional memory. In *SPAA*, 2013.
32. C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, 2007.
33. J. O'Leary, B. Saha, and M. R. Tuttle. Model checking transactional memory with spin. In *ICDCS*, 2009.
34. D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in stm. In *PODC*, 2010.
35. B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.
36. N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.
37. A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. Michael. Evaluation of blue gene/q hardware support for transactional memories. In *PACT*, 2012.