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 [15, 16], 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 [17], NOrec [3]) are non-interfering in this sense. In contrast, some *encounter-time* implementations, such as TinySTM [5], 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* [6] 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 [7, 8], 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

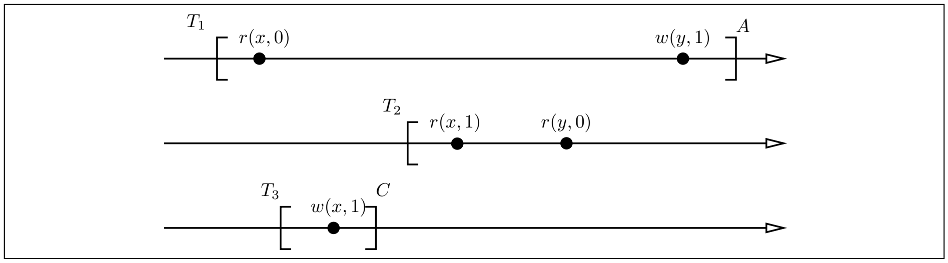

Fig. 1: An opaque-permissive opaque but not opaque-non-interfering history: T_2 forces T_1 to abort

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 $T_1 \cdot T_3$.

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 [10], 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* [9, 14].

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 [14].

The full paper is available as a technical report [12]. 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.

2 System Model and 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_i 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 $<_H$ 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[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 $T_k \in$ *incomplete*(H).

Transaction orders. For two transactions $T_k, T_m \in$ *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 $\langle evts(SH)$, $<_{SH}\rangle$ and is defined as: (1) $<_{SH} \subseteq <_H$; (2) $evts(SH) \subseteq evts(H)$; (3) If an event of a transaction $T_k \in$ *txns*(H) is in SH then all the events of T_k in H should also be in SH. (Recall that $<_H$ denotes the total order of events in H.) For a history H, let R be a subset of *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 $evts(T_j)$. Formally, $\langle r_k(x, v)$ is valid $\Rightarrow \exists T_j : (c_j <_H r_k(x, v)) \wedge (w_j(x, v) \in evts(T_j)) \wedge (v \neq A)\rangle$. The history H is valid if all its successful read operations are valid.

[c] This restriction brings no loss of generality [13].

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 $x \in Wset(T_i)$ (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, $\langle r_k(x, v)$ is legal $\Rightarrow (v \neq A) \wedge (H.lastWrite(r_k(x, v)) = c_i) \wedge (w_i(x, v) \in evts(T_i))\rangle$. 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* [7,8] 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., $\prec_H^{RT} \subset \prec_S^{RT}$. 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 $H.subhist(committed(H))$ 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 permisiveness [6] 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 $tryC_i(A)$ (and not $r_i(x, A)$ for some x), then the final $tryC_i(A)$ is not included in $applicable(T_i)$.

We denote, H^{T_i} as the shortest prefix of H containing all the events of T_i in H. Now for $T_i \in aborted(H)$, let $\mathcal{H}^{T_i,C}$ denote the set of histories constructed from H^{T_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) $w_i(x, v, A)$, if the last operation is $w_i(x, v, A)$, (3) $tryC_i(C)$, if the last operation is $tryC_i(A)$.

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, $\mathcal{H}_{-R}^{T_i,C}$ 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 $H \in Perm(P)$ if:*

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

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 $txns(H)$, $IncAbort(T, H)$ 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, $IncAbort(T, H)$ is a subset of $aborted(H) \cup incomplete(H)$.

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

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 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 $subC(H, T_i)$ denote

$$H^{T_i}.subhist(committed(H^{T_i}) \cup \{applicable(T_i)\}),$$

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 $tryC(A)$ (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 $T_i \in txns(H)$, $subC(H, T_i) \in P$.

As we show in this section, one example of a local property is virtual world consistency [10]. Then we will introduce another local property that we call conflict local opacity (CLO), in the next section and describe a simple permissive CLO implementation.

Theorem 2. *For every local correctness property P, $Perm(P) \subseteq NI(P)$.*

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 $NI(P) = Perm(P)$.

4.1 Virtual world consistency

The correctness criterion of *virtual world consistency* (VWC) [10] 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 $T_i \prec_H^{CP} T_j$ if one of the following conditions hold (1) T_i and T_j are executed by the same process and $T_i \prec_H^{RT} T_j$, (2) T_i commits and T_j reads the value written by T_i to some object $x \in \mathit{Wset}(T_i) \cap \mathit{Rset}(T_j)$(recall that we assumed for simplicity that all written values are unique), or (3) there exists T_k, such that $T_i \prec_H^{CP} T_k$ and $T_k \prec_H^{CP} T_j$. The set of transactions T_i such that $T_i \prec_H^{CP} T_j$ and T_j itself is called the *causal past* of T_j, denoted $CP(T_j)$.

Now H is in VWC if (1) $H.\mathit{subhist}(\mathit{committed})$ is opaque and (2) for every $T_i \in \mathit{txns}(H)$, $H.\mathit{subhist}(CP(T_i))$ 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 $\mathit{subC}(H, T_i) \in VWC$. By Theorem 2, any VWC-permissive implementation is also VWC-non-interfering.

4.2 Conflict local opacity

As shown in [10], 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: $\mathit{subC}(H, T_i)$ 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 $\mathit{subC}(H, T_i)$ is going to be $H.\mathit{subhist}(\mathit{committed}(H))$, 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: $\mathit{subC}(H, T_1)$ is equivalent to $T_3 T_1$, $\mathit{subC}(H, T_2)$ is equivalent to $T_3 T_2$, $\mathit{subC}(H, T_3)$ is equivalent to T_3.

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 $\mathit{txns}(H)$, we say that T_k *precedes* T_m *in conflict order*, denoted $T_k \prec_H^{CO} T_m$, if (w-w order) $\mathit{tryC}_k(C) <_H \mathit{tryC}_m(C)$ and $Wset(T_k) \cap Wset(T_m) \neq \emptyset$, (w-r order)

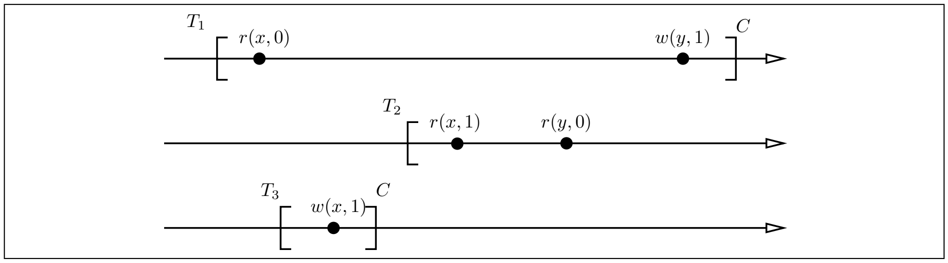

Fig. 2: A locally opaque, but not opaque history (the initial value for each object is 0)

$tryC_k(C) <_H r_m(x,v)$, $x \in Wset(T_k)$ 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 $\prec_H^{RT}$ and $\prec_H^{CO}$.*

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. [18, 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 $T_i \prec_H^{RT} T_j$ or $T_i \prec_H^{CO} T_j$, 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. [18, Chap. 4]). Hence, we call it *SGT* algorithm.

Algorithm 1 Read of a t-object x by a transaction T_i

```
procedure $read_i(x)$
    // read gComHist
    $tHist_i = gComHist$; // create a local copy of gComHist
    // create $v$, to store a the value of $x$
    $v$ = the latest value written to $x$ in $tHist_i$;
    // create $lseq_i$, the local copy of $gseqn$
    $lseq_i$ = the value of largest seq. no. of a transaction in $lComHist_i$;
    create the $readVar$ $rop_i(x, v, lseq_i)$;
    // update $lComHist_i$
    $lComHist_i$ = merge $lComHist_i$ and $tHist_i$; append $rop_i(x, v, lseq_i)$ to $lComHist_i$;

    // check for consistency of the read operation
    if ($CG(lComHist_i)$ is cyclic) then
        replace $rop_i(x, v, lseq_i)$ with ($rop_i(x, A, lseq_i)$ in $lComHist_i$);
        return abort;
    end if
    // current read is consistent; hence store it in the read set and return $v$
    return $v$;
end procedure
```

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

```
procedure $write_i(x, v)$
    if $write_i(x, v)$ is the first operation in $T_i$ then
        // read gComHist
        $lComHist_i = gComHist$;
        $lseq_i$ = the value of largest seq. no. of a transaction in $lComHist_i$;
    end if
    create the $writeVar$ $wop_i(x, v, lseq_i)$;
    append $wop_i(x, v, lseq_i)$ to $lComHist_i$;
    return $ok$;
end procedure
```

Algorithm 3 TryCommit operation by a transaction T_i

```
procedure $tryC_i$
    lock $gLock$;
    // create the next version of $gseqn$ for the current $T_i$
    $lseq_i = gSeqNum + 1$;
    $tHist_i = gComHist$; // create a local copy of $gComHist$
    $lComHist_i$ = merge $lComHist_i$ and $tHist_i$; // update $lComHist_i$
    // create the commit operation with $lseq_i$
    create the $comVar$ $cop_i(lseq_i)$;
    append $cop_i(lseq_i)$ to $lComHist_i$;
    if ($CG(lComHist_i)$ is cyclic) then
        Replace $cop_i(lseq_i)$ with $a_i$ in $lComHist_i$;
        Release the lock on $gLock$;
        return abort;
    end if
    $gComHist = lComHist_i$;
    $gSeqNum = lseq_i$;
    Release the lock on $gLock$;
    return commit;
end procedure
```

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:

- *gSeqNum*, initialized to 0 in the start of the system: global variable that counts the number of transactions committed so far.
- $lseq_i$: 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 $lseq_i$.
- *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 $rop_i(x, v, s)$ to denote the read operation in the local or global history.
- *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 $wop_i(x, v, s)$ to denote the *writeVar* operation.
- *comVar*: captures a commit operation of a transaction T_i. It stores the $lseq_i$ of the transaction. We use the notation $cop_i(s)$ to denote the *comVar* operation where s is the $lseq_i$ of the transaction.
- *gComHist*: captures the history of events of committed transactions. It is a list of *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).

- *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 $read_i(x)$, $write_i(x, v)$ and $tryC_i()$ 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 is initialized with 0 (which simulates the effect of having an initial transaction T_0).

$read_i(x)$: 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 $read_i(x)$ in $lComHist_i$, then v is assumed to be the initial value 0. Then, a *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 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 rop_i of $lComHist_i$ and then `abort` is returned. Otherwise, the value v is returned.

$write_i(x, v)$: adds a *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 $gComHist_i$.)

$tryC_i(x)$: The main idea for this procedure is similar to $read_i$, 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 $cop_i(lseq_i)$ 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 $cop_i(seq_i)$ is replaced with a_i in $lComHist_i$, the lock is released and `abort` is returned. Otherwise, $lseq_i, lComHist_i$, are copied back into *gSeqNum*, *gComHist*, the lock is released and *ok* is returned.

Now, we have the following theorems

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

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

Thus from Theorem 2 we get that SGT algorithm is CLO-non-interfering.

Theorem 6. *Assuming that no transaction fails while executing the tryC operation and gLock is starvation-free, every operation of SGT eventually returns.*

5.3 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 $lComHist_i$ 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_ℓ be any obsolete transaction in H_{ig}. Note that all transactions that committed before T_ℓ in H_{ig} are also obsolete in H_{ig}, and let U denote the set of all these obsolete transactions, including T_ℓ. Respectively, let $obs(H_{ig}, U)$ be a prefix of H_{ig} in which all transactions in $liveSet(T_\ell)$ 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 [12].

6 Concluding remarks

In this paper, we explored the notion of non-interference in transactional memory, originally highlighted in [15, 16]. 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) [10] and the (novel) local opacity (LO). Unlike VWC, LO does not allow a transaction that is doomed to abort to waste system resources. TMS1 [4] 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 [9, 14] 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 with respect to an implementation and 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. This seems to make a lot of sense in permissive implementations, where efficient verification for strict serializability or opacity appear elusive [14].

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, 11]. 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. S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.
5. P. Felber, C. Fetzer, P. Marlier, and T. Riegel. Time-based software transactional memory. *IEEE Trans. Parallel Distrib. Syst.*, 21(12):1793–1807, 2010.
6. 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.
7. 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.
8. R. Guerraoui and M. Kapalka. *Principles of Transactional Memory,Synthesis Lectures on Distributed Computing Theory.* Morgan and Claypool, 2010.
9. V. Hadzilacos. A theory of reliability in database systems. *J. ACM*, 35(1):121–145, Jan. 1988.
10. 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.
11. 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.
12. P. Kuznetsov and S. Peri. Non-interference and local correctness in transactional memory. *CoRR*, abs/1211.6315, 2013.
13. P. Kuznetsov and S. Ravi. On the cost of concurrency in transactional memory. In *OPODIS*, pages 112–127, 2011.
14. C. H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, 1979.
15. 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.
16. 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.
17. 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*, SPAA '08, pages 275–284, 2008.
18. G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann, 2002.