Non-Interference and Local Correctness in Transactional Memory

Petr Kuznetsov\(^1\) and Sathya Peri\(^2\)

\(^1\) Télécom ParisTech
petr.kuznetsov@telecom-paristech.fr

\(^2\) IIT Patna
sathya@iitp.ac.in

Abstract. Transactional memory promises to make concurrent programming tractable and efficient by allowing the user to assemble sequences of actions in atomic transactions with all-or-nothing semantics. It is believed that, by its very virtue, transactional memory must ensure that all committed transactions constitute a serial execution respecting the real-time order. In contrast, aborted or incomplete transactions should not “take effect.” But what does “not taking effect” mean exactly? It seems natural to expect that aborted or incomplete transactions do not appear in the global serial execution, and, thus, no committed transaction can be affected by them. We investigate another, less obvious, feature of “not taking effect” called non-interference: aborted or incomplete transactions should not force any other transaction to abort. In the strongest form of non-interference that we explore in this paper, by removing a subset of aborted or incomplete transactions from the history, we should not be able to turn an aborted transaction into a committed one without violating the correctness criterion. We show that non-interference is, in a strict sense, not implementable with respect to the popular criterion of opacity that requires all transactions (be they committed, aborted or incomplete) to witness the same global serial execution. In contrast, when we only require local correctness, non-interference is implementable. Informally, a correctness criterion is local if it only requires that every transaction can be serialized along with (a subset of) the transactions committed before its last event (aborted or incomplete transactions ignored). We give a few examples of local correctness properties, including the recently proposed criterion of virtual world consistency, and present a simple though efficient implementation that satisfies non-interference and local opacity.

1 Introduction

Transactional memory (TM) promises to make concurrent programming efficient and tractable. The programmer simply represents a sequence of instructions that should appear atomic as a speculative transaction that may either commit or abort. It is usually expected that a TM serializes all committed transactions, i.e., makes them appear as in some sequential execution. An implication of this requirement is that no committed transaction can read values written by a transaction that is aborted or might abort in the future. Intuitively, this is a desirable property because it does not allow a write performed within a transaction to get “visible” as long as there is a chance for the transaction to abort.

But is this all we can do if we do not want aborted or incomplete transactions to “take effect”? We observe that there is a more subtle side of the “taking effect” phenomenon that is usually not taken into consideration. An incomplete or aborted transaction may cause another transaction to abort. Suppose we have an execution in which an aborted transaction \(T\) cannot be committed without violating correctness of the execution, but if we remove some incomplete or aborted transactions, then \(T\) can be committed. This property, originally highlighted in [14, 15], is called non-interference.

Thus, ideally, a TM must “insulate” transactions that are aborted or might abort in the future from producing any effect, either by affecting reads of other transactions or by provoking forceful aborts.

Defining non-interference. Consider non-interference as a characteristics of an implementation. A TM implementation \(M\) is non-interfering if removing an aborted or incomplete not concurrently committing transaction from a history (a sequence of events on the TM interface) of \(M\) would still result in a history in \(M\). We
observe that many existing TM implementations that employ *commit-time* lock acquisition or version update (e.g., RingSTM [16], NOrec [3]) are non-interfering in this sense. In contrast, some *encounter-time* implementations, such as TinySTM [4], are not non-interfering.

This paper rather focuses on non-interference as a characteristics of a *correctness criterion*, which results in a much stronger restriction on implementations. We intend to understand whether this strong notion of non-interference is achievable and at what cost, which we believe is a challenging theoretical question. For a given correctness criterion $C$, a TM implementation $M$ is $C$-non-interfering if removing an aborted or incomplete transaction from any history of $M$ does not allow committing another aborted transaction while still preserving $C$. We observe that $C$-non-interference produces a subset of *permissive* [5] with respect to $C$ histories. This is not difficult to see if we recall that in a permissive (with respect to $C$) history, no aborted transaction can be turned into a committed one while still satisfying $C$.

In particular, when we focus on *opaque* histories [6, 7], we observe that non-interference gives a *strict* subset of permissive opaque histories. Opacity requires that all transactions (be they committed, aborted, or incomplete) constitute a consistent sequential execution in which every read returns the latest committed written value. This is a strong requirement, because it expects every transaction (even aborted or incomplete) to witness the same sequential execution. Indeed, there exist permissive opaque histories that do not provide non-interference: some aborted transactions force other transactions to abort.

For example, consider the history in Figure 1. Here the very fact that the incomplete operation $T_2$ read the “new” (written by $T_3$) value in object $x$ and the “old” (initial) value in object $y$ prevents an updating transaction $T_1$ from committing. Suppose that $T_1$ commits. Then $T_2$ can only be *serialized* (put in the global sequential order) after $T_3$ and before $T_1$, while $T_1$ can only be serialized before $T_3$. Thus, we obtain a cycle which prevents any serialization. Therefore, the history does not provide opaque-non-interference: by removing $T_2$ we can commit $T_1$ by still allowing a correct serialization $T_1$, $T_3$. But the history is permissive with respect to opacity: no transaction aborts without a reason!

This example can be used to show that opaque-non-interference is, in a strict sense, *non-implementable*. Every opaque permissive implementation that guarantees that every transactional operation (*read*, *write*, *tryCommit* or *tryAbort*) completes if it runs in the absence of concurrency (note that it can complete with an *abort* response), may be brought to the scenario above, where the only option for $T_1$ in its last event is *abort*.

*Local correctness*. But are there relaxed definitions of TM correctness that allow for non-interfering implementations? Intuitively, the problem with the history in Figure 1 is that $T_2$ should be consistent with a global order of *all* transactions. But what if we only expect every transaction $T$ to be consistent *locally*, i.e., to fit to *some* serialization composed of the transactions that committed before $T$ terminates? This way a transaction does not have to account for transactions that are aborted or incomplete at the moment it completes and local serializations for different transactions do not have to be mutually consistent.

For example, the history in Figure 1, assuming that $T_1$ commits, is still *locally* opaque: the local serialization of $T_2$ would simply be $T_3 \cdot T_2$, while $T_1$ (assuming it commits) and $T_3$ would both be consistent with the serialization $T_1 \cdot T_3$. 

---

**Fig. 1**: An opaque-permissive opaque but not opaque-non-interfering history: $T_2$ forces $T_1$ to abort
In this paper, we introduce the notion of local correctness. A history satisfies a local correctness property \( C \) if and only if all its “local sub-histories” satisfy \( C \). Here a local sub-history corresponding to \( T_i \) consists of the events from all transactions that committed before the last event of \( T_i \) (transactions that are incomplete or aborted at that moment are ignored) and: (1) if \( T_i \) is committed then all its events; (2) if \( T_i \) is aborted then all its read operations. We show that every implementation that is permissive with respect to a local correctness criterion \( C \) is also \( C \)-non-interfering.

Virtual world consistency [9], that expects the history to be strictly serializable and every transaction to be consistent with its causal past, is one example of a local correctness property. We observe, however, that virtual world consistency may allow a transaction to proceed even if it has no chances to commit. To avoid this useless work, we introduce a slightly stronger local criterion that we call local opacity. As the name suggests, a history is locally opaque if each of its local sub-histories is opaque. In contrast with VWC, a locally opaque history, a transaction may only make progress if it still has a chance to be committed.

Implementing conflict local opacity. Finally, we describe a novel TM implementation that is permissive (and, thus, non-interfering) with respect to conflict local opacity (CLO). CLO is a restriction of local opacity that additionally requires each local serialization to be consistent with the conflict order [8,13].

Our implementation is interesting in its own right for the following reasons. First, it ensures non-interference, i.e., no transaction has any effect on other transactions before committing. Second, it only requires polynomial (in the number of concurrent transactions) local computation for each transaction. Indeed, there are indications that, in general, building a permissive strictly serializable TM may incur non-polynomial time [13].

The full paper is available as a technical report [11]. Future work includes focusing on an arguably more practical notion of non-interference as an implementation property, in particular, on the inherent costs of implementing non-interference.

Roadmap. The paper is organized as follows. We describe our system model in Section 2. In Section 3 we formally define the notion of \( C \)-non-interference, recall the definition of permissiveness, and relate the two. In Section 4, we introduce the notion of local correctness, show that any permissive implementation of a local correctness criterion is also permissive, and define the criterion of conflict local opacity (CLO). In Section 5 present our CLO-non-interfering implementation. Section 6 concludes the paper with remarks on the related work and open questions. The appendix contains omitted definitions and proofs.

2 Preliminaries

We assume a system of \( n \) processes, \( p_1, \ldots, p_n \) that access a collection of objects via atomic transactions. The processes are provided with four transactional operations: the write(\( x, v \)) operation that updates object \( x \) with value \( v \), the read(\( x \)) operation that returns a value read in \( x \), tryC() that tries to commit the transaction and returns commit (c for short) or abort (a for short), and tryA() that aborts the transaction and returns A. The objects accessed by the read and write operations are called as t-objects. For the sake of presentation simplicity, we assume that the values written by all the transactions are unique.

Operations write, read and tryC() may return a, in which case we say that the operations forcefully abort. Otherwise, we say that the operation has successfully executed. Each operation specifies a unique transaction identifier. A transaction \( T_i \) starts with the first operation and completes when any of its operations returns a or c. Abort and commit operations are called terminal operations. For a transaction \( T_k \), we denote all its read operations as \( Rset(T_k) \) and write operations \( Wset(T_k) \). Collectively, we denote all the operations of a transaction \( T_k \) as \( evts(T_k) \).

Histories. A history is a sequence of events, i.e., a sequence of invocation-response pairs of transactional operations. The collection of events is denoted as \( evts(H) \). For simplicity, we only consider sequential histories here: the invocation of each transactional operation is immediately followed by a matching response. Therefore, we treat each transactional operation as one atomic event, and let \( \prec_H \) denote the total order on the transactional operations incurred by \( H \). With this assumption the only relevant events of a transaction \( T_k \) are of the types: \( r_k(x, v) \), \( r_k(x, A) \), \( w_k(x, v) \), \( w_k(x, v, A) \), \( \text{try}C_k(C) \) (or \( c_k \) for short), \( \text{try}C_k(A) \), \( \text{try}A_k(A) \) (or \( a_k \) for short). We identify a history \( H \) as tuple \( \langle evts(H), \prec_H \rangle \).
Let $H|T$ denote the history consisting of events of $T$ in $H$, and $H|p_i$ denote the history consisting of events of $p_i$ in $H$. We only consider well-formed histories here, i.e., (1) each $H|T$ consists of a read-only prefix (consisting of read operations only), followed by a write-only part (consisting of write operations only), possibly completed with a tryC or tryA operation\(^c\), and (2) each $H|p_i$ consists of a sequence of transactions, where no new transaction begins before the last transaction completes (commits or aborts).

We assume that every history has an initial committed transaction $T_0$ that initializes all the data-objects with 0. The set of transactions that appear in $H$ is denoted by txns($H$). The set of committed (resp., aborted) transactions in $H$ is denoted by committed($H$) (resp., aborted($H$)). The set of incomplete transactions in $H$ is denoted by incomplete($H$) (incomplete($H$) = txns($H$) − committed($H$) − aborted($H$)).

For a history $H$, we construct the completion of $H$, denoted $\overline{H}$, by inserting $a_k$ immediately after the last event of every transaction $T_k \in \text{incomplete}(H)$.

**Transaction orders.** For two transactions $T_k, T_m \in \text{txns}(H)$, we say that $T_k$ precedes $T_m$ in the real-time order of $H$, denote $T_k \prec_H^RT T_m$, if $T_k$ is complete in $H$ and the last event of $T_k$ precedes the first event of $T_m$ in $H$. If neither $T_k \prec_H^RT T_m$ nor $T_m \prec_H^RT T_k$, then $T_k$ and $T_m$ overlap in $H$. A history $H$ is $t$-sequential if there are no overlapping transactions in $H$, i.e., every two transactions are related by the real-time order.

**Sub-histories.** A sub-history, $SH$ of a history $H$ denoted as the tuple $(evts(SH), \prec_H)$ and is defined as: (1) $\prec_SH \subseteq \prec_H$; (2) $\text{evts}(SH) \subseteq \text{evts}(H)$; (3) If an event of a transaction $T_k \in \text{txns}(H)$ is in $SH$ then all the events of $T_k$ in $H$ should also be in $SH$. (Recall that $\prec_H$ denotes the total order of events in $H$.) For a history $H$, let $R$ be a subset of $\text{txns}(H)$, the transactions in $H$. Then $H.'subhist'$(R) denotes the sub-history of $H$ that is formed from the operations in $R$.

**Valid and legal histories.** Let $H$ be a history and $r_k(x, v)$ be a read operation in $H$. A successful read $r_k(x, v)$ (i.e., $v \neq A$), is said to be valid if there is a transaction $T_j$ in $H$ that commits before $r_k$ and $w_j(x, v)$ is in $\text{evts}(T_j)$. Formally, $(r_k(x, v))$ is valid $\Rightarrow \exists T_j : (c_j < r_k(x, v)) \land (w_j(x, v) \in \text{evts}(T_j)) \land (v \neq A)).$ The history $H$ is valid if all its successful read operations are valid.

We define $r_k(x, v)$’s lastWrite to be the latest commit event $c_i$ such that $c_i$ precedes $r_k(x, v)$ in $H$ and $x \in \text{Wset}(T_i)$ ($T_i$ can also be $T_0$). A successful read operation $r_k(x, v)$ (i.e., $v \neq A$), is said to be legal if transaction $T_i$ (which contains $r_k$’s lastWrite) also writes $v$ onto $x$. Formally, $(r_k(x, v))$ is legal $\Rightarrow (v \neq A) \land (H.'lastWrite'(r_k(x, v)) = c_i) \land (w_i(x, v) \in evts(T_i)))$. The history $H$ is legal if all its successful read operations are legal. Thus from the definitions we get that if $H$ is legal then it is also valid.

**Strict Serializability and Opacity.** We say that two histories $H$ and $H'$ are equivalent if they have the same set of events. Now a history $H$ is said to be opaque [6, 7] if $H$ is valid and there exists a $t$-sequential legal history $S$ such that (1) $S$ is equivalent to $\overline{H}$ and (2) $S$ respects $\prec_H^RT$, i.e., $\prec_H^RT \subseteq \prec_S^RT$. By requiring $S$ being equivalent to $\overline{H}$, opacity treats all the incomplete transactions as aborted.

Along the same lines, a valid history $H$ is said to be strictly serializable if $H.'subhist'(\text{committed}(H))$ is opaque. Thus, unlike opacity, strict serializability does not include aborted transactions in the global serialization order.

**3 P-Non-Interference**

A correctness criterion is a set of histories. In this section, we recall the notion of permissiveness [5] and then we formally define non-interference. First, we define a few auxiliary notions.

For a transaction $T_i$ in $H$, applicable events of $T_i$ or applicable($T_i$) denotes: (1) all the events of $T_i$, if it is committed; (2) if $T_i$ is aborted then all the read operations of $T_i$. Thus, if $T_i$ is an aborted transaction ending with tryC($A$) (and not $r_i(x, A)$ for some $x$), then the final tryC($A$) is not included in applicable($T_i$).

We denote, $H'^C$ as the shortest prefix of $H$ containing all the events of $T_i$ in $H$. Now for $T_i \in \text{aborted}(H)$, let $\mathcal{H}C$ denote the set of histories constructed from $H'^C$, where the last operation of $T_i$ in $H$ is replaced with (1) $r_i(x, v)$ for some value non-abort value $v$, if the last operation is $r_i(x, A)$, (2) $w_i(x, v, A)$, if the last operation is $w_i(x, v, A)$, (3) tryC($C$), if the last operation is tryC($A$).

\(^c\) This restriction brings no loss of generality [12].
If $R$ is a subset of transactions of $txns(H)$, then $H_{-R}$ denotes the sub-history obtained after removing all the events of $R$ from $H$. Respectively, $\mathcal{H}^{T_i,C}_{-R}$ denotes the set of histories in $\mathcal{H}^{T_i,C}$ with all the events of transaction in $R$ removed.

**Definition 1.** Given a correctness criterion $P$, we say that a history $H$ is $P$-permissive, and we write $H \in \text{Perm}(P)$ if:

1. $H \in P$;
2. $\forall T \in \text{aborted}(H), \forall H' \in \mathcal{H}^{T_i,C} : H' \notin P$.

From this definition we can see that a history $H$ is permissive w.r.t. $P$, if no aborted transaction in $H$ can be turned into committed, while preserving $P$.

The notion of non-interference or $NI(P)$ is defined in a similar manner as a set of histories parameterized by a property $P$. For a transaction $T$ in $txns(H)$, $\text{IncAbort}(T, H)$ denotes the set of transactions that have (1) either aborted before $T$’s terminal operation or (2) are incomplete when $T$ aborted. Hence, for any $T$, $\text{IncAbort}(T, H)$ is a subset of $\text{aborted}(H) \cup \text{incomplete}(H)$.

**Definition 2.** Given a correctness criterion $P$, we say that a history $H$ is $P$-non-interfering, and we write $H \in NI(P)$ if:

1. $H \in P$;
2. $\forall T \in \text{aborted}(H), R \subseteq \text{IncAbort}(T, H), \forall H' \in \mathcal{H}^{T_i,C}_{-R} : H' \notin P$.

Informally, non-interference states that none of transactions that aborted prior to or are live at the moment when $T$ aborts caused $T$ to abort: removing any subset of these transactions from the history does not help $t$ to commit. By considering the special case $R = \emptyset$ in Definition 2, we obtain Definition 1, and, thus:

**Observation 1** For every correctness criterion $P$, $NI(P) \subseteq \text{Perm}(P)$.

The example in Figure 1 (Section 1) shows that $NI(\text{opacity}) \neq \text{Perm}(\text{opacity})$ and, thus, no implementation of opacity can satisfy non-interference. This motivated us to define a new correctness criterion, a relaxation of opacity, which satisfies non-interference.

### 4 Local correctness and non-interference

Intuitively, a correctness criterion is local if is enough to ensure that, for every transaction, the corresponding local sub-history is correct. One feature of any local property $P$ is that any $P$-permissive implementation is also $P$-non-interfering.

Formally, for $T_i$ in $txns(H)$, let $\text{subC}(H, T_i)$ denote $H^{T_i}_{\text{subhist}}(\text{committed}(H^{T_i}) \cup \{\text{applicable}(T_i)\})$,

i.e., the sub-history of $H^{T_i}$ consisting of the events of all committed transactions in $H^{T_i}$ and all the applicable events of $T_i$. We call it local sub-history of $T_i$ in $H$. Note that here we are considering applicable events of $T_i$. So if $T_i$ is committed, all its events are considered. But if $T_i$ is an aborted transaction ending with $\text{tryC}(A)$ (or $r_i(x, A)$), then only its read operations are considered.

**Definition 3.** A correctness criterion $P$ is local if for all histories $H$:

$H \in P$ if and only if, for all $T_i \in txns(H)$, $\text{subC}(H, T_i) \in P$.

As we show in this section, one example of a local property is virtual world consistency [9]. Then we will introduce another local property that we call conflict local opacity ($\text{CLO}$), in the next section and describe a simple permissive $\text{CLO}$ implementation.
Theorem 2. For every local correctness property $P$, $\text{Perm}(P) \subseteq \text{NI}(P)$.

Proof. We proceed by contradiction. Assume that $H$ is in $\text{Perm}(P)$ but not in $\text{NI}(P)$. More precisely, let $T_a$ be an aborted transaction in $H$, $R \subseteq \text{IncAbort}(T_a, H)$ and $\tilde{H} \in \mathcal{H}^{{T_a,C}}_{T_a}$, such that $H \in P$.

On the other hand, since $H \in \text{Perm}(P)$, we have $\tilde{H}^{{T_a,C}} \cap P = \emptyset$. Since $P$ is local and $H \in P$, we have $\forall T_i \in \text{txns}(P)$, $\text{subC}(H, T_i) \in P$. Thus, for all transactions $T_i$ that committed before the last event of $T_a$ we have $\text{subC}(H, T_i) = \text{subC}(\tilde{H}^{{T_a,C}}, T_i) \in P$.

Now we construct $\tilde{H}$ as $\tilde{H}^{{T_a,C}}$, except that the aborted operation of $T_a$ is replaced with the last operation of $T_a$ in $\tilde{H}$. Since $\tilde{H}$ is in $P$ and $P$ is local, we have $\text{subC}(\tilde{H}, T_a) = \text{subC}(\tilde{H}, T_a) \in P$. For all transactions $T_i$ that committed before the last event of $T_a$ in $\tilde{H}$, we have $\text{subC}(\tilde{H}, T_i) = \text{subC}(\tilde{H}^{{T_a,C}}, T_i) \in P$. Hence, since $P$ is local, we have $\tilde{H} \in P$. But, by construction, $\tilde{H} \in \mathcal{H}^{{T_a,C}}$—a contradiction with the assumption that $\tilde{H}^{{T_a,C}} \cap P = \emptyset$. \hfill $\square$

As we observed earlier, for any correctness criterion $P$, $\text{NI}(P) \subseteq \text{Perm}(P)$. Hence, Theorem 2 implies that for any local correctness criterion $P \text{ NI}(P) = \text{Perm}(P)$.

4.1 Virtual world consistency

The correctness criterion of virtual world consistency (VWC) [9] relaxes opacity by allowing aborted transactions to be only consistent with its local causal past. More precisely, we say that $T_i$ causally precedes $T_j$ in a history $H$, and we write $T_i \prec^C_H T_j$, if one of the following conditions hold: (1) $T_i$ and $T_j$ are executed by the same process and $T_i \prec^R_H T_j$, (2) $T_i$ commits and $T_j$ reads the value written by $T_i$ to some object $x \in \text{Wset}(T_i) \cap \text{Rset}(T_j)$ (recall that we assumed for simplicity that all written values are unique), or (3) there exists $T_k$, such that $T_i \sim^C_H T_k$ and $T_k \sim^C_H T_j$. The set of transactions $T_i$ such that $T_i \prec^C_H T_j$ and $T_j$ itself is called the causal past of $T_j$, denoted $CP(T_j)$.

Now $H$ is in $\text{VWC}$ if (1) $H.\text{subhist}(\text{committed})$ is opaque and (2) for every $T_i \in \text{txns}(H)$, $H.\text{subhist}(CP(T_i))$ is opaque. Informally, $H$ must be strictly serializable and the causal past of every transaction in $H$ must constitute an opaque history.

It is easy to see that $H \in \text{VWC}$ if and only if for all $\text{subC}(H, T_i) \in \text{VWC}$. By Theorem 2, any VWC-permissive implementation is also VWC-non-interfering.

4.2 Conflict local opacity

As shown in [9], the VWC criterion may allow a transaction to proceed if it is “doomed” to abort: as long as the transaction’s causal past can be properly serialized, the transaction may continue if it is no more consistent with the global serial order and, thus, will have to eventually abort. We propose below a stronger local property that, intuitively, aborts a transaction as soon as it cannot be put in a global serialization order.

Definition 4. A history $H$ is said to be locally opaque or LO, if for each transaction $T_i$ in $H$: $\text{subC}(H, T_i)$ is opaque.

It is immediate from the definition that a locally opaque history is strictly serializable: simply take $T_i$ above to be the last transaction to commit in $H$. The resulting $\text{subC}(H, T_i)$ is going to be $H.\text{subhist}(\text{committed}(H))$, the sub-history consisting of all committed transactions in $H$. Also, one can easily see that local opacity is indeed a local property.

Every opaque history is also locally opaque, but not vice versa. To see this, consider the history $H$ in Figure 2 which is like the history in Figure 1, except that transaction $T_1$ is now committed. Notice that the history is not opaque anymore: $T_1$, $T_2$ and $T_3$ form a cycle that prevents any legal serialization. But it is locally opaque: each transaction witnesses a state which is consistent with some legal total order on transactions committed so far: $\text{subC}(H, T_1)$ is equivalent to $T_3T_1$, $\text{subC}(H, T_2)$ is equivalent to $T_3T_2$, $\text{subC}(H, T_3)$ is equivalent to $T_3$.  

6
We denote the set of locally opaque histories by $LO$. Finally, we propose a restriction of local opacity that ensures that every local serialization respects the conflict order [17, Chap. 3]. For two transactions $T_k$ and $T_m$ in $txns(H)$, we say that $T_k$ precedes $T_m$ in conflict order, denoted $T_k \prec_{CO} T_m$, if (w-w order) $tryC_k(C) <_H tryC_m(C)$ and $Wset(T_k) \cap Wset(T_m) \neq \emptyset$, (w-r order) $tryC_k(C) <_H r_m(x, v)$, $x \in Wset(T_k)$ and $v \neq A$, or (r-w order) $r_k(x, v) <_H tryC_m(C)$, $x \in Wset(T_m)$ and $v \neq A$. Thus, it can be seen that the conflict order is defined only on operations that have successfully executed. Using conflict order, we define a subclass of opacity, conflict opacity (co-opacity).

**Definition 5.** A history $H$ is said to be conflict opaque or co-opaque if $H$ is valid and there exists a t-sequential legal history $S$ such that (1) $S$ is equivalent to $H$ and (2) $S$ respects $\prec_{RT} H$ and $\prec_{CO} H$.

Now we define a “conflict” restriction of local opacity, conflict local opacity (CLO) by replacing opaque with co-opaque in Definition 4. Immediately, we derive that co-opacity is a subset of opacity and CLO is a subset of $LO$.

## 5 Implementing Local Opacity

In this section, we present our permissive implementation of CLO. By Theorem 2 it is also CLO-non-interfering. Our implementation is based on conflict-graph construction of co-opacity, a popular technique borrowed from databases (cf. [17, Chap. 3]). We then describe a simple garbage-collection optimization that prevents the memory used by the algorithm from growing without bound.

### 5.1 Graph characterization of co-opacity

Given a history $H$, we construct a conflict graph, $CG(H) = (V, E)$ as follows: (1) $V = txns(H)$, the set of transactions in $H$ (2) an edge $(T_i, T_j)$ is added to $E$ whenever $T_i \prec_{RT} T_j$ or $T_i \prec_{CO} T_j$, i.e., whenever $T_i$ precedes $T_j$ in the real-time or conflict order.

From this characterization, we get the following theorem:

**Theorem 3.** A legal history $H$ is co-opaque iff $CG(H)$ is acyclic.

### 5.2 The Algorithm for Implementing CLO

Our CLO implementation is presented in Algorithms 1, 2 and 3 (we omit the trivial implementation of $tryA$ here). The main idea is that the system maintains a sub-history of all the committed transactions. Whenever a live transaction $T_i$ wishes to perform an operation $o_i$ (read, write or commit), the TM system checks to see if $o_i$ and the transactions that committed before it, form a cycle. If so, $o_i$ is not permitted to execute and $T_i$ is aborted. Otherwise, the operation is allowed to execute. Similar algorithm(s) called as serialization graph testing have been proposed for databases (cf. [17, Chap. 4]). Hence, we call it SGT algorithm.

Our SGT algorithm maintains several variables. Some of them are global to all transactions which are prefixed with the letter ‘g’. The remaining variables are local. The variables are:
Algorithm 1 Read of a t-object $x$ by a transaction $T_i$

1: procedure read$_i(x)$
2: // read $gComHist$
3: $tHist_i = gComHist$; // create a local copy of $gComHist$
4: // create $v$, to store the value of $x$
5: $v = \text{the latest value written to } x \text{ in } tHist_i$;
6: // create $lseq_i$, the local copy of $gseqn$
7: $lseq_i = \text{the value of largest seq. no. of a transaction in } lComHist_i$;
8: create the readVar rop$_i(x, v, lseq_i)$;
9: // update $lComHist_i$
10: $lComHist_i = \text{merge } lComHist_i \text{ and } tHist_i$; append rop$_i(x, v, lseq_i)$ to $lComHist_i$;
11: // check for consistency of the read operation
12: if $(CG(lComHist_i) \text{ is cyclic})$ then
13: replace rop$_i(x, v, lseq_i)$ with $(rop_i(x, A, lseq_i) \text{ in } lComHist_i)$;
14: return abort;
15: end if
16: // current read is consistent; hence store it in the read set and return $v$
17: return $v$;
18: end procedure

Algorithm 2 Write of a t-object $x$ with value $v$ by a transaction $T_i$

1: procedure write$_i(x, v)$
2: if write$_i(x, v)$ is the first operation in $T_i$ then
3: // read $gComHist$
4: $lComHist_i = gComHist$;
5: $lseq_i = \text{the value of largest seq. no. of a transaction in } lComHist_i$;
6: end if
7: create the writeVar wop$_i(x, v, lseq_i)$;
8: append wop$_i(x, v, lseq_i)$ to $lComHist_i$;
9: return ok;
10: end procedure

Algorithm 3 TryCommit operation by a transaction $T_i$

1: procedure tryC$_i$
2: lock $gLock$;
3: // create the next version of $gseqn$ for the current $T_i$
4: $lseq_i = gSeqNum + 1$;
5: $tHist_i = gComHist$; // create a local copy of $gComHist$
6: $lComHist_i = \text{merge } lComHist_i \text{ and } tHist_i$; // update $lComHist_i$
7: // create the commit operation with $lseq_i$
8: create the comVar cop$_i(lseq_i)$;
9: append cop$_i(lseq_i)$ to $lComHist_i$;
10: if $(CG(lComHist_i) \text{ is cyclic})$ then
11: Replace cop$_i(lseq_i)$ with $a_i$ in $lComHist_i$;
12: Release the lock on $gLock$;
13: return abort;
14: end if
15: $gComHist = lComHist_i$;
16: $gSeqNum = lseq_i$;
17: Release the lock on $gLock$;
18: return commit;
19: end procedure
- \textit{gSeqNum}, initialized to 0 in the start of the system: global variable that counts the number of transactions committed so far.
- \textit{lseq}: a transaction-specific variable that contains the number of transactions currently observed committed by \(T_i\). When a transaction \(T_i\) commits, the current value of \(gSeqNum\) is incremented and assigned to \(lseq\).
- \textit{readVar}: captures a read operation \(r_i\) performed by a transaction \(T_i\). It stores the variable \(x\), the value \(v\) returned by \(r_i\), and the sequence number \(s\) of \(r_i\), computed as the sequence number of the committed transaction \(r_i\) reads from. We use the notation \(\text{rop}_i(x, v, s)\) to denote the read operation in the local or global history.
- \textit{writeVar}: captures a write operation \(w_i(x, v)\) performed by a transaction \(T_i\). It stores the variable \(x\), the value written by the write operation \(v\) and the sequence number \(s\) of \(w_i\), computed as the sequence number of the previous op in \(T_i\) or the sequence number of the last committed transaction preceding \(T_i\) if \(w_i\) is the first operation in \(T_i\). We use the notation \(\text{wop}_i(x, v, s)\) to denote the write operation.
- \textit{comVar}: captures a commit operation of a transaction \(T_i\). It stores the \(lseq\) of the transaction. We use the notation \(\text{cop}_i(s)\) to denote the comVar operation where \(s\) is the \(lseq\) of the transaction.
- \textit{gComHist}: captures the history of events of committed transactions. It is a list of \(\text{readVar, writeVar, comVar}\) variables ordered by real-time execution. We assume that \(gComHist\) also contains initial values for all t-variables (later updates of these initial values will be used for garbage collection).
- \textit{gLock}: This is a global lock variable. The TM system locks this variable whenever it wishes to read and write to any global variable.

The implementations of \(T_i\)'s operations, denoted by \(\text{read}_i(x), \text{write}_i(x, v)\) and \(\text{tryC}_i()\) are described below. We assume here that if any of these is the first operation performed by \(T_i\), it is preceded with the initialization all \(T_i\)'s local variables.

We also assume that all the t-objects accessed by the STM system are initialized with 0 (which simulates the effect of having the initializing transaction \(T_0\)).

\textit{read}_i(x): Every transaction \(T_i\) maintains \(lComHist_i\) which is a local copy \(gComHist\) combined with events of \(T_i\) taken place so far, put at the right places in \(gComHist\), based on their sequence numbers. From \(lComHist_i\) the values \(v\) and \(lseq_i\) are computed. If there are no committed writes operation on \(x\) preceding \(\text{read}_i(x)\) in \(lComHist_i\), then \(v\) is assumed to be the initial value 0. Then, a \(\text{readVar rop}_i\) is created for the current read operation using the latest value of \(x, v\) and the current value of \(gSeqNum\), \(lseq_i\). Then \(\text{rop}_i\) is inserted into \(lComHist_i\). A conflict graph is constructed from the resulting \(lComHist_i\) and checked for acyclicity. If the graph is cyclic then \(A\) is inserted into \(\text{rop}_i\) of \(lComHist_i\) and then \text{abort} is returned. Otherwise, the value \(v\) is returned.

\textit{write}_i(x, v): adds a \textit{writeVar} containing \(x\) and \(v\) and \(lseq_i\) is inserted to \(lComHist_i\). (If the write is the first operation of \(T_i\), then \(lComHist_i\) and \(lseq_i\) are computed based on the current state of \(gComHist_i\).)

\textit{tryC}_i(x): The main idea for this procedure is similar to \textit{read}_i, except that the TM system first obtains the lock on \(gLock\). Then it makes local copies of \(gSeqNum, gComHist\) which are \(lseq_i, tHist_i,\) and \(lComHist_i\). The value \(lseq_i\) is incremented, and the \(\text{cop}_i(lseq_i)\) item is appended to \(lComHist_i\). Then a conflict graph is constructed for the resulting \(lComHist_i\) and checked for acyclicity. If the graph is cyclic then \(\text{cop}_i(seq_i)\) is replaced with \(a_i\) in \(lComHist_i\), the lock is released and \text{abort} is returned. Otherwise, \(lseq_i, lComHist_i\), are copied back into \(gSeqNum, gComHist\), the lock is released and \text{ok} is returned.

5.3 Correctness of SGT

In this section, we will prove that our implementation is permissive w.r.t. CLO. Consider the history \(H\) generated by SGT algorithm. Recall that only read, \textit{tryC} and write operation (if it is the first operation in a transaction) access shared memory. Hence, we call such operations \textit{memory operations}.

Note that \(H\) is not necessarily sequential: the transactional operations can execute in overlapping manner. Therefore, to reason about correctness, we first order all the operations in \(H\) to get an equivalent sequential history. We then show that this sequential history is permissive with respect to CLO.
We place the memory operations, say \( r_i(x, v/A) \), \( \text{try}C_i(C/A) \) based on the order in which they access the global variable \( g\text{ComHist} \), storing the history of currently committed transactions. The remaining write operations are placed anywhere between the last preceding memory operation and its \( \text{try}C_i \) operation. We denote the resulting history, completed by adding \( \text{try}C_i(A) \) operation for every incomplete transaction \( T_i \), by \( H_g \). It can be seen that \( H_g \) represents a complete sequential history that respects the real time ordering of memory operations in \( H \). In the rest of this section, we show that \( H_g \) is permissive (and, thus, non-interfering) with respect to \( CLO \).

Since \( CLO \) is local, to show that \( H_g \) is in \( CLO \), it is sufficient to show that, for each transaction \( T_i \) in \( \text{txns}(H_g) \), \( \text{sub}(H_g, T_i) \) is in \( CLO \). We denote \( \text{sub}(H_g, T_i) \) by \( H_{ig} \).

Consider a transaction \( T_i \in \text{txns}(H_g) \). Consider the last complete memory operation of \( T_i \) in \( H \), denoted as \( m_i \). Note that every \( T_i \) performs at least one successful memory operation (the proof for the remaining case is trivial). We define a history \( H_{im} \) as the local history \( i\text{ComHist}_i \) computed by SGT with the last complete memory operation of \( T_i \) in \( H \) (line 10 of Algorithm 1 and line 9 of Algorithm 3).

**Lemma 1.** \( \text{H}_{im} \) and \( \text{H}_{ig} \) are equivalent.

**Proof.** Obviously, \( \text{H}_{im} \) and \( \text{H}_{ig} \) agree on the events of \( T_i \). The SGT algorithm assigns commitSeqNum (a sequence number) to each committed transaction \( T_j \). Similarly it also assigns readSeqNum to each successfully completed read operation, i.e. the read that did not return abort. Based on these sequence numbers, the SGT algorithm constructs \( \text{H}_{im} \) (line 10 of Algorithm 1 and line 9 of Algorithm 3) of all the events that committed before the last successful memory operation of \( T_i \) in \( H_g \). On the other hand, every event that appears in \( \text{H}_{im} \) belongs to \( T_i \) or a transaction that committed before the last successful memory operation of \( T_i \) in \( H_g \). Thus, \( \text{H}_{im} \) and \( \text{H}_{ig} \) are equivalent.

Even though \( \text{H}_{im} \) and \( \text{H}_{ig} \) are equivalent, the ordering of the events in these histories could be different. However, the two histories agree on the real-time and conflict orders of transactions.

**Lemma 2.** \( \prec^{CO}_{\text{H}_{im}} = \prec^{CO}_{\text{H}_{ig}} \) and \( \prec^{RT}_{\text{H}_{ig}} = \prec^{RT}_{\text{H}_{im}} \).

**Proof.** We go case by case for each possible relation in \( \prec^{CO} \cup \prec^{RT} \).

**Write-write order:** we want to show that \( (\text{try}C_p <_{im} \text{try}C_q) \Leftrightarrow (T_p.\text{commitSeqNum} < T_q.\text{commitSeqNum}) \Leftrightarrow (\text{try}C_p <_{ig} \text{try}C_q) \).

The result \( (\text{try}C_p <_{im} \text{try}C_q) \Leftrightarrow (T_p.\text{commitSeqNum} < T_q.\text{commitSeqNum}) \Leftrightarrow (\text{try}C_p <_{ig} \text{try}C_q) \).

**Write-read order:** For a committed transactions \( T_p \) and a successful read operation \( q_r \), we want to show that \( (\text{try}C_p <_{im} q_r) \Leftrightarrow (T_p.\text{commitSeqNum} < q_r.\text{readSeqNum}) \Leftrightarrow (\text{try}C_p <_{ig} q_r) \).

The result \( (\text{try}C_p <_{im} q_r) \Leftrightarrow (T_p.\text{commitSeqNum} < q_r.\text{readSeqNum}) \Leftrightarrow (\text{try}C_p <_{ig} q_r) \).

The SGT algorithm stores as a part of the read operation \( r_j \), readSeqNum which is same as the commitSeqNum of the latest transaction that committed before \( r_j \), say \( T_i \). Thus \( T_i.\text{commitSeqNum} = r_j.\text{readSeqNum} \). From the above argument for the write-write order, we have that any transaction \( T_k \) that committed before \( T_i \) will have lower commitSeqNum. This holds in \( H_g \) and as a result also holds in \( H_{ig} \). This shows that \( (T_p.\text{commitSeqNum} < q_r.\text{readSeqNum}) \Leftrightarrow (\text{try}C_p <_{ig} q_r) \).

**Read-write order:** For a committed transactions \( T_q \) and a successful read operation \( r_p \), we want to show that \( (r_p <_{im} \text{try}C_q) \Leftrightarrow (r_p.\text{readSeqNum} < T_q.\text{commitSeqNum}) \Leftrightarrow (r_p <_{ig} \text{try}C_q) \).

The reasoning is similar to the above cases.

Hence, \( \prec^{CO}_{\text{H}_{im}} = \prec^{CO}_{\text{H}_{ig}} \).

**Real-time order:** Consider two transaction \( T_p, T_q \) in \( H_{ig} \) such that \( T_p \prec^{RT}_{\text{H}_{ig}} T_q \) which also holds in \( H_{ig} \). From the construction of \( H_{ig} \), we get that \( T_p \) is a committed transaction with its last event being \( \text{try}C_p \). Indeed, the only possibly uncommitted transaction in \( H_{ig} \) is \( T_i \) that performs the last event in \( H_{ig} \) and, thus, cannot precede any transaction in \( \prec^{RT}_{\text{H}_{ig}} \).
Consider the first memory operation of \( T_q \) (by our assumption, there is one in each \( T_q \) in \( H_{ig} \)). By the algorithm, the sequence number associated with the memory operation is at least as high as the sequence number of \( \text{try}C \). Thus \( T_p \prec_{RT} T_q \). The other direction is analogous.

Hence, \( \prec_{RT}^{H_{im}} = \prec_{RT}^{H_{ig}} \).

Lemmas 1 and 2 imply that \( H_{im} \) and \( H_{ig} \) generate the same conflict graph:

**Corollary 1.** \( CG(H_{ig}) = CG(H_{im}) \)

Now we argue about legality of \( H_{im} \) and \( H_{ig} \).

**Lemma 3.** \( H_{ig} \) is legal.

**Proof.** By the algorithm, every successful read operation on a variable \( x \) within \( T_i \) returns the argument of the last committed write on \( x \) that appears in \( \text{IComHist}_i \) (and, thus, in \( H_{im} \)). By applying this argument to every prefix of \( H_{im} \), we derive that \( H_{im} \) is legal. By Lemmas 5 and 2, we derive that \( H_{ig} \) is also legal. \( \square \)

**Theorem 4.** Let \( H_g \) be a history generated by the SGT algorithm. Then \( H_g \) is in \( CLO \).

**Proof.** By the algorithm, the corresponding \( H_{im} \) produces an acyclic conflict graph \( CG(H_{im}) \) (cf. checks in line 12 of Algorithm 1 and line 10 of Algorithm 3). By Corollary 1, \( CG(H_{ig}) \) is also acyclic.

Thus, by Theorem 3 and Lemma 3, for every \( T_i \in \text{txns}(H_g) \), \( H_{ig} \) is co-opaque. Since \( CLO \) is a local property, we derive that \( H_g \) is in \( CLO \). \( \square \)

Having proved that SGT algorithm generates \( CLO \) histories, we now show that SGT algorithm is in fact permissive w.r.t. \( CLO \).

**Theorem 5.** Let \( H_g \) be a history generated by SGT algorithm. Then \( H_g \) is in \( \text{Perm}(CLO) \).

**Proof.** We shall prove this by contradiction. Assume that \( H_g \) is not in \( \text{Perm}(CLO) \). From Theorem 4, we get that \( H_g \) is in \( CLO \). Hence, condition (2) of Definition 1 is not true. Thus, there is an aborted transaction \( T_a \) in \( H_g \) which can be committed so that the resulting history is still in \( CLO \). We denote the modified transaction as \( T_a^C \) and the resulting history as \( H_g' \). There are two cases depending on the final event of \( T_a \):

*Case 1: The last event of \( T_a \) is a read operation \( r_a(x, A) \). In order for \( T_a^C \) to be committed in \( H_g' \), \( r_a(x, A) \) is converted to \( r_a(x, v) \) for some \( v \). If \( H_g' \) is in \( CLO \), then \( subC(H_g', T_a^C) \) is co-opaque. By Corollary 2, we get that \( subC(H_g', T_a^C) \) is legal. Therefore, \( v \) is the value written by the transaction committing \( r_a \)'s lastWrite in \( H_g' \) (the current value on \( v \)). It can be seen that \( H_g' \) differs from \( H_g \) only in \( r_a \).

But when SGT algorithm attempts to read this value of \( x \) in line 10 of Algorithm 1, it causes the conflict graph maintained to be cyclic. From Corollary 1 applied to \( H_g' \), we get that the conflict graph of \( subC(H_g', T_a^C) \) is also cyclic. By Theorem 3, we derive that \( subC(H_g', T_a^C) \) is not co-opaque. This implies that \( H_g' \) is not in \( CLO \)—a contradiction.

*Case 2: The last event of \( T_a \) is an abort operation \( \text{try}C_a(A) \). The argument in this case is similar to the above case. In order for \( T_a^C \) to be committed in \( H_g' \), \( \text{try}C_a(A) \) is converted into \( \text{try}C_a(C) \). When SGT algorithm attempts to commit \( T_a \) in line 9 of Algorithm 3, it causes the conflict graph maintained to be cyclic. By Corollary 1 applied to \( H_g' \), we derive that the conflict graph of \( subC(H_g', T_a^C) \) is also cyclic. From Theorem 3, we then get that \( subC(H_g', T_a^C) \) is not co-opaque. This implies that \( H_g' \) is not in \( CLO \) and hence again a contradiction.

Therefore, no transaction \( T_a \) in \( H_g \) can not be transformed into a committed transaction \( T_a^C \) while still staying in \( CLO \). Hence, \( H_g \) is in \( \text{Perm}(CLO) \). \( \square \)

It is left to show that our algorithm is live, i.e., under certain conditions, every operation eventually completes.

**Theorem 6.** Assuming that no transaction fails while executing the \( \text{try}C \) operation and \( g\text{Lock} \) is starvation-free, every operation of SGT eventually returns.
Proof. It can be seen that read and write functions do not involve any waiting. Therefore, tryC is the only function which involves waiting for the gLock variable. But since the lock is starvation-free and no transaction executing tryC obtains the lock forever, every such waiting is finite. Thus, every tryC operation eventually grabs the lock and, after, computing the outcome, returns. 

**Theorem 7.** Let \( H_g \) be a history generated by the SGT algorithm. Then \( H_g \) is in CLO.

**Theorem 8.** Let \( H_g \) be a history generated by SGT algorithm. Then \( H_g \) is in Perm(CLO).

Now Theorem 2 implies that our SGT implementation is CLO-non-interfering.

**Theorem 9.** Assuming that no transaction fails while executing the tryC operation and gLock is starvation-free, every operation of SGT eventually returns.

### 5.4 Garbage Collection

Over time, the history of committed transactions maintained by our SGT algorithm in the global variable gComHist grows without bound. We now describe a simple garbage-collection scheme that allows to keep the size of gComHist proportional to the current contention, i.e., to the number of concurrently live transactions. The idea is to periodically remove from gComHist the sub-histories corresponding to committed transactions that become obsolete, i.e., the effect of them can be reduced to the updates of t-objects.

More precisely, a transaction \( T_i \)'s liveSet is the set of the transactions that were incomplete when \( T_i \) terminated. A t-complete transaction \( T_i \) is said to be obsolete (in a history \( H \)) if all the transactions in its liveSet have terminated (in \( H \)).

To make sure that obsolete transactions can be correctly identified based on the global history gComHist, we update our algorithm as follows. When a transaction performs its first operation, it grabs the lock on gComHist and inserts the operation in it. Now when a transaction commits it takes care of all committed transactions in gComHist which have become obsolete. All read operations preceding the last event of an obsolete transaction are removed. In case there are multiple obsolete transactions writing to the same t-object, only the writes of the last such obsolete transaction to commit are kept in the history. If an obsolete transaction is not the latest to commit an update on any t-object, all events of this transactions are removed.

In other words, \( H_{im} \) defined as the local history lComHist, computed by SGT within the last complete memory operation of \( T_i \) in the updated algorithm (which corresponds to line 10 of Algorithm 1 and line 9 of Algorithm 3) preserves write and commit events of the latest obsolete transaction to commit a value for every t-object. All other events of other obsolete transactions are removed. The computed history \( H_{im} \) is written back to gComHist in line 15 of Algorithm 1.

Let this gComHist be used by a transaction \( T_i \) in checking the correctness of the current local history (line 12 of Algorithm 1 or line 10 of Algorithm 3). Recall that \( H_{ig} \) denotes the corresponding local history of \( T_i \). Let \( T_i \) be any obsolete transaction in \( H_{ig} \). Note that all transactions that committed before \( T_i \) in \( H_{ig} \) are also obsolete in \( H_{ig} \), and let \( U \) denote the set of all these obsolete transactions, including \( T_i \). Respectively, let \( obs(H_{ig}, U) \) be a prefix of \( H_{ig} \) in which all transactions in liveSet(\( T_i \)) are complete. Also, let \( trim(H_{ig}, U) \) be the “trimmed” local history of \( T_i \) where all transactions in \( U \) are removed or replaced with committed updates, as described above. We can show that \( H_{ig} \) is in CLO if and only if \( obs(H_{ig}, U) \) and \( trim(H_{ig}, U) \) are in CLO.

Iteratively, for each \( T_i \), all our earlier claims on the relation between the actual local history \( H_{ig} \) and the locally constructed history \( H_{im} \) hold now for the “trimmed” history \( trim(H_{ig}, U) \) and \( H_{im} \). Therefore, \( H_{im} \) is in CLO if and only if \( H_{ig} \) is in CLO. Hence, every history \( H_g \) generated by the updated algorithm with garbage collection is CLO-permissive (and, thus, CLO-non-interfering).

Note that removing obsolete transactions from gComHist essentially boils down to dropping a prefix of it that is not concurrent to any live transactions. As a result, the length of gComHist is \( O(M + C) \), where \( M \) is the number of t-objects and \( C \) is the upper bound on the number of concurrent transactions. A complete correctness proof for the optimized algorithm is given in [11].
6 Concluding remarks

In this paper, we explored the notion of non-interference in transactional memory, originally highlighted in [14,15]. We focused on P-non-interference that grasps the intuition that no transaction aborts because of aborted or incomplete transactions in the sense that by removing some of aborted or incomplete transactions we cannot turn a previously aborted transaction into a committed one without violating the given correctness criterion $P$. We showed that no TM implementation can provide opacity-non-interference. However, we observed that any permissive implementation of a local correctness criterion is also non-interfering. Informally, showing that a history is locally correct is equivalent to showing that every its local sub-history is correct.

We discussed two local criteria: virtual-world consistency (VWC) [9] and the (novel) local opacity (LO). Unlike VWC, LO does not allow a transaction that is doomed to abort to waste system resources. TMS1 [?] was recently proposed as a candidate for the “weakest reasonable” TM correctness criterion. Interestingly, at least for the case of atomic transactional operations, LO seems to coincide with TMS1.

We then considered CLO, a restriction of LO that, in addition, requires every local serialization to respect the conflict order [8,13] of the original sub-history. We presented a permissive, and thus non-interfering, CLO implementation. This appears to be the only non-trivial permissive implementation known so far (the VWC implementation in [2] is only probabilistically permissive).

Our definitions and our implementation intend to build a “proof of concept” for non-interference and are, by intention, as simple as possible (but not simpler). Of course, interesting directions are to consider a more realistic notion of non-interference as a characteristics of an implementation, to extend our definitions to non-sequential histories, and to relax the strong ordering requirements in our correctness criteria. Indeed, the use of the conflict order allowed us to efficiently relate correctness of a given history to the absence of cycles in its graph characterization. Respecting conflict order makes a lot of sense if we aim at permissiveness, as efficient verification of strict serializability or opacity appear elusive [13]. But it may be too strong as a requirement for less demanding implementations.

Also, our implementation is quite simplistic in the sense that it uses one global lock to protect the history of committed transactions and, thus, it is not disjoint-access-parallel (DAP) [1,10]. An interesting challenge is to check if it is possible to construct a permissive DAP CLO implementation with invisible reads.

References

1. H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures, SPAA ’09, pages 69–78, New York, NY, USA, 2009. ACM.
2. T. Crain, D. Imbs, and M. Raynal. Read invisibility, virtual world consistency and probabilistic permissiveness are compatible. In ICA3PP (1), pages 244–257, 2011.
3. L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In PPOPP, pages 67–78, 2010.
4. P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. IEEE Trans. Parallel Distrib. Syst., 21(12):1793–1807, 2010.
5. R. Guerraoui, T. Henzinger, and V. Singh. Permissiveness in transactional memories. In DISC ’08: Proc. 22nd International Symposium on Distributed Computing, pages 305–319, sep 2008. Springer-Verlag Lecture Notes in Computer Science volume 5218.
6. R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming, pages 175–184, New York, NY, USA, 2008. ACM.
7. R. Guerraoui and M. Kapalka. Principles of Transactional Memory, Synthesis Lectures on Distributed Computing Theory. Morgan and Claypool, 2010.
8. V. Hadzilacos. A theory of reliability in database systems. J. ACM, 35(1):121–145, Jan. 1988.
9. D. Imbs and M. Raynal. A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition. In Proceedings of the 16th international conference on Structural Information and Communication Complexity, SIROCCO’09, pages 266–280, Berlin, Heidelberg, 2010. Springer-Verlag.
10. A. Israeli and L. Rappoport. Disjoint-access-parallel implementations of strong shared memory primitives. In Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing, PODC ’94, pages 151–160, New York, NY, USA, 1994. ACM.
11. P. Kuznetsov and S. Peri. Non-interference and locality in transactional memory. CoRR, abs/1211.6315, 2012.
12. P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In OPODIS, pages 112–127, 2011.
13. C. H. Papadimitriou. The serializability of concurrent database updates. J. ACM, 26(4):631–653, 1979.
14. S. Peri and K. Vidyasankar. Correctness of concurrent executions of closed nested transactions in transactional memory systems. In 12th International Conference on Distributed Computing and Networking, pages 95–106, 2011.
15. S. Peri and K. Vidyasankar. An efficient scheduler for closed nested transactions that satisfies all-read-consistency and non-interference. In 13th International Conference on Distributed Computing and Networking, 2012.
16. M. F. Spear, M. M. Michael, and C. von Praun. Ringstm: scalable transactions with a single atomic instruction. In Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures, pages 275–284, 2008.
17. G. Weikum and G. Vossen. Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery. Morgan Kaufmann, 2002.

A Appendix

A.1 Graph characterization of co-opacity

In the following lemmas, we show that the graph characterization indeed helps us verify the membership in co-opacity. Note, since \( \text{txns}(H) = \text{txns}(\overline{H}) \) and \((\sim_{\overline{H}}^{RT} \cup \sim_{\overline{H}}^{CO}) = (\sim_{RT}^{RT} \cup \sim_{\overline{RT}}^{CO}) \), we have \( CG(H) = CG(\overline{H}) \).

Lemma 4. Consider two histories \( H_1 \) and \( H_2 \) such that \( H_1 \) is equivalent to \( H_2 \) and \( H_1 \) respects conflict order of \( H_2 \), i.e., \( \prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO} \). Then, \( \prec_{H_1}^{CO} = \prec_{H_2}^{CO} \).

Proof. Here, we have that \( \prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO} \). In order to prove \( \prec_{H_1}^{CO} = \prec_{H_2}^{CO} \), we have to show that \( \prec_{H_1}^{CO} \subseteq \prec_{H_2}^{CO} \). We prove this using contradiction. Consider two events \( p, q \) belonging to transaction \( T_1, T_2 \) respectively in \( H_2 \) such that \( (p, q) \in \prec_{H_2}^{CO} \) but \( (p, q) \notin \prec_{H_1}^{CO} \). Since the events of \( H_2 \) and \( H_1 \) are same, these events are also in \( H_1 \). This implies that the events \( p, q \) are also related by \( CO \) in \( H_1 \). Thus, we have that either \( (p, q) \in \prec_{H_1}^{CO} \) or \( (q, p) \in \prec_{H_1}^{CO} \). But from our assumption, we get that the former is not possible. Hence, we get that \( (q, p) \in \prec_{H_1}^{CO} \) \( \Rightarrow (q, p) \in \prec_{H_2}^{CO} \). But we already have that \( (p, q) \in \prec_{H_2}^{CO} \). This is a contradiction. \( \square \)

Lemma 5. Let \( H_1 \) and \( H_2 \) be equivalent histories such that \( \prec_{H_1}^{CO} = \prec_{H_2}^{CO} \). Then \( H_1 \) is legal iff \( H_2 \) is legal.

Proof. It is enough to prove the ‘if’ case, and the ‘only if’ case will follow from symmetry of the argument. Suppose that \( H_1 \) is legal. By contradiction, assume that \( H_2 \) is not legal, i.e., there is a read operation \( r_j(x, v) \) (of transaction \( T_j \)) in \( H_2 \) with lastWrite as \( c_k \) (of transaction \( T_k \)) and \( T_k \) writes \( u \neq v \) to \( x \), i.e. \( w_k(x, u) \in \text{evts}(T_k) \). Let \( r_j(x, v) \)'s lastWrite in \( H_1 \) be \( c_i \) of \( T_i \). Since \( H_1 \) is legal, \( T_i \) writes \( v \) to \( x \), i.e \( w_i(x, v) \in \text{evts}(T_i) \).

Since \( \text{evts}(H_1) = \text{evts}(H_2) \), we get that \( c_i \) is also in \( H_2 \), and \( c_k \) is also in \( H_1 \). As \( \prec_{H_1}^{CO} = \prec_{H_2}^{CO} \), we get \( c_i <_{H_2} r_j(x, v) \) and \( c_k <_{H_1} r_j(x, v) \).

Since \( c_i \) is the lastWrite of \( r_j(x, v) \) in \( H_1 \) we derive that \( c_k <_{H_1} c_i \) and, thus, \( c_k <_{H_2} c_i <_{H_2} r_j(x, v) \). But this contradicts the assumption that \( c_k \) is the lastWrite of \( r_j(x, v) \) in \( H_2 \). Hence, \( H_2 \) is legal. \( \square \)

From the above lemma we get the following interesting corollary.

Corollary 2. Every co-opaque history \( H \) is legal as well.

Based on the conflict graph construction, we have the following graph characterization for co-opaque.

Theorem 10. A legal history \( H \) is co-opaque iff \( CG(H) \) is acyclic.
Proof. (Only if) If \( H \) is co-opaque and legal, then \( CG(H) \) is acyclic: Since \( H \) is co-opaque, there exists a legal t-sequential history \( S \) equivalent to \( \overline{H} \) and \( S \) respects \( \prec_{RT}^{\overline{H}} \) and \( \prec_{CO}^{\overline{H}} \). Thus from the conflict graph construction we have that \( CG(\overline{H}) (= CG(H)) \) is a sub graph of \( CG(S) \). Since \( S \) is sequential, it can be inferred that \( CG(S) \) is acyclic. Any sub graph of an acyclic graph is also acyclic. Hence \( CG(H) \) is also acyclic.

(i) If \( H \) is legal and \( CG(H) \) is acyclic then \( H \) is co-opaque: Suppose that \( CG(H) = CG(\overline{H}) \) is acyclic. Thus we can perform a topological sort on the vertices of the graph and obtain a sequential order. Using this order, we can obtain a sequential schedule \( S \) that is equivalent to \( \overline{H} \). Moreover, by construction, \( S \) respects \( \prec_{RT}^{\overline{H}} = \prec_{RT}^{\overline{H}} \) and \( \prec_{CO}^{\overline{H}} = \prec_{CO}^{\overline{H}} \).

Since every two events related by the conflict relation (w-w, r-w, or w-r) in \( S \) are also related by \( \prec_{CO}^{\overline{H}} \), we obtain \( \prec_{S}^{\overline{H}} = \prec_{CO}^{\overline{H}} \). Since \( H \) is legal, \( \overline{H} \) is also legal. Combining this with Lemma 5, we get that \( S \) is also legal. This satisfies all the conditions necessary for \( H \) to be co-opaque. \( \square \)

A.2 Proofs of local correctness and non-interference

Theorem 11. For every local correctness property \( P \), \( Perm(P) \subseteq NI(P) \).

Proof. We proceed by contradiction. Assume that \( H \) is in \( Perm(P) \) but not in \( NI(P) \). More precisely, let \( T_a \) be an aborted transaction in \( H \), \( R \subseteq IncAbort(T_a, H) \) and \( H \in H_{T_a}^{CLO} \), such that \( H \in P \).

On the other hand, since \( H \in Perm(P) \), we have \( H_{T_a}^{CLO} \cap P = \emptyset \). Since \( P \) is local and \( H \in P \), we have \( \forall T_i \in txns(P), subC(H, T_i) \in P \). Thus, for all transactions \( T_i \) that committed before the last event of \( T_a \), we have \( subC(H, T_i) = subC(H_{T_a}^{CLO}, T_i) \in P \).

Now we construct \( \tilde{H} \) as \( H_{T_a}^{CLO} \) except that the aborted operation of \( T_a \) is replaced with the last operation of \( T_a \) in \( \tilde{H} \). Since \( \tilde{H} \) is in \( P \), and \( P \) is local, we have \( subC(\tilde{H}, T_a) \in P \). For all transactions \( T_i \) that committed before the last event of \( T_a \) in \( \tilde{H} \), we have \( subC(\tilde{H}, T_i) = subC(H_{T_a}^{CLO}, T_i) \in P \). Hence, since \( P \) is local, we have \( \tilde{H} \in P \). But, by construction, \( \tilde{H} \in H_{T_a}^{CLO} \) — a contradiction with the assumption that \( H_{T_a}^{CLO} \cap P = \emptyset \). \( \square \)

A.3 Proof for Garbage Collection

Lemma 6. \( H_{ig} \) is in \( CLO \) if and only if \( obs(H_{ig}, U) \) and \( trim(H_{ig}, U) \) are in \( CLO \).

Proof. (Only if) Suppose that \( H_{ig} \) is in \( CLO \). By Corollary 2, \( H_{ig} \) is legal. Since \( obs(H_{ig}, U) \) is a prefix of \( H_{ig} \), it is also legal, and its conflict graph is a sub-graph of \( CG(H_{ig}) \). By Theorem 3, \( CG(obs(H_{ig}, U)) \) is acyclic and, thus, \( obs(H_{ig}, U) \) is in \( CLO \).

Further, let \( r_k(x, v) \) be any read operation in \( trim(H_{ig}, U) \). Since \( H_{ig} \) is legal, \( r_k(x, v) \) is also legal. Note that since no read operation of obsolete transactions in \( H_{ig} \) appears in \( trim(H_{ig}, U) \), \( T_k \) is not in \( U \). Let \( c_m \) be \( r_k(x, v) \) ’s lastWrite in \( H_{ig} \). If \( c_m \) appears in \( trim(H_{ig}, U) \), then \( c_m \) is also \( r_k(x, v) \) ’s lastWrite in \( trim(H_{ig}, U) \), and, thus, \( r_k(x, v) \) is also legal. Now, suppose, by contradiction, that \( c_m \) does not appear in \( trim(H_{ig}, U) \), i.e., \( c_m \) is not the last (obsolete) transaction in \( U \) to commit a value on \( x \), i.e., there exists a transaction \( T_s \in U \) writing to \( x \) such that \( c_s \) appears after \( c_m \) in \( H_{ig} \). Since \( c_m \) is \( r_k(x, v) \) ’s lastWrite in \( H_{ig} \), \( c_s \) appears after \( r_k(x, v) \) in \( H_{ig} \). But \( T_s \) is obsolete, and, thus, no read operation can appear before \( c_s \) in \( trim(H_{ig}, U) \) — a contradiction. Thus, \( c_m \) is \( r_k(x, v) \) ’s lastWrite in \( trim(H_{ig}, U) \), and, hence, \( trim(H_{ig}, U) \) is legal.

Since \( trim(H_{ig}, U) \) is a legal sub-sequence of \( H_{ig} \), \( CG(trim(H_{ig}, U)) \) is a sub-graph of \( CG(H_{ig}) \) and, by Theorem 3, \( CG(trim(H_{ig}, U)) \) is acyclic and \( trim(H_{ig}, U) \) is in \( CLO \).

(If) Suppose now that \( obs(H_{ig}, U) \) and \( trim(H_{ig}, U) \) are in \( CLO \). By Corollary 2, both histories are legal, and, by Theorem 3, produce acyclic conflict graphs. Immediately, every read operation in \( H_{ig} \) that also appears in \( obs(H_{ig}, U) \) is legal. By the arguments above, the lastWrite for every read operation in \( trim(H_{ig}, U) \) is also its lastWrite in \( H_{ig} \). Thus, \( H_{ig} \) is legal.
Recall that $H_{ig}$ can be represented as $\text{trim}(H_{ig}, U)$ with read events of transactions in $U$ inserted in accordance to its prefix $\text{obs}(H_{ig}, U)$. Thus, $CG(H_{ig})$ can be represented as $CG(\text{trim}(H_{ig}, U))$ with several additional edges directed to and from transactions in $U$.

Suppose, by contradiction that $CG(H_{ig})$ contains a cycle $C$. Since $CG(\text{trim}(H_{ig}, U))$ is acyclic, $C$ must contain an edge directed to or from a transaction in $U$ that does not appear in $CG(\text{trim}(H_{ig}, U))$.

Thus, we can represent the cycle $C$ as $T_{i_1}, T_{i_2}, \ldots, T_{i_k}$, where $T_{i_1} = T_{i_k} \in U$ and for all $j = 1, \ldots, k - 1$, $(T_{i_j}, T_{i_{j+1}}) \in CG(H_{ig})$. Since $CG(\text{obs}(H_{ig}, U))$ is acyclic $C$ must contain an edge that does not appear in $CG(\text{obs}(H_{ig}, U))$. Let $T_{i_j}$ be the latest transaction in $T_{i_2}, \ldots, T_{i_k}$ such that $(T_{i_{j-1}}, T_{i_j}) \notin CG(\text{obs}(H_{ig}, U))$.

Note that $j \neq k$. This is because $T_{i_{k-1}}$ must precede or be concurrent to $T_{i_k} \in H_{ig}$. Since $T_{i_k} \in U$, by the construction of $\text{obs}(H_{ig}, U)$, $T_{i_{k-1}}$ must have committed in $\text{obs}(H_{ig}, U)$. But then $(T_{i_{j-1}}, T_{i_j}) \in CG(\text{obs}(H_{ig}, U))$—a contradiction.

Now, since $(T_{i_{j-1}}, T_{i_j}) \notin CG(\text{obs}(H_{ig}, U))$, $T_{i_j}$ cannot be complete in $\text{obs}(H_{ig}, U)$. Again, by the construction of $\text{obs}(H_{ig}, U)$, no transaction that is not complete in $\text{obs}(H_{ig}, U)$ can begin before $T_{i_k} \in U$ completes. Hence, $T_{i_k}$ precedes $T_{i_j}$ in the real-time order and, since $T_{i_j}$ also appears in $\text{obs}(H_{ig}, U)$, $(T_{i_k}, T_{i_j}) \in CG(\text{obs}(H_{ig}, U))$. Thus, $CG(\text{obs}(H_{ig}, U))$ contains a cycle $T_{i_k}, T_{i_j}, T_{i_{j+1}} \ldots, T_{i_k}$—a contradiction.

Thus, $CG(H_{ig})$ is acyclic and, by Theorem 3, $H_{ig}$ is in $\mathcal{CLO}$. $\square$