On Non-Interference and Locality in Transactional Memory

Petr Kuznetsov\textsuperscript{1} Sathya Peri\textsuperscript{2} \textsuperscript{*}

\textsuperscript{1}Téléc\textsuperscript{oon} ParisTech, petr.kuznetsov@telecom-paristech.fr
\textsuperscript{2}CSE Dept, IIT Patna, sathya@iitp.ac.in

Abstract

The promise of transactional memory is 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 usually expected that 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 the writes of aborted or incomplete transactions do not appear in the global serial execution, and, thus, no committed transaction can be affected by them. We consider another, less obvious, feature of “not taking effect” called non-interference: aborted or incomplete transactions should not force any other transaction to abort. More precisely, 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.

We show that for a correctness criterion to be implementable in a non-interfering way it is sufficient to be local, i.e., to only require that every transaction can be serialized along with (a subset of) the transactions committed before its last event. For example, opacity requires that all aborted transactions to fit in a single global serialization (along with all the committed transactions) is not local and cannot achieve non-interference. We propose a simple though efficient implementation that satisfies non-interference and local opacity, a novel correctness criterion that is interesting in its own right. In addition to strict serializability, local opacity captures the safety semantics of opacity: aborted transactions do not witness inconsistent states.

Regular paper, not eligible for best student paper award

\textsuperscript{*}Contact author: Sathya Peri, CSE Dept, IIT Patna, Patliputra Colony, Patna - 800 013, Bihar, India, Ph: +91-612-2552089
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 safely committed. We call a TM that never exports such executions non-interfering. The non-interference phenomenon was first highlighted in [17].

Thus, we may want to “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. Of course, in real environments, there can be other, software or hardware-specific, reasons to “spuriously” abort, but we are concerned with aborts caused by inter-transaction interactions affecting the execution’s correctness.

Non-interference and permissiveness. In this paper, we formally define the notion of non-interference. We observe that, when defined with respect to a given correctness criterion $C$, non-interference produces a subset of permissive [5] with respect to $C$ histories. This is not difficult to see if we recall that a permissive (with respect to $C$) implementation only aborts a transaction if committing it would violate $C$.

Moreover, 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. As a result, there exist permissive opaque histories that do not provide non-interference: some aborted transactions force other transactions to abort.

![Figure 1: A permissive opaque but not non-interfering history: $T_2$ forces $T_1$ to abort](image)

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 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 non-interference, when applied to opacity, is, in a strict sense, non-implementable. Indeed, assuming 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), any opaque permissive implementation may be brought to the scenario above, where the only option for $T_1$ in its last event is to abort. Note that this natural condition filters out pessimistic STMs [1] in which transactions never 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 to be locally consistent with the transactions that committed before it terminates? This way a transaction does not have to account for transactions that are aborted or incomplete at the moment it completes.

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 $T_3, T_2$, while $T_1$ (assuming it commits) and $T_3$ would both be consistent with the serialization $T_1 \cdot T_3$.

In this paper, we introduce the notion of local correctness. Informally, a history satisfies a local correctness property $P$ if and only if all its “local sub-histories” satisfy $P$. Here a local sub-history corresponding to $T_i$ consists of the events of $T_i$ and all transactions that committed before the last event of $T_i$ (transactions that are incomplete or aborted at that moment are ignored). We show that every permissive, with respect to a local correctness criterion $P$, implementation is also non-interfering with respect to $P$. Moreover, we conjecture that for a large class of strictly serializable properties, locality is also necessary for non-interfering implementability, i.e., no non-local property in the class can be implemented in a non-interfering way.

Virtual world consistency [11], which requires a history to be strictly serializable and every transaction in it 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 chance to commit. To avoid this, we introduce a 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 to VWC, in a locally opaque history a transaction may make progress only if it still has a chance to be commit.

Implementing conflict local opacity. Finally, we describe a 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, 15].

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 [15]. A simple garbage-collection scheme prevents the memory...
used by the algorithm from growing without bound. Informally, the size of the memory used by
the algorithm is proportional to the current contention, i.e., to the number of concurrently live
transactions [16]. Due to space constraints, we only describe the main idea of the implementation
here. The complete details on the algorithm and its proof can be found in the technical report [13].

Roadmap. The paper is organized as follows. We describe our system model in Section 2. In
Section 3 we formally define the notion of 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 we present the main idea behind our non-interfering
CLO implementation. Section 6 concludes the paper with remarks on the related work and open
questions.

2 System Model

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 is equipped with 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 \( T_i \) as \( evts(T_i) \).

Histories. A history is a sequence of events, i.e., a sequence of invocations and responses 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 \( <_H \) 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: \( r_k(x, v) \), \( r_k(x, A) \), \( w_k(x, v) \), \( w_k(x, v, A) \), \( tryC_k(C) \) (or \( c_k \) for short), \( tryC_k(A) \), \( tryA_k(A) \) (or \( a_k \) for short). We identify a history \( H \) as tuple \( \langle evts(H), <_H \rangle \).

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\(^a\), and (2) each \( H|p_i \) consists of a sequence of transactions, where no new transaction begins before the last transaction completes 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 \( trans(H) \). The set

\(^a\)This restriction brings no loss of generality [14].
of committed (resp., aborted) transactions in $H$ is denoted by $\text{committed}(H)$ (resp., $\text{aborted}(H)$). The set of incomplete transactions in $H$ is denoted by $\text{incomplete}(H)$ ($\text{incomplete}(H) = \text{txns}(H) - \text{committed}(H) - \text{aborted}(H)$).

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 $T_k \prec^R HT_m$, 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 $T_k \prec^R HT_m$ nor $T_m \prec^R HT_k$, 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 $\langle \text{evts}(SH), <_{SH} \rangle$ and is defined as: (1) $<_{SH} \subseteq <_H$; (2) $\text{evts}(SH) \subseteq \text{evts}(H)$; (3) If an event of a transaction $T_k \text{txns}(H)$ is in $SH$ then all the events of $T_k$ in $H$ should also be in $SH$. For a history $H$, let $R$ be a subset of $\text{txns}(H)$, the transactions in $H$. Then $H.\text{subhist}(R)$ 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 $T_i$ in $H$ that commits before $r_K$ and $w_j(x, v)$ is in $\text{evts}(T_j)$. Formally, $\langle r_k(x, v) \rangle$ is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x, v)) \land (w_j(x, v) \in \text{evts}(T_j)) \land (v \neq A)$. The history $H$ is valid if all its successful read operations are valid.

We define $r_k(x, v)$’s lastWrite as 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, $\langle r_k(x, v) \rangle$ is legal $\Rightarrow (v \neq A) \land (H.\text{lastWrite}(r_k(x, v)) = c_i) \land (w_i(x, v) \in \text{evts}(T_i)))$. 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 $\prec^R_H$, i.e $\prec^R_H \subset \prec^R_S$. 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 $H.\text{subhist}(\text{committed}(H))$ is opaque. Thus, unlike opacity, strict serializability does not include aborted transactions in the global serialization order.

**Implementations and Linearizations.** A (STM) implementation is typically a library of functions for implementing: $\text{read}_k, \text{write}_k, \text{tryC}_k$ and $\text{tryA}_k$ for a transaction $T_k$. We say that an implementation $M_p$ is correct w.r.t a property $P$ if all the histories generated by $M_p$ are in $P$. The histories generated by an STM implementations are normally not sequential, i.e., they may have overlapping transactional operations. Since our correctness definitions are proposed for sequential histories, to reason about correctness of an implementation, we order the events in a non-concurrent history in a sequential manner. The ordering must respect the real-time ordering of the operations in the original history. In other words, if the response operation $o_i$ occurs before the invocation operation $o_j$ in the original history then $o_i$ occurs before $o_j$ in the sequential history as well. Overlapping events, i.e. events whose invocation and response events do not occur either before or after each other, can be ordered in any way.

We call such an ordering as linearization [9]. Now for a (non-sequential) history $H$ generated
by an implementation $M$, multiple such linearizations are possible. As we will define later, an implementation $M$ is considered correct (for a given correctness property $P$) if every its history has a correct linearization (we say that this linearization is exported by $M$).

We assume that the implementation has enough information to generate an unique linearization for $H$ to reason about its correctness. For instance, implementations that use locks for executing conflicting transactional operations, the order of access to locks by these (overlapping) operations can decide the order in obtaining the sequential history. This is true with STM systems such as [2, 4, 10] which use locks.

### 3 Non-Interference

In this section, we recall the notion of permisiveness [5] and then we formally define non-interference. First, we define a few auxiliary notions.

For a transaction $T_i$ in $H$, $H^{T_i}_i$ denotes the shortest prefix of $H$ containing all events of $T_i$ in $H$. Now for $T_i \in \text{aborted}(H)$, let $H^{T_i,C}$ denote the set of histories constructed from $H^{T_i}_i$, 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) $w_i(x, v)$, if the last operation is $w_i(x, v, A)$, (3) try$C_i(C)$, if the last operation is try$C_i(C)$.

If $R$ is a subset of transactions of $\text{txns}(H)$, then $H_{-R}$ denotes the sub-history obtained after removing all the events of $R$ from $H$. Respectively, $H^{T_i,C}_{-R}$ denotes the set of histories in $H^{T_i,C}$ with all the events of transaction in $R$ removed.

Finally, $\text{IncAbort}(H, T)$ 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 1** Given a correctness criterion $P$, we say that a history $H$ is permissive with respect to $P$, and we write $H \in \text{Perm}(P)$ if:

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

We can see from this definition that a history $H$ is permissive w.r.t. $P$, if no aborted transaction in $H$ can be turned into committed, while preserving $P$. An implementation $M$ is considered permissive with respect to $P$ [5] if $\text{Perm}(P)$ is the set of sequential histories exported by $M$.

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 $\text{txns}(H)$, $\text{IncAbort}(T, H)$ 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 non-interfering with respect to $P$, and we write $H \in NI(P)$ if:

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

The example in Figure 1 (Section 1) shows that \( NI(opacity) \neq Perm(opacity) \) 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 to check correctness, it is enough to ensure that, for every transaction, the corresponding local sub-history is correct. An important feature of local properties is that their permissive implementations ensure non-interference.

Formally, for \( T_i \) in \( txns(H) \), let \( subC(H, T_i) \) denote

\[
H^{T_i}_{\text{subhist}}(\text{committed}(H^{T_i}) \cup \{T_i\}),
\]

i.e., the sub-history of \( H^{T_i} \) consisting of the events of all committed transactions in \( H^{T_i} \) and \( T_i \) itself.

**Definition 3** A correctness criterion \( P \) is local if for all histories \( H \):

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

As we show in this section, one example of a local property is virtual world consistency \([11]\). Then we will introduce another local property that we call conflict local opacity (CLO), in the next section, describe a simple permissive CLO implementation.

### 4.1 Local correctness and non-interference

**Theorem 2** 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 \( \tilde{H} \in \mathcal{H}^{T_{a,C}}_{-R} \), such that \( \tilde{H} \notin P \).

On the other hand, since \( H \in Perm(P) \), we have \( \mathcal{H}^{T_{a,C}} \cap P = \emptyset \). 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 \( subC(H, T_i) = subC(H^{T_a}, T_i) \in P \).

Now we construct \( \tilde{H} \) as \( H^{T_a} \), 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 \( subC(\tilde{H}, T_a) = subC(\tilde{H}, T_a) \in P \). For all transactions \( T_i \) that committed before the last event of \( T_a \) in \( \tilde{H} \), we have \( subC(\tilde{H}, T_i) = subC(H^{T_a}, T_i) \in P \). 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 \( \mathcal{H}^{T_{a,C}} \cap P = \emptyset \). \( \square \)

As we observed earlier, for any correctness criterion \( P \), \( NI(P) \subseteq Perm(P) \). Hence, Theorem 2 implies that for any local correctness criterion \( P \), \( NI(P) = Perm(P) \).
4.2 Virtual world consistency

The correctness criterion of virtual world consistency (VWC) [11] 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 \( x \in Wset(T_i) \cap Rset(T_j) \) (recall that we assumed for simplicity that all written values are unique), or (3) there exists \( T_k \), such that \( T_i \prec^C_H T_k \) and \( T_k \prec^C_H T_j \).

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 VWC if (1) \( H.subhist(\text{committed}) \) is opaque and (2) for every \( T_i \in \text{txns}(H) \), \( H.subhist(CP(T_i)) \) 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 \( H \in VWC \) if and only if for all \( \text{subC}(H, T_i) \in VWC \). By Lemma 2, any permissive implementation of VWC is also non-interfering.

4.3 Local opacity

As shown in [11], 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 \( H.subhist(\text{committed}(H)) \), 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 \( T_3 \).

![Figure 2: A locally opaque, but not opaque history (the initial value for each object is 0)](image-url)
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 [18, 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 $T_k \prec_H T_m$, 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 $\overline{H}$ and (2) $S$ respects $\prec_R$ and $\prec_C$.

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, due to space constraints, we only describe the main idea for implementation CLO. The complete details of the implementation can be found in [13]. Our implementation is permissive w.r.t CLO. Hence, by Lemma 2 it is also non-interfering. The implementation is based on conflict-graph construction of co-opacity, a popular technique borrowed from databases (cf. [18, Chap. 3]). Now, we describe about graph characterization of co-opacity, which is used in our implementation.

#### 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 $T_i \prec_R T_j$ or $T_i \prec_C T_j$, i.e., whenever $T_i$ precedes $T_j$ in the real-time or conflict order.

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

**Lemma 3** 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., $\prec_C^{H_1} \subseteq \prec_C^{H_2}$. Then, $\prec_C^{H_1} = \prec_C^{H_2}$.

**Proof.** Here, we have that $\prec_C^{H_1} \subseteq \prec_C^{H_2}$. In order to prove $\prec_C^{H_1} = \prec_C^{H_2}$, we have to show that $\prec_C^{H_2} \subseteq \prec_C^{H_1}$. We prove this using contradiction. Consider two events $p, q$ belonging to transaction $T_1, T_2$ respectively in $H_2$ such that $(p, q) \in \prec_C^{H_2}$ but $(p, q) \notin \prec_C^{H_1}$. 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 $(p, q) \in \prec_C^{H_1}$ or $(q, p) \in \prec_C^{H_1}$. But from our assumption, we get that the former is not possible. Hence, we get that $(q, p) \in \prec_C^{H_1}$ or $(q, p) \notin \prec_C^{H_1}$. But we already have that $(p, q) \notin \prec_C^{H_2}$. This is a contradiction. \[\Box\]

**Lemma 4** Let $H_1$ and $H_2$ be equivalent histories such that $\prec_C^{H_1} = \prec_C^{H_2}$. 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 $H1$ is legal. By contradiction, assume that $H2$ is not legal, i.e., there is a read operation $r_j(x,v)$ (of transaction $T_j$) in $H2$ with lastWrite as $c_k$ (of transaction $T_k$) and $T_k$ writes $u \neq v$ to $x$, i.e. $w_k(x,u) \in evts(T_k)$. Let $r_j(x,v)$’s lastWrite in $H1$ be $c_i$ of $T_i$. Since $H1$ is legal, $T_i$ writes $v$ to $x$, i.e $w_i(x,v) \in evts(T_i)$.

Since $evts(H1) = evts(H2)$, we get that $c_i$ is also in $H2$, and $c_k$ is also in $H1$. As $<_{H1}^CO = <_{H2}^CO$, we get $c_i <_{H2} r_j(x,v)$ and $c_k <_{H1} r_j(x,v)$.

Since $c_i$ is the lastWrite of $r_j(x,v)$ in $H1$ we derive that $c_k <_{H1} c_i$ and, thus, $c_k <_{H2} c_i <_{H2} r_j(x,v)$. But this contradicts the assumption that $c_k$ is the lastWrite of $r_j(x,v)$ in $H2$. Hence, $H2$ is legal.

From the above lemma we get the following interesting corollary.

Corollary 5 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 6 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 $H$ and $S$ respects $<^CO_H$ and $<^RT_H$. Thus from the conflict graph construction we have that $CG(H)$ 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.

(if) If $H$ is legal and $CG(H)$ is acyclic then $H$ is co-opaque: Suppose that $CG(H) = CG(H)$ 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 $H$. Moreover, by construction, $S$ respects $<^RT_H= <^CO_H$. Since every two events related by the conflict relation (w-w, r-w, or w-r)in $S$ are also related by $<^CO_H$, we obtain $<^CO_H = <^CO_H$. Since $H$ is legal, $H$ is also legal. Combining this with Lemma 4, we get that $S$ is also legal. This satisfies all the conditions necessary for $H$ to be co-opaque.

Main Idea behind the Implementation. The algorithm [13] 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 tryC), the TM system constructs a conflict graph with the sub-history of all the committed transactions maintained and $T_i$ (which includes the event $e_i$). If the graph constructed has a cycle then $e_i$ is not permitted to execute and $T_i$ is aborted. Otherwise, the event $e_i$ is allowed to execute.

The system uses one global lock to protect the sub-history of committed transactions which is acquired by a transaction when it performs a memory operation. By validating each read and tryC operation only against all the previous committed transactions, a transaction is aborted only when it violates the correctness property. Hence, our implementation is permissive w.r.t to CLO. We formally prove this in [13].

The algorithm also performs garbage collection to limit the amount of state information maintained. It deletes the states of committed transactions that are never going to be used in future (i.e.
obsolete transactions). With this optimisation, the amount of state information stored is proportional to the current degree of concurrency. But it is important to show that the algorithm ensures correctness and permissiveness w.r.t CLO despite the ongoing garbage collection procedure. In fact it turns out that showing that both these properties are true with the ongoing garbage collection is non-trivial. In [13], we formally prove both these properties.

6 Concluding remarks

In this paper, we formally defined the notion of non-interference in transactional memory, originally highlighted in [17]. The notion grasps the intuition that aborted or incomplete transactions should not “cause” other transactions to abort. We observe that no opaque TM implementation can provide non-interference. We consider implementations in which every transactional operation returns if does not overlap with another one, which, e.g., filters out STMs based on 2-phase locking or pessimistic STMs [1] in which transactions never abort.

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) [11] and the (novel) local opacity (LO). Interestingly, unlike VWC, LO does not allow a transaction that is doomed to abort to waste system resources. We conjecture that for a large class of strictly serializable properties, locality is not only sufficient for achieving non-interference but also necessary, i.e., no non-local property in the class can be implemented in a non-interfering way. Intuitively, a non-local property has a correct history where a transaction aborts because of another aborted or incomplete transaction and thus, non-interference is violated. But nailing down the exact argument why non-interference cannot then be implemented requires a more specific notion of a correctness criterion and is left for future work.

We then considered CLO, a restriction of LO that, in addition, requires every local serialization to respect the conflict order [8,15] of the original sub-history. We gave the main idea for a permissive, and thus non-interfering, CLO implementation. The complete implementation along with the details of garbage collection can be found at [13]. This appears to be the only non-trivial permissive implementation known so far (the VWC implementation in [4] 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 extend our definitions to (more realistic) 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. This seems to make a lot of sense in permissive implementations, where efficient verification for strict serializability or opacity appear elusive [15].

Also, our implementation is quite simplistic as it uses one global lock to protect the history of committed transactions and, thus, it is not disjoint-access-parallel (DAP) [3,12]. An interesting challenge is to check if it is possible to construct a permissive DAP CLO implementation with invisible reads.
References


