Breaching the Wall of Impossibility Results on Disjoint-Access Parallel TM

Technical Report, Virginia Tech, May 2014

Sebastiano Peluso* — Virginia Tech — peluso@vt.edu

Roberto Palmieri — Virginia Tech — robertop@vt.edu

Paolo Romano — INESC-ID / IST — romano@inesc-id.pt

Binoy Ravindran — Virginia Tech — binoy@vt.edu

Francesco Quaglia — Sapienza University of Rome — quaglia@dis.uniroma1.it

Abstract

Transactional Memory (TM) is a powerful abstraction for synchronizing activities of different threads through transactions. TM implementations guaranteeing Disjoint-Access Parallelism (DAP) are highly desirable on current multi-core architectures because they can exploit low-level parallelism. Unfortunately, a number of results have been proved concerning the impossibility of implementing TMs that guarantee different variants of the DAP property, as well as alternative consistency and liveness criteria.

This paper looks for a breach in the wall of existing impossibility results, by attempting to identify the strongest consistency and liveness guarantees that a TM can ensure while remaining scalable — by ensuring DAP — and maximizing efficiency in read-dominated workloads — by having invisible and wait-free read-only transactions.

We show that implementing such a TM is indeed possible if one adopts as consistency criterion Extended Update Serializability, combined with a weaker variant of real-time order, which we name *Witnessable Real Time Order*. Interestingly the resulting semantics share a number of similarities with classic TM safety criteria like Opacity and Virtual World Consistency, while allowing for scalable and efficient implementations.

Along the path of designing this protocol, we report two impossibility results related to ensuring real-time order in a weakly DAP TM that guarantees wait-free read-only transactions considering different progress criteria and both visible and invisible read-only transactions.

Finally, we also provide a lower bound on the space complexity of a strictly DAP TM that ensures a very weak consistency criterion, called Consistent View. We leverage this result to prove that the proposed protocol is optimal in terms of per object version spatial utilization.

Keywords: Software Transactional Memory, Disjoint-Access Parallelism, Real-Time Order.

*Contact author. Address: Virginia Tech, Blacksburg VA 24060, USA. Tel: +15402311418

1 Introduction

Over the last decade, Transactional Memory (TM) [27] has emerged as an attractive paradigm for simplifying parallel programming. The recent integration of TM in hardware by major chip vendors (e.g., Intel, IBM), together with the development of dedicated GCC extensions for TM (i.e., GCC-4.7) has significantly increased TM's traction, paving the way for its adoption in mainstream software projects.

A property that is deemed as crucial for the scalability of a TM implementation is its ability to avoid any contention on shared objects, also called *base objects*, between transactions that access disjoint data sets – a property called *disjoint-access parallelism* (or DAP) [20]. Also, since many real-world workloads are often read-dominated, another aspect that has a strong impact on performance of TM algorithms is to what extent these optimize the processing of read-only transactions. In this sense, two main properties are regarded as particularly important for read-only transactions: wait-freedom, i.e. (read-only) transactions should never be blocked or aborted, and invisible reads, i.e. the execution of a read operation issued by a read-only transaction should never trigger the update of any data or base object. We succinctly denote the former property as WFRO and the latter as IRO, and their union as WFIRO.

Unfortunately, the literature on theory of TM has developed a number of impossibility results related to implementing TM algorithms that guarantee different variants of the DAP property, as well as alternative consistency and liveness criteria [4, 25, 15, 13, 8]. For instance, Attiya et al. [4] proved that an TM cannot be weak DAP, ensure obstruction freedom and WFIRO, while guaranteeing Strict Serializability or even Snapshot Isolation. More recently, Bushkov et al. [8] proved the impossibility of implementing a strict DAP TM that guarantees obstruction freedom and a very weak consistency criterion, namely Weak Adaptive Consistency.

In this paper we attempt to find a breach in this wall of impossibility results, seeking an answer to the following question: what are the strongest *consistency* and *liveness* guarantees that a TM can ensure while remaining scalable — by ensuring DAP — and maximizing efficiency in read-dominated workloads — by having invisible and wait-free read-only transactions? Our search space considers the Cartesian product of the consistency criteria specified by Adya's hierarchy [1] and of a set of liveness properties that comprises both TM-specific criteria (weak and strong progressiveness [17]) as well as classical progress criteria originally defined for shared objects (obtruction-, lock- and wait-freedom [18]).

Along the path that will lead us to answer the above question, we first prove 2 novel impossibility results. We show that if one selects *any* consistency criterion that ensures Real Time Order (RTO), and independently of the isolation guarantees ensured among concurrent transactions, it is impossible to ensure also WFRO, obstruction-freedom for update transactions and the weakest form of DAP. We also show that even assuming weakly progressive update transactions [17], we are still faced with an impossibility result if we want the TM to preserve the efficiency of read-only transactions by having them performing invisible reads.

These results highlight the necessity of relaxing RTO in order to implement a scalable TM that maximizes the efficiency of read-only transactions by jointly guaranteeing DAP and WFIRO. This leads us to introduce a weaker variant of RTO, which we name *Witnessable Real Time Order* (WRTO), which demands that the real time order relation between two transactions is enforced only if these exhibit a data conflict. WRTO preserves some desirable properties of classic RTO, such as that if a transaction T running solo issues a read on a data item x, T is guaranteed to return the version of x produced by the last transaction to have committed before T's start and updated x. On the other hand, WRTO admits schedules in which a set of sequential transactions T_2, T_3 accessing disjoint data sets can be observed in an arbitrary order, which possibly contradicts their RTO relations, by a concurrent transaction T_4 (as exemplified by history $\mathcal{H}_{WRTO}$ in Figure 1). The WRTO property is indeed designed in order to be amenable for DAP implementations, as it demands that RTO is enforced only among conflicting transactions (that clearly access non-disjoint data sets), and which can track such ordering relations via some shared base-object (which serves as a witness) without violating DAP.

We show that, by adopting WRTO, it is in fact possible to implement a WFIRO TM that guarantees the strongest variant of DAP, strong progressiveness and a consistency criterion whose semantics is very close to those provided by popular safety properties for TM, such as Opacity [16] or Virtual World Consistency [19]. This consistency criterion, known in the literature as Extended Update Serializability (EUS) [24] or PL-3U

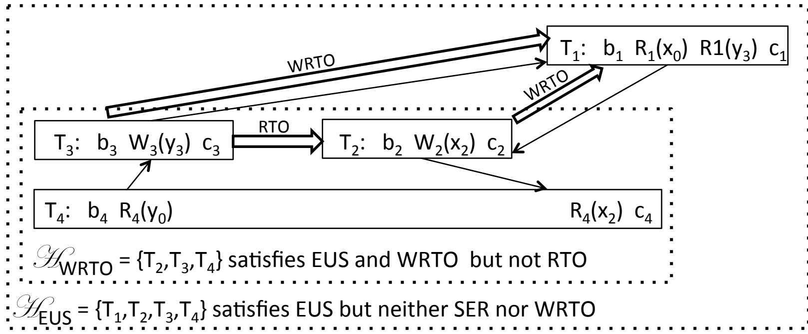


Figure 1: Example of histories accepted by EUS and WRTO (dependence edges shown with thin lines, (W)RTO edges with thick lines).

extended to executing transactions [1], guarantees the serializability of the history of committed update transactions — hence ensuring the absence of anomalies for transactions that update the state of the system. Further, analogously to Opacity or Virtual World Consistency, EUS provides guarantees also on transactions that eventually abort, ensuring that the snapshots that they observe must be producible by some equivalent serialization of the history of (committed) update transactions. On the other hand, EUS admits non-serializable histories in which read-only transactions may serialize update transactions in different orders (as it is the case for read-only transactions T_1 and T_4 that observe T_2 and T_3 in different orders in history $\mathcal{H}_{EUS}$, see Figure 1). We argue that this anomaly is a necessary price to pay to implement a DAP WFIRO TM (that ensures meaningful progress guarantees for update transactions), as, while demonstrating the impossibility of implementing a TM that guarantees DAP, WFIRO and Serializability, Attiya et al. [4] show the ineluctability precisely of this anomaly.

2 Formalism and Assumptions

System and transaction execution model. We consider an asynchronous shared memory system composed of N_p processes (or threads) $p_1, \ldots, p_{N_p}$ that communicate by executing transactions and may fail by crashing.

A transaction starts with a *begin* operation, and can be followed by a sequence of *read* and *write* operations on shared objects, and finally completed by either a *commit* (or *abort*) operation. We denote with x_i the version of the object x committed by a transaction T_i, where i is an index that univocally identifies a transaction. We note op_i an operation issued by T_i and with OP_i the set of operations issued by T_i, which is assumed to be totally ordered. We denote the write operation of T_i on object x with $W_i(x_i)$, and use the notation $R_i(x_j)$ to indicate that transaction T_i has read version x_j of x written by transaction T_j. We say that two operations op_i and op_j, with $i \neq j$, are *conflicting* if they access a common object x and at least one of them is a write.

History and DSG. A history $\mathcal{H}$ over a set of transactions $\{T_1, \ldots, T_n\}$ is a partial order $\prec_{\mathcal{H}}$ defined over the set of operations $OP_{\mathcal{H}} = \bigcup_{i=1}^{n} OP_i$ such that i) $\prec_{\mathcal{H}}$ preserves the ordering of the operations of each transaction T_i ($\prec_{\mathcal{H}} \supseteq \bigcup_{i=1}^{n} OP_i$), and ii) for any two conflicting operations op_i, $op_j \in \mathcal{H}$, either $op_i \prec_{\mathcal{H}} op_j$ or $op_j \prec_{\mathcal{H}} op_i$. In addition $\mathcal{H}$ implicitly induces a total order $\ll$ on the committed versions of each object [1].

We define the Direct Serialization Graph $DSG(\mathcal{H})$ on a history $\mathcal{H}$ (as in [1, 7]) as a direct graph with a vertex for each transaction T_i in $\mathcal{H}$ and a directed edge from T_i to T_j, where $i \neq j$, if there exist two operations op_i,$op_j \in OP_{\mathcal{H}}$ such that $op_i \prec_{\mathcal{H}} op_j$ or $op_j \prec_{\mathcal{H}} op_i$. As in [1], we distinguish three types of edges depending on the type of conflicts between T_i and T_j:

- *Direct read-dependence edge*, if there exists an object x such that both $W_i(x_i)$ and $R_j(x_i)$ are in $\mathcal{H}$. We say that T_j directly read-depends on T_i and we use the notation $T_i \xrightarrow{wr} T_j$.
- *Direct write-dependence edge*, if there exists an object x such that both $W_i(x_i)$ and $R_j(x_j)$ are in $\mathcal{H}$ and x_j immediately follows x_i in the total order defined by $\ll$. We say that T_j *directly write-depends* on T_i

and we use the notation $T_i \xrightarrow{ww} T_j$.

- *Direct anti-dependence edge*, if there exists an object x and a committed transaction T_k in $\mathcal{H}$, with $k \neq i$ and $k \neq j$, such that both $R_i(x_k)$ and $W_j(x_j)$ are in $\mathcal{H}$ and x_j immediately follows x_k in the total order defined by $\ll$. We say that T_j *directly anti-depends* on T_i and we use the notation $T_i \overset{rw}{\dashrightarrow} T_j$.

DAP and Invisible Reads. In this paper we rely on two versions of *disjoint-access parallelism* (DAP), namely *strict disjoint-access parallelism* [15] (SDAP), and *weak disjoint-access parallelism* [4] (WDAP). A TM ensures SDAP if two transactions conflict on a common base object only if they access some common transactional object. As base object we mean any information (data and meta-data) associated with a high-level transactional object, that is accessible by transactions. On the other hand, a TM implementation ensures WDAP if two concurrent transactions T_1 and T_2 conflict on a common base object only if there is a path that connect the two transactions in the undirected version of the DSG of the minimal execution interval containing the execution intervals of T_1 and T_2 [4].

A read operation R_i performed by T_i is called *invisible* if does not apply a write operation on any base object. Otherwise it is called *visible*. A read-only transaction performing only invisible reads, is called invisible.

Progress guarantees. A TM is *strongly progressive* [17] if (i) transactions executed by the TM that do not encounter any conflict must be able to commit, and (ii) at least one transaction among a set of transactions that only conflict on one common object must be able to commit. A weaker form of this progress condition, i.e., *weak progressiveness*, has also been defined in [17], which requires that a transaction can only abort if it experiences a conflict.

We consider two additional liveness properties, namely *obstruction-freedom* and *wait-freedom*. A TM is *obstruction-free* [15] if for every history $\mathcal{H}$ executed by the TM, a transaction $T_i \in \mathcal{H}$ is forcefully aborted only if T_i encounters *step contention*. We have a *step contention* for a transaction T_i if a process different from the one running T_i executes a step after the first operation of T_i and before its completion (whether commit or abort). As for *wait-freedom* we adopt the definition adapted for TM that was introduced in Attya et al. [4]: a TM is *wait-free* [18] if any transaction executed by a non-faulty process eventually commits in a finite number of steps despite the behavior of concurrent transactions[1]. We consider processes as non-faulty if they do not crash and they are not parasitic, i.e., they eventually request the commit of every transaction that they start unless they crash before [9].

Consistency criteria. Throughout our paper we will refer to the hierarchy of consistency criteria defined by Adya [1], which encompass a number of criteria defined in terms of the anomalies that they proscribe.

The minimum correctness level considered in this paper is the well known Read Committed level [6] included by the formalization of the PL-2 level in [1]. PL-2 includes both PL-2+ and EUS, and it proscribes the anomalies G1a, G1b and G1c. Proscribing *G1a* means that values created by transactions that abort cannot be observed. Anomaly *G1b* allows for observing intermediate non-committed values. Finally, avoiding anomaly *G1c* ensures the absence of an oriented cycle of all dependence edges in the $DSG(\mathcal{H}^c)$ graph built on the history $\mathcal{H}^c$, where $\mathcal{H}^c$ is derived from $\mathcal{H}$ by removing aborted and executing (i.e. ongoing) transactions. Informally, an TM implementation that guarantees PL-2, allows a transaction T_k to only read the updates of transactions that have committed by the time T_k commits.

We consider also a correctness criterion stronger than PL-2, named *Consistent View* (PL-2+) [1]. Besides G1a, G1b, G1c, PL-2+ prevents the G-single anomaly, hence avoiding that $DSG(\mathcal{H})$ contains an oriented cycle with exactly one anti-dependence edge. Roughly speaking, PL-2+ demands that transactions are always provided with a consistent view of the transactional state, as long as write transactions apply their changes consistently.

Finally, EUS [24], also called PL-3U extended to executing transactions by Adya [1], is a consistency criterion stronger that PL-2+. EUS proscribes the same anomalies of PL-2 as well as Extended G-update, namely the $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed write transactions in $\mathcal{H}$ plus transaction T_k in $\mathcal{H}$ contains an oriented cycle with one or more anti-dependence edges.

[1]We adopt the definition provided in [4] because we want to relate the results presented in this paper with the ones presented in [4]. For a formal definition of the strongest progress condition specifically defined for (S)TM, i.e., *local progress*, refer to the work in [9].

The graph considered in the Extended G-update anomaly only includes committed write transactions and at most one additional transaction T_k belonging to one among the following categories: aborted, executing or read-only transactions. EUS admits non-serializable histories, as illustrated in history $\mathcal{H}_{EUS}$ of Figure 1, which allows two read-only transactions (T_4 and T_1 in $\mathcal{H}_{EUS}$) to observe in different orders the commits of non-conflicting update transactions (T_2 and T_3 in $\mathcal{H}_{EUS}$).

On the other hand, the only discrepancies in the serialization orders observable by read-only, executing and aborted transactions under EUS are imputable to the re-ordering of update transactions that neither conflict (directly or transitively) on data, nor are causally dependent. In other words, the only discrepancies perceivable by end-users are associated with the ordering of logically independent concurrent events, which has typically no impact on the correctness of a wide range of real-world applications [1].

Real-Time Order and Witnessable Real-Time Order. Real-Time Order (RTO) is a partial order defined over a transaction history $\mathcal{H}$, noted $\prec_{\mathcal{H}}^{RTO}$, which reflects the happened-before relation between transactions in a history. A transaction T_q is ordered before T_k in RTO, $T_q \prec_{\mathcal{H}}^{RTO} T_k$, if the commit operation c_q of T_q precedes the begin operation b_k of T_k in $\mathcal{H}$.

We introduce a weaker variant of RTO, which we call *Witnessable Real Time Order* (WRTO), which tracks happened-before relations exclusively between directly conflicting transactions, or formally $T_q \prec_{\mathcal{H}}^{WRTO} T_k$ if $T_q \prec_{\mathcal{H}}^{RTO} T_k$ and T_q and T_k conflict.

A history $\mathcal{H}$ preserves RTO, respectively WRTO, if after having included in DSG($\mathcal{H}$) a direct edge $\forall T_q, T_k$ in $\mathcal{H}$, such that $T_q \prec_{\mathcal{H}}^{RTO} T_k$, respectively $T_q \prec_{\mathcal{H}}^{WRTO} T_k$, then the resulting graph does not contain cycles that include T_q and T_k. An example history that ensures WRTO but not RTO is shown in Figure 1 ($\mathcal{H}_{WRTO}$), in which T_4 that runs concurrently with two update transactions T_3 and T_2, where T_2 runs sequentially after T_3 (hence $T_2 \prec_{\mathcal{H}}^{WRTO} T_3$) and update disjoint data sets (hence $T_2 \not\prec_{\mathcal{H}}^{WRTO} T_3$), observes the committed versions of T_2 but not those of T_3.

3 The Impossibility Results on DAP and Real-Time Order

In this Section we try to understand whether it is possible to combine RTO and a reasonable consistent criterion in DAP TM that guarantees also desirable guarantees such as WFRO and IRO.

To answer this, we prove that a DAP TM cannot guarantee both RTO and WFRO if the progress requirement for write transactions is obstruction-freedom (Theorem 1). The result is independent of the provided consistency level and the visibility of read-only transactions. The intuition underlying the proof is that any TM that does not violate the RTO between any pair of transactions having a direct conflict (i.e., WRTO), then the TM must also admit a history that violates RTO.

Theorem 1 *Given a WDAP, obstruction-free TM, that guarantees WFRO, $\exists \mathcal{H}$ accepted by the TM such that $\mathcal{H}$ does not preserve RTO.*

Proof. The proof follows by contradiction and throughout the proof we assume that two different transactions are executed by two distinct processes. We assume by contradiction that $\forall \mathcal{H}$ accepted by the TM $\mathcal{H}$ preserves RTO. Hence, the TM must at least preserve WRTO.

Therefore every read operation in $\mathcal{H}$ must return the last value committed at the time the operation was executed. Formally, for each transaction T_h in $\mathcal{H}$ and object x, if $r_h(x_j)$ is in $\mathcal{H}$, then $\nexists x_k$, where $x_j \ll x_k$, at the time $r_h(x_j)$ begins its execution.

This is mandatory because of the following two reasons: (i) As the system must ensure WDAP, and as we are assuming that the set of data items to be accessed during the transactions execution is not a priori known, then during the begin a transaction T_h cannot access any base object in order to determine the set of transactions that have already committed before T_h started. If it did, in fact, there always exists a history in which T_h accesses a base object y that is being updated by a transaction T_q, which registers its commit event in y and such that T_q is not connected to T_h via a path in the conflict graph, hence violating WDAP. (ii) If the last committed value is not returned by a read operation, there always exist two histories $\mathcal{H}^{\star}$ and $\bar{\mathcal{H}}$, and a transaction T_h such that $\mathcal{H}^\star$ and $\bar{\mathcal{H}}$ are indistinguishable for T_h and T_h violates WRTO in $\bar{\mathcal{H}}$. The two histories are defined as $\mathcal{H}^\star = b_h, b_q, W_q(x_q), c_q, R_h(x_0), c_h$ and $\bar{\mathcal{H}} = b_q, W_q(x_q), c_q, b_h, R_h(x_0), c_h$ (where we assume that x_0 is the initial value of x), and we say that they are indistinguishable for T_h because the operations of T_h perform the same steps (i.e., returns the same values) in both histories.

Note that T_h must commit in both histories because it is wait-free (as it is a read-only transaction). In addition, T_q must commit because it is obstruction-free and it has the opportunity to run solo in both histories.

Now we show that always reading the last available version of an object causes the violation of RTO, thus contradicting the hypothesis. In fact, a TM adopting that policy can accept history $\mathcal{H}_{WRTO}$ depicted in Figure 1, where T_4 is a read-only transaction and y_0 is the initial version of y. History $\mathcal{H}_{WRTO}$ does not preserve RTO because *i)* $T_3 \prec^{RTO}_{\mathcal{H}_{WRTO}} T_2$ and *ii)* there exists the oriented path $T_2 \xrightarrow{wr} T_4 \xrightarrow{rw} T_3$ from T_2 to T_3 in $DSG(\mathcal{H}_{WRTO})$.

Note that, in the execution that generates $\mathcal{H}_{WRTO}$, T_3 cannot abort even if the TM accepts visible read-only transactions, otherwise the TM would not guarantee obstruction-free update transactions. This follows by the fact that: *i)* T_3 is not able to distinguish between an execution that generates $\mathcal{H}_{WRTO}$ and an execution in which T_4 commits before T_3 begins; *ii)* T_3 cannot wait for the commit of T_4, otherwise the execution of T_3 is slowed down due to possible interruption (or crash) of the execution of the process running T_4.

Therefore we have showed that a WDAP, obstruction-free TM, that guarantees WFRO and at least WRTO, can always generate a history like $\mathcal{H}_{WRTO}$ that violates RTO. Hence a WDAP, obstruction-free TM, that guarantees WFRO, does not preserve RTO, namely $\exists\mathcal{H}$ accepted by the TM such that $\mathcal{H}$ does not preserve RTO. □

In the next result we analyze the possibility to have real-time order when considering weak progressiveness as the progress guarantee for write transactions. The answer is still negative (Theorem 2) if we require WFIRO, namely wait-free and invisible read-only transactions. The idea behind the proof follows the one of Theorem 1 and considers also that write transactions cannot detect a conflict with read-only transactions due to the invisibility of the latter. For space constraints we leave the proof of Theorem 2 in the Appendix.

Theorem 2 *Given a WDAP, weakly-progressive TM, that guarantees WFIRO, $\exists\mathcal{H}$ accepted by the TM such that $\mathcal{H}$ does not preserve RTO.*

Interestingly there exists a SDAP TM implementation proposed in [3] that guarantees Opacity (and hence RTO), and which can be easily shown to ensure WFRO. However, this TM adopts visible read-only transactions (hence not contradicting Theorem 2), because their execution needs to block the commit of concurrent and conflicting write transactions.

Another SDAP TM implementation that also guarantees invisible read-only and strongly progressive transactions while preserving Opacity is TLC [5]. However TLC is not able to guarantee wait-free read-only transactions, thus again one of the hypothesis of Theorem 2 is not met by that algorithm.

4 A SDAP TM with Real-Time Order of Conflicting Transactions

In [4], authors prove that it is impossible to combine WDAP and WFIRO in a TM implementation that guarantees (Strict) Serializability [23] or Snapshot Isolation [6]. Since these properties are still highly desirable, in this section we look for the strongest consistency criterion among those included in the Adya's hierarchy that a TM can ensure while preserving meaningful progress guarantee for update transactions.

As for what concerns RTO, our result in Theorem 2 assesses the impossibility of implementing a WDAP TM that guarantees WFIRO, RTO, even assuming a very weak progress criterion such as weakly progressive write transactions.

In the light of this set of impossibility results we target as consistency criterion EUS combined with WRTO. The choice of these consistency levels allows us to design a SDAP TM algorithm, which enforces WFIRO and guarantees strong progressiveness for update transactions. We note that the impossibility result in [8] prevents

Algorithm 1 Read operation of transaction tx on process p_i

```
Val read(T tx, Var v)
    if ∃ val' : < v, val' > ∈ tx.writeSet then
        return val'
    [Val result, bool mostRecent] ← doRead(tx, v)
    if tx.writeSet ≠ ∅ ∧ mostRecent = ⊥ then
        abort(tx)
    else
        return result

[Val,bool] doRead(T tx, Var v)
    Version ver ← v.mostRecentVers
    while nonVisible(tx,ver) ∨ unsafe(tx,ver,v) do
        if tx.upperS[ver.cid] ≥ ver.S[ver.cid] then
            tx.upperS[ver.cid] ← ver.S[ver.cid] − 1
        ver ← ver.prev
    end while
    p_i.maxS[ver.cid] ← max(p_i.maxS[ver.cid], ver.S[ver.cid]
    tx.readSet ← tx.readSet ∪ < v, ver >
    return [ver.value, ver = v.mostRecentVers]


bool nonVisible(T tx, Version ver)
    if ∃k : tx.upperS[k] ≠ −1 ∧ tx.upperS[k] < ver.S[k] then
        return ⊤
    return ⊥

bool unsafe(T tx, Version ver, Var v)
    if ∃k : p_i.maxS[k] < ver.S[k] then
        if locked(v) then
            return ⊤
        for all < v, version >∈ tx.readSet do
            if overwritten(tx, v, version, ver) then
                return ⊤
    return ⊥

bool overwritten(T tx, Var vRead, Version verRead, Version target)
    Version curr ← vRead.mostRecentVers
    while curr ≠ verRead do
        if curr.S ≤ target.S then
            return ⊤
    end while
    return ⊥
```

our algorithm from being able to achieve obstruction freedom, as it guarantees EUS (which is stronger than Weak Adaptive Consistency) and SDAP.

In the following algorithm we rely on vector clocks as identifiers of the snapshots committed and as references to select the right versions during read operations. Specifically, each process p_i maintains a vector clock, $maxS$, where $maxS[k]$ records the maximum timestamp of process p_k as seen by p_i; and a scalar, tc, that stores the timestamp associated to the last commit of process p_i. In addition, each transaction T_i has also a vector clock, $upperS$, where $upperS[k]$ represents the bound that T_i cannot exceed when reading a version written by process p_k.

To univocally identify the commits in a totally decentralized way, each version is associated with two identifiers, i.e., cid and S. The former is the identifier of the process having committed that version, while the latter is the vector clock the identifies the committed snapshot containing the version.

The core idea behind the proposed algorithm is similar to the one of the SDAP extension of TL2 presented in [5], i.e., TLC. Both protocols, in fact, ensure SDAP since transactions can only synchronize on public data structures, e.g., cid and S, associated with transactional objects and only if they execute read/write operations on those objects. Further, unlike TLC, our proposal is able to guarantee that read-only transactions always commit because their read operations always return the right version belonging to the last committed consistent snapshot that they can observe without violating EUS or WRTO.

Throughout the description of the algorithm the binary relation $\leq$ is used to define an order for both scalar values and vector clock values. In case of scalar values the relation is the standard *less-than-or-equal* relation defined for natural numbers. On the contrary, in case of vector clock values the relation has the meaning defined as follows. For each pair of vector clock values v_1, v_2, the pair $\langle v_1, v_2 \rangle$ is in $\leq$, by also writing $v_1 \leq v_2$, if $\forall i, v_1[i] \leq v_2[i]$. If there exists also an index j such that $v_1[j] < v_2[j]$, where $<$ is the standard *less* relation defined for natural numbers, then $v_1 < v_2$ holds.

Handling read and write operations. Now we focus on the key aspects of the protocol. The read operation (see Algorithm 1) on x of transaction T_i is responsible for seeking the appropriate object version to read, according to the transaction's history. Clearly, if x has been previously written by T_i, the read operation returns the written value. Otherwise, the versions chain associated with x is traversed from the newest committed version to the oldest one. Specifically for each version ver, the vector clocks $upperS$ and $maxS$ are compared to ver's snapshot S. If $maxS \geq S$, which means that the process that is executing T_i has already observed a snapshot at least as recent as S, then ver can be observed by the read of T_i. There are two scenarios in

Algorithm 2 Commit operation of transaction tx on process p_i

```
commit(T tx)
    if tx.writeSet ≠ ∅ then
        VectorClock newS ← [0, . . . , 0]
        for < v, val >∈ tx.writeSet do
            bool locked ← tryLock(v)
            if locked = ⊥ then
                abort(tx)
            else
                newS ← max(newS, v.mostRecentVers.S)
        for < v, version >∈ tx.readSet do
            bool locked ← trySharedLock(v)
            if locked = ⊥ then
                abort(tx)
        for < v, version >∈ tx.readSet do
            if version ≠ v.mostRecentVers then
                abort(tx)
            else
                newS ← max(newS, v.mostRecentVers.S)
        p_i.tc + +
        newS[i] ← p_i.tc
        p_i.maxS[i] ← p_i.tc
        for < v, val >∈ tx.writeSet do
            Version newVersion
            newVersion.value ← val
            newVersion.S ← newS
            newVersion.cid ← i
            newVersion.prev ← v.mostRecentVers
            v.mostRecentVers ← newVersion
        for < v, − >∈ tx.readSet ∪ tx.writeSet do
            unlock(v)
```

which the current version could not be readable by T_i: when $upperS < S$ on the significant entries (i.e., those different from -1), or when $maxS$ is less than or not comparable with S. In these scenarios, in fact, reading the version could lead to a history that violates EUS. In the former scenario, T_i cannot read ver because it belongs to a snapshot already skipped by T_i in a previous read, which has serialized T_i before the transaction that committed the snapshot S (that includes version ver). In the latter scenario, T_i has to check if by reading version ver, which implies serializing T_i after the transaction T' that committed ver, it is still possible to serialize all the reads already performed by the transaction after T' (which is tracked by advancing $maxS$ to S). For this reason, a re-validation of the read-set of T_i is needed to check if there exists a version $ver^\star$ that has been committed after any version in T_i's read-set, and the snapshot that contains $ver^\star$ is serialized before the snapshot that contains ver (which can be determined by comparing the vector clocks of their snapshots S). In this aspect, the proposed algorithm shares similarities with LSA [26], which also forces a re-validation of the read-set in analogous circumstances, but which relies on a shared global clock and is therefore non-DAP.

After each read operation, $maxS$ is updated by computing the maximum between the snapshot S of the returned version and the current transaction's $maxS$. Finally the transaction keeps track of the read version through the read-set. During a read operation only write transactions can be aborted if they cannot access the newest version of read object.

The write operation is straightforward. It logs only the written object in the transaction's write-set and, in case a write is executed multiple times on the same object, only the last value is maintained in the write-set.

Handling commit operation. When a transaction tries to commit (see Algorithm 2), it tries to acquire an exclusive lock on each object stored in its write-set, thus it can safely add a new version to the chain. If at least one of the lock acquisitions fails the transaction immediately aborts. After that, transaction tries to acquire shared locks on the objects listed in its read-set. As before, a failed lock acquisition triggers the abort of the transaction. Only after a successful acquisition of all the requested locks, the transaction validates its read-set (by checking that the read versions are the last committed ones) and flushes the write-set into shared memory, as in classical multi-version TM implementations [26, 10].

Finally the snapshot S of each newly committed version (i.e., $newS$ in the pseudocode) results in a vector clock greater than all the most recent committed snapshots associated with the objects in the read-set and write-set. Further, the cid of those versions is equal to the identifier of the process executing the committing transaction.

For space constraint we have to insert the correctness proof of the algorithm in the Appendix. Concerning liveness guarantees, the presented algorithm ensures wait-free read-only transactions (recall that we are assuming parasitic-free histories, see Section 2) and strong progressive update transactions. The former follows trivially from that we never block or abort a read-only transaction. As for update transactions, they achieve strong progressiveness as the commit scheme that they adopt follows the lock-based scheme implemented in TL2 [12], which has been already proved to guarantee strong progressiveness in [17].

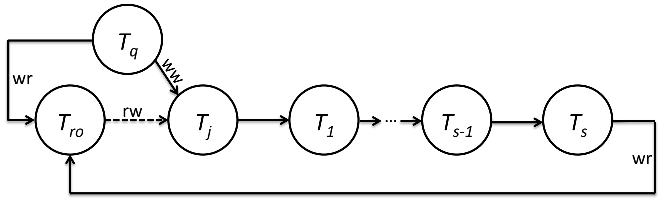


Figure 2: Read-only transaction T_i creating a cycle C with exactly one anti-dependence edge in $DSG(\mathcal{H})$.

5 Spatial complexity of ensuring WFIRO in a SDAP TM

We now investigate the spatial cost, in terms of metadata to be stored in the base object associated with each object version, which need to be incurred by a SDAP TM that guarantees WFIRO and WRTO. We prove a lower bound that holds assuming the Consistent View consistency criterion (a.k.a. as PL-2+, see Section 2) and assuming weak-progressiveness or obstruction freedom for update transactions.

In order to derive an implementation-independent proof, we use an innovative proof technique, which shows an equivalence between the problem of detecting cycles containing exactly-one anti-dependence edges using a SDAP TM, and determining causality in a distributed message passing system.

The intuition behind the proof is that whenever a read-only transaction executes a read operation, it needs to detect whether that operation creates a cycle with one anti-dependence edge in the conflict graph associated to the current history. This check has to be performed without indiscriminately access all the information associated to the conflict graph due to the existence of the SDAP requirement, but only extracting this information via the base objects associated with the objects that it accesses.

Theorem 3 *Given a SDAP TM that guarantees WFIRO, Consistent View, WRTO and either obstruction freedom or weak-progressiveness for the update transactions, then the space complexity for each version of a datum is* $\Omega(m)$*, where* $m = min(N_o, N_p)$*.*

Proof. To guarantee Consistent View, the TM has to ensure that every accepted history $\mathcal{H}$ does not contain a cycle C with exactly one anti-dependence edge in the $DSG(\mathcal{H})$. We assume that an initial version of each data item d exists in the TM, which we denote with d_0. Now consider the history $\mathcal{H}$ whose $DSG(\mathcal{H})$ is shown in Figure 2, in which the first transaction to execute in absence of concurrency is T_q, which commits version x_q. As we are assuming obstruction-freedom or weak-progressiveness, the TM cannot refuse T_q's commit.

Next in $\mathcal{H}$ a read-only transaction T_{ro} issues a read on object x. As we are assuming WFRO, the read operation of T_{ro} must eventually return some value. Assume, with no loss of generality, that the value x_q is returned. Next, and before T_{ro} takes any other step (e.g., because it was descheduled), transaction T_j starts, writes x_j and d_j^1 (where we assume object $d^1 \neq x$) and commits (we will shortly prove that this commit cannot be rejected by the TM). Following the commit of T_j, the set of update transactions $\mathcal{T}$=$\{T_1,..,T_i,..,T_{s-1},T_s\}$ is executed sequentially. Each transaction $T_i \in \mathcal{T}$ issues the following operations in $\mathcal{H}$: T_i starts, reads a object d^i, writes a different object d^{i+1}, and requests to commit. We further assume that each transaction T_i runs solo, i.e. T_{i+1} starts only after T_i commits. As we are assuming that transaction T_1 and the transactions in $\mathcal{T}$ run solo, they must commit if we assume obstruction-freedom. If we assume weak-progressiveness, on the other hand, T_j may abort due to the presence of anti-dependence from T_{ro}. However, since we are assuming invisible read-only transactions, T_j cannot detect the occurrence of this conflict, and, also in this case, it cannot abort.

Now assume that T_{ro} issues a read operation on object d^{s+1}. At this point, as T_s committed version d_s^{s+1}, T_{ro} needs to decide whether to observe this version. Note that since T_{ro} has already developed an anti-dependence towards T_1, if T_{ro} observed d_s^{s+1}, Consistent View would be violated, as a cycle with exactly one anti-dependence would be created. Also, since we are assuming that the TM ensures WRTO, it cannot deterministically return the initial version d_0^{s+1}. In fact, using such a deterministic policy, it is straightforward to show that a read-only transaction T' may trivially miss the version committed by an update transaction that commits before T' starts.

Also, in the assumed history, T_{ro} cannot be aware of having developed an anti-dependence towards T_j, as we are assuming IRO. Hence, by no means, T_{ro} could have transmitted any information to T_j on the execution of its read on x_q. Further, no other transaction could have notified T_{ro} of the existence of such anti-dependence.

Note that if T_{ro} ignored the possibility of having developed new anti-dependences when determining the visibility of d_s^{s+1}, it could miss cycle C, and violate Consistent View. It follows that T_{ro} has to first validate its current read-set, which comprises only x_q. This allows T_{ro} to detect the anti-dependence with T_j, and poses T_{ro} with the problem to determine whether there exists an oriented path from T_j to T_s (in which case d_s^{s+1} should not be observed). Note that since we are assuming a SDAP TM, T_{ro} needs to detect the existence of a path of direct dependencies from T_j to T_s, without however being able to query any of the base objects that the transactions T_1,. . .,T_{s-1} accessed (as T_{ro} accesses a disjoint data set with respect to these transactions).

In a SDAP TM in fact the only way for transactions to transmit information concerning the conflicts that they develop is via the base objects associated with the transactional objects that they access. The transmission of this information can be emulated considering a distributed message passing system (DS) comprising the same number of processes considered in the TM, namely N_p. Consider, in particular, the following simulation: for each direct read-dependence edge $T_i \xrightarrow{wr} T_{i+1}$ $\in$DSG($\mathcal{H}$) developed by a pair of write/read operations on version d_i^{i+1} of transactional object d^{i+1}, we can associate the events of send, resp. receive, of a message $m_{i,i+1}$ in DS from p_i, resp. to p_{i+1}. Since the communication of any type of information on the ordering of events in a SDAP TM can only take place via base objects, this can be simulated in the DS by assuming that $m_{i,i+1}$ can only be tagged with the information that T_i had stored in the base object associated with version d_i^{i+1}, at the time in which T_i created it. Analogously for the direct anti-dependence edge $T_{ro} \xrightarrow{rw} T_j$ $\in$DSG($\mathcal{H}$) developed by the operations $R(x_q)$ and $W(x_j)$, we can associate the events of send, resp. receive, of a message $m_{j,ro}$ in DS from p_j, resp. to p_{ro}. What triggers the sending of this message in this history is the fact that T_{ro} has to access the base object of x_j (and of all existing versions of x) in order to validate its read-set.

With this simulation we have transformed the problem of determining whether there exists a path from T_j to T_s in DSG($\mathcal{H}$) based on the information available to T_{ro}, to the problem of having the process p_{ro} (that executes transaction T_{ro}) in DS to determine whether the two messages $m_{j,ro}$ and $m_{s,ro}$ are causally ordered [21], namely $m_{j,ro} \prec_{DS} m_{s,ro}$. Thanks to the result in [14, 22], in a distributed system of N_p processes such as DS_{TM}, given two events e and e', $e \prec_{DS} e'$ iff $\Theta(e) < \Theta(e')$, where $\Theta(e)$ (respectively $\Theta(e')$) is the vector clock of size N_p associated to event e (respectively e'). Hence, the base objects, which represent the only way to exchange information on the relative ordering of operations in a SDAP TM, need to have a space capacity equals [11] to $\Omega(N_p)$.

An alternative approach to encode the entire set of dependencies developed by a transaction T during its execution is to store in the base objects associated to the versions created by T a vector containing a scalar for each transactional object in the TM (hence vector clocks have size equal to N_o). A TM implementing such a technique is for instance shown in Ardekani et al. [2], and is based on the idea of tracking in the d-th entry of a base object associated with an object version created by transaction T, a scalar that identifies the version of d that is visible to T.

To summarize, we have shown that a lower bound on the spatial cost for the base objects of a SDAP TM that guarantees Consistent View, WFIRO, WRTO and obstruction-freedom (or weak-progressiveness), is $\Omega(m)$, where $m = min(N_o, N_p)$.

□

It should be noted that, whenever the number of processes is less than the number of shared objects (which is normally the case), the algorithm presented in Section 4 meets this lower bound and is therefore optimal.

6 Relations with Existing Impossibility Results

The result of Theorem 1 enriches the result showed in [4] because the lower bound defined on the number of write operations that have to be executed by read-only transactions in a Strict Serializable and WDAP TM with obstruction-free write transactions, is only a necessary condition to ensure wait-free read-only transactions. In fact, our result is independent of the visibility of read-only transactions and states that such a TM does not preserve the real-time order. Since Strict Serializability demands that the equivalent serialization order of committed transactions also preserves real-time order among them [23, 16], by Theorem 1 no WDAP TM with obstruction-free write transactions and wait-free read-only transactions can ever guarantee Strict Serializability even by applying the number of non-trivial, i.e., write, operations in the read-only transactions according to the lower bound in Attiya et al. [4].

In Perelman et al. [25], the authors prove that a WDAP TM cannot guarantee MV-permissiveness and Strict Serializability. MV-permissiveness allows only write transactions to abort due to conflicts with other write transactions, and therefore it guarantees that read-only transactions are not forcefully aborted. If we suppose a parasitic-free system, this property defines liveness guarantees for both read-only and write transactions. In particular, on one side by ensuring that a transaction eventually requests to commit, MV-permissiveness implies wait-free read-only transactions; on the other side, since write transactions may be forcefully aborted due to a conflict with other write transactions, the property entails weakly progressive write transactions. Furthermore the authors suppose the invisibility of read-only transactions since they cannot either abort or block the execution of concurrent write transactions. Therefore the impossibility presented in Perelman et al. [25] use the same assumptions of Theorem 2 and proves that such a TM cannot guarantee Strict Serializability. However the result is weaker than the one of Theorem 2, because the latter states that the impossibility to combine WDAP, wait-free invisible read-only transactions and weakly progressive write transactions is due to the real-time order, and it is independent of the isolation level required, e.g. Serializability.

The impossibility result by Guerraoui and Kapalka [15] rules out any possibility to combine a SDAP TM and obstruction-free transactions if the target isolation level is Serializability. This is because, besides the formal proof, the paper shows an execution as a counterexample to support the proof that we can use as is to extend the result to EUS as well. The authors consider an execution admitted by obstruction-free TM implementations with 3 write transactions T_1, T_2, T_3 such that: (i) T_2 and T_3 must commit because of the obstruction-freedom condition, (ii) the commit of T_1 would violate Serializability of the resulting history and (iii) the abort of T_1 would violate the SDAP condition. Since the example only considers write transactions and proves that is impossible to combine Serializability of write transactions with obstruction-freedom in a SDAP TM, we could not consider obstruction-free as target liveness property for write transactions in the TM presented in Section 4, because EUS demands committed write transactions to be Serializable.

This same impossibility result [15] has been recently superseded by the results of the PCL theorem [8]. The theorem proves that transactions cannot be parallel, consistent and live even by assuming obstruction-freedom and Weak Adaptive Consistency — a consistency criterion even weaker than Processor Consistency and Snapshot Isolation. The proposed SDAP implementation of TM overcomes this impossibility result by assuming a different liveness property. In particular, by changing the liveness from obstruction-freedom to strong progressiveness of only write transactions we are able to ensure: (i) the maximum level of liveness for read-only transactions without enforcing their visibility; (ii) a consistency criterion that is close to Opacity and combines EUS and WRTO.

7 Conclusion

We presented a possibility, as well as two impossibility results about implementing DAP TMs that ensure efficient (i.e., wait-free and invisible) read-only transactions. On one side, we presented a protocol proving the feasibility of building a SDAP TM, combined with invisible and wait-free read-only transactions, and preserving EUS (a consistency criterion that provides guarantees very close to Opacity and Virtual World Consistency) and WRTO (a variant of classic real-time order restricted to conflicting transactions). In addition, we derived a lower bound on the space complexity of implementing a SDAP TM that guarantees EUS, WRTO, WFIRO and obstruction freedom (or weak progressiveness). We also proved that ensuring real-time order and DAP is impossible independently of the assumed isolation criterion and considering different liveness criteria.

References

[1] Atul Adya. *Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions*. PhD thesis, 1999. AAI0800775.

[2] Masoud Saeida Ardekani, Pierre Sutra, and Marc Shapiro. Non-monotonic snapshot isolation: Scalable and strong consistency for geo-replicated transactional systems. In *SRDS*, pages 163–172, 2013.

[3] Hagit Attiya and Eshcar Hillel. Single-version STMs Can Be Multi-version Permissive. In *Proceedings of the 12th International Conference on Distributed Computing and Networking*, ICDCN, 2011.

[4] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. *Theory Comput. Syst.*, 49(4):698–719, 2011.

[5] Hillel Avni and Nir Shavit. Maintaining Consistent Transactional States Without a Global Clock. In *Proceedings of the 15th International Colloquium on Structural Information and Communication Complexity*, SIROCCO, 2008.

[6] Hal Berenson, Phil Bernstein, Jim Gray, Jim Melton, Elizabeth O'Neil, and Patrick O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[7] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] Victor Bushkov, Dmytro Dziuma, Panagiota Fatourou, and Rachid Guerraoui. The PCL Theorem. Transactions cannot be Parallel, Consistent and Live. In *Proceedings of the 26th annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2014.

[9] Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In *PODC*, pages 9–18, 2012.

[10] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[11] Bernadette Charron-Bost. Concerning the size of logical clocks in distributed systems. *Inf. Process. Lett.*, 39(1):11–16, 1991.

[12] Dave Dice, Ori Shalev, and Nir Shavit. Transactional Locking II. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC, 2006.

[13] Faith Ellen, Panagiota Fatourou, Eleftherios Kosmas, Alessia Milani, and Corentin Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *PODC*, pages 115–124, 2012.

[14] C. J. Fidge. Timestamps in message-passing systems that preserve the partial ordering. *Proceedings of the 11th Australian Computer Science Conference*, 10(1):5666, 1988.

[15] Rachid Guerraoui and Michal Kapalka. On Obstruction-free Transactions. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA, 2008.

[16] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175–184, 2008.

[17] Rachid Guerraoui and Michal Kapalka. The Semantics of Progress in Lock-based Transactional Memory. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, 2009.

[18] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1), 1991.

[19] Damien Imbs and Michel Raynal. Virtual World Consistency: A Condition for STM Systems (with a Versatile Protocol with Invisible Read Operations). *Theoretical Computer Science*, 444:113–127, July 2012.

[20] Amos Israeli and Lihu Rappoport. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, PODC, 1994.

[21] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[22] Friedemann Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms*, pages 215–226, 1989.

[23] Christos H. Papadimitriou. The Serializability of Concurrent Database Updates. *Journal of the ACM*, 26(4):631–653, October 1979.

[24] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.

[25] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM.

[26] Torvald Riegel, Pascal Felber, and Christof Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *Proceedings of the 20th International Conference on Distributed Computing*, DISC, 2006.

[27] Nir Shavit and Dan Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

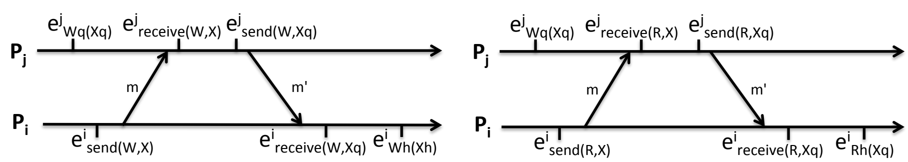
Appendix

Theorem 2 *Given a WDAP, weakly-progressive TM, that guarantees WFIRO, $\exists\mathcal{H}$ accepted by the TM such that $\mathcal{H}$ does not preserve RTO.*

Proof. The proof follows by contradiction and throughout the proof we assume that two different transactions are executed by two distinct processes. We assume by contradiction that $\forall\mathcal{H}$ accepted by the TM $\mathcal{H}$ preserves RTO. Hence, the TM must at least preserve WRTO. As showed in the proof of Theorem 1, this is possible only if the read policy implemented in the WDAP TM ensures that transactions always read the last available version of an object. But if this is the case, the TM can accept history $\mathcal{H}_{WRTO}$ depicted in Figure 1, where T_4 is a read-only transaction and y_0 is the initial version of y. History $\mathcal{H}_{WRTO}$ does not preserve RTO because *i)* $T_3 \prec^{RTO}_{\mathcal{H}_{WRTO}} T_2$ and *ii)* there exists the oriented path $T_2 \xrightarrow{wr} T_4 \overset{rw}{\dashrightarrow} T_3$ from T_2 to T_3 in $DSG(\mathcal{H}_{WRTO})$.

Note that, even if the TM guarantees weakly progressive write transactions, T_q and T_k cannot abort in $\mathcal{H}_{WRTO}$ because they cannot detect their conflict with T_h, as read-only transaction T_h is invisible according to the hypothesis.

Therefore we have showed that a WDAP, weakly progressive TM, that guarantees WFIRO and at least WRTO, can always generate a history like $\mathcal{H}_{WRTO}$ that violates RTO. As a consequence a WDAP, weakly progressive TM, that guarantees WFIRO, does not preserve RTO, namely $\exists\mathcal{H}$ accepted by the TM such that $\mathcal{H}$ does not preserve RTO. $\square$



(a) Write operation.

(b) Read operation.

Figure 3: Write and read operations a transaction on process p_i simulated by DS_{TM}.

Lemma 1 *Given a SDAP, multi-version TM that guarantees PL-2 isolation level, wait-free and invisible read-only transactions and has N_p processes that execute transactions, we can build a message-passing distributed system DS_{TM} with N_p processes, such that:*

th1. *$\forall i, q$ and object x, the DS_{TM} has a sequence of events that simulate the operations $R_i(x_q)$ and $W_i(x_i)$.*

th2. *Given p_i and p_j, two processes executing respectively transactions T_q and T_h and history $\mathcal{H}$ executed by TM such that $T_q \in \mathcal{H}$ and $T_h \in \mathcal{H}$, if $T_q \xrightarrow{ww} T_h \in DSG(\mathcal{H}) \vee T_q \xrightarrow{wr} T_h \in DSG(\mathcal{H})$ then for each pair of objects x, y (with possibly x equals to y),*
if $W_q(x_q) \in \mathcal{H}$ and $W_h(y_h) \in \mathcal{H} \Rightarrow e^i_{W_q(x_q)} \prec_{DS} e^j_{W_h(y_h)}$.

The events $e^i_{W_q(x_q)}$, $e^j_{W_h(y_h)}$ are the events that simulate the finalization respectively of the write operation $W_q(x_q)$ of T_q on the process p_i in DS_{TM} and of the write operation $W_h(y_h)$ of T_h on the process p_j in DS_{TM}. The $\prec_{DS}$ is the happened-before relation in DS_{TM} as defined in [21].

Proof. We can build the distributed system DS_{TM} as follows: for each process p_i running transactions, we have a process p_i in DS_{TM}. Processes in DS_{TM} can communicate only by exchanging messages. Since the TM is SDAP we allow two processes in DS_{TM} to exchange messages only if the associated processes in the TM execute conflicting transactions. To do so, we associate with process p_i, a version x_k of object x if the corresponding process in TM runs a transaction T_k that executes a write operation $W_k(x_k)$. Note that the DS_{TM} does not simulate read operations of read-only transactions since those are invisible. Operations executed by write transactions in TM are simulated in DS_{TM} as listed below.

Write operation $W_h(x_h)$. Process p_i in TM runs transaction T_h that writes version x_h of object x. We suppose that the last version of x before this write operation is x_q. In addition x_q has been previously written by transaction T_q in execution on process p_j. Then DS_{TM} executes the following steps as in Figure 3(a): *i)* p_i sends a message m to p_j by means of the event $e^i_{send(W,x)}$ in order to simulate that T_h writes after T_q on x; *ii)* p_j sends a message m' to p_i by means of the event $e^j_{send(W,x_q)}$ after the event $e^j_{receive(W,x)}$ on the reception of message m; *iii)* p_i generates the event $e^i_{W_h(x_h)}$ after the event $e^i_{receive(W,x_q)}$ on the reception of message m'. Since the previous write operation $W_q(x_q)$ is completed in DS_{TM} by means of the event $e^j_{W_q(x_q)}$ and before the event $e^j_{receive(W,x)}$, it follows that $e^j_{W_q(x_q)} \prec_{DS} e^i_{W_h(x_h)}$.
Read operation $R_h(x_q)$. Process p_i in TM runs transaction T_h that reads version x_q of object x. We suppose that transaction T_q writes x_q and process p_j runs T_q. Then DS_{TM} executes the following steps as in Figure 3(b): *i)* p_i sends a message m to p_j by means of the event $e^i_{send(R,x)}$; *ii)* p_j sends a message m' to p_i by means of the event $e^j_{send(R,x_q)}$ after the event $e^j_{receive(R,x)}$ on the reception of message m; *iii)* p_i generates the event $e^i_{R_h(x_q)}$ after the event $e^i_{receive(R,x_q)}$ on the reception of message m'. Since the write operation $W_q(x_q)$ is completed in DS_{TM} by means of the event $e^j_{W_q(x_q)}$ and before the event $e^j_{receive(R,x)}$, it follows that $e^j_{W_q(x_q)} \prec_{DS} e^i_{R_h(x_q)}$.

Following the above rules, we have showed (*th1*) how to build DS_{TM} on a SDAP TM with invisible read-only transactions.

Now we prove *th2* by contradiction. We assume by a way of contradiction that, given a SDAP, multi-version TM that guarantees wait-free and invisible read-only transactions, for each history $\mathcal{H}$ if $T_q \xrightarrow{ww} T_h \in DSG(\mathcal{H}) \vee T_q \xrightarrow{wr} T_h \in DSG(\mathcal{H})$ and there exist two write operations $W_q(x_q) \in \mathcal{H}$ and $W_h(y_h) \in \mathcal{H}$ such that $e^j_{W_h(y_h)} \prec_{DS} e^i_{W_q(x_q)} \vee e^i_{W_q(x_q)} \parallel e^j_{W_h(y_h)}$, then $\mathcal{H}$ may violate the PL-2 isolation level. We use $e^i_{W_q(x_q)} \parallel e^j_{W_h(y_h)}$ to state that neither $e^j_{W_h(y_h)} \prec_{DS} e^i_{W_q(x_q)}$ nor $e^i_{W_q(x_q)} \prec_{DS} e^j_{W_h(y_h)}$ hold.

In particular we distinguish two cases:

- $e^j_{W_h(y_h)} \prec_{DS} e^i_{W_q(x_q)}$. Then there may exist a transaction T_k such that $W_k(y_k) \in \mathcal{H}$, where $y_h \ll y_k$, and $W_k(x_k) \in \mathcal{H}$, where $x_k \ll x_q$, and T_k executes $W_k(y_k)$ before $W_k(x_k)$. This is admissible because there exists a process p_l in DS_{TM} that generates two events $e^l_{W_k(y_k)}$ and $e^l_{W_k(x_k)}$ such that $e^j_{W_h(y_h)} \prec_{DS} e^l_{W_k(y_k)}$, $e^l_{W_k(y_k)} \prec_{DS} e^l_{W_k(x_k)}$, $e^l_{W_k(x_k)} \prec_{DS} e^i_{W_q(x_q)}$.
- $e^i_{W_q(x_q)} \parallel e^j_{W_h(y_h)}$. Then there may exist a transaction T_k such that $W_k(y_k) \in \mathcal{H}$, where $y_h \ll y_k$, and $W_k(x_k) \in \mathcal{H}$, where $x_k \ll x_q$, and T_k executes $W_k(x_k)$ before $W_k(y_k)$. This is admissible because $e^i_{W_q(x_q)} || e^j_{W_h(y_h)}$ and therefore there exists a process p_l in DS_{TM} that generates two events $e^l_{W_k(y_k)}$ and $e^l_{W_k(x_k)}$ such that $e^j_{W_h(y_h)} \prec_{DS} e^l_{W_k(y_k)}$, $e^l_{W_k(x_k)} \prec_{DS} e^l_{W_k(y_k)}$, $e^l_{W_k(x_k)} \prec_{DS} e^i_{W_q(x_q)}$.

However in both cases transactions T_h, T_q and T_k generate a cycle of dependencies in $DSG(\mathcal{H})$,i.e. $T_q \xrightarrow{ww/wr} T_h \xrightarrow{ww} T_k \xrightarrow{ww} T_q$, and this contradicts the hypothesis of an TM that guarantees PL-2 isolation level [1]. □

Algorithm Correctness proof

In this Section we prove the correctness of the algorithm presented in Section 4 and therefore we prove that every history $\mathcal{H}$ accepted by the algorithm does not violate EUS. For the sake of clarity, throughout the proof we refer to the algorithm as $\mathcal{A}$. In addition we prove that every history $\mathcal{H}$ accepted by the algorithm does not violate WRTO.

We recall that a history $\mathcal{H}$ does not violate EUS if the following anomalies are prevented (as defined in [1]):

- G1a. $\mathcal{H}$ contains the operations $W_q(x_q)$, $R_k(x_q)$ and a_q. This means that transactions T_k has read a version written by an aborted transaction T_q.
- G1b. $\mathcal{H}$ contains the operations $W_q(x_q)$, $R_k(x_q)$ and $W_q(x_q)$ is not the last write of T_q on x. This means that transaction T_k has read an intermediate non-committed value of x.

- G1c. The $DSG(\mathcal{H}^c)$ graph built on the history $\mathcal{H}^c$ derived from $\mathcal{H}$ by removing aborted and executing (i.e. ongoing) transactions contains an oriented cycle of all dependence edges.
- Extended G-update. The $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed write transactions in $\mathcal{H}$ plus transaction T_k in $\mathcal{H}$ contains an oriented cycle with one or more anti-dependence edges.

We use the prefix Extended for the G-update anomaly because T_k is allowed to be either completed, thus aborted or committed, or ongoing. Note that EUS includes PL-2 isolation level, which prevents anomalies G1a, G1b and G1c, and Consistent View (PL-2+) isolation level because the G-update anomaly is more restrictive than the one characterizing Consistent View, i.e. the $DSG(\mathcal{H}^c)$ graph contains an oriented cycle with exactly one anti-dependence edge.

We do not formally prove that $\mathcal{A}$ avoids anomalies G1a and G1b because this is trivially guaranteed since *i)* for each object x and transaction T_k, T_k's write-set always contains only the outcome of the last write operation executed on x by T_k and *ii)* the T_k's write-set is made available to read operations at commit time and only if T_k commits.

The formal prove is organized as follows: we first prove that the history $\mathcal{H}^{upc}$ derived from $\mathcal{H}$ by removing aborted, executing (i.e. ongoing) and read-only transactions does not contain any oriented cycle, thus showing that $\mathcal{A}$ prevents anomaly G1c and the anomaly Extended G-update where T_k is a committed write transaction (Lemma 2); then we prove that the $DSG(\mathcal{H}^{upc}_{T_k})$ graph does not contain any oriented cycle, where T_k is a committed read-only transaction in $\mathcal{H}$, thus showing that $\mathcal{A}$ prevents anomaly Extended G-update where T_k is a committed read-only transaction (Lemma 3). Finally the Theorem 4 concludes the formal proof by taking into account that an aborted or ongoing transaction at time t can be considered as a committed read-only transaction constituted by its prefix at time t that contains all its read operations performed until time t, except the read operation which has triggered an abort (if any).

Lemma 2 *For each history $\mathcal{H}$ accepted by $\mathcal{A}$, the $DSG(\mathcal{H}^{upc})$ graph built on the history $\mathcal{H}^{upc}$ derived from $\mathcal{H}$ by removing aborted, executing (i.e. ongoing) and read-only transactions does not contain any oriented cycle.*

Proof. The proof follows by contradiction. In particular we prove that if such a cycle exists, this violates the total order property on natural numbers. Therefore we suppose that the $DSG(\mathcal{H}^{upc})$ contains an oriented cycle C. In addition, for each vertex T_q in C, we associate: *i)* the vector clock $T_q.commitVC$ that is the $newS$ vector clock used by T_q to commit new versions; *ii)* the vector clock $T_q.readVC$ that is the T_q's $maxS$ vector clock at the time T_q starts the commit phase.

Now we show that for each edge $T_q \rightarrow T_k \in C$, where T_q and T_k are committed update transactions in $\mathcal{H}^{upc}$, $T_q.commitVC < T_k.commitVC$. Therefore we distinguish three cases depending on the type of dependence between T_q and T_k:

- $T_q \xrightarrow{ww} T_k$. In this case there exists an object x such that $W_q(x_q) \in \mathcal{H}^{upc}$, $W_k(x_k) \in \mathcal{H}^{upc}$ and $x_q \ll x_k$. Due to the lock acquisition on the objects in the write-sets, T_q already committed when T_k starts the commit. Therefore $T_q.commitVC < T_k.commitVC$ because: *i)* $T_k.commitVC \geq T_q.commitVC$ since T_k builds $T_k.commitVC$ starting from $T_k.readVC$ and by means of a maximum operation among the vector clocks associated to the newest versions of the objects to be written; *ii)* there exists an index j such that the $T_k.commitVC[j] > T_q.commitVC[j]$, where p_j is the process executing T_k. The latter is true because T_k increments by 1 the $j\text{-}th$ entry of $T_k.commitVC$ before writing the new versions.
- $T_q \xrightarrow{wr} T_k$. In this case there exists an object x such that $W_q(x_q) \in \mathcal{H}^{upc}$, $R_k(x_q) \in \mathcal{H}^{upc}$. Therefore the status of $T_k.readVC$ right after the execution of $R_k(x_q)$ is at least equals to $T_q.commitVC$ because $T_k.readVC$ is updated by means of $max(T_k.readVC, T_q.commitVC)$.
  In addition $T_k.commitVC > T_k.readVC$ because: *i)* T_k builds $T_k.commitVC$ starting from $T_k.readVC$ and by means of a maximum operation among the vector clocks associated to the newest versions of the objects to be written; *ii)* there exists an index j such that the $T_k.commitVC[j] > T_k.readVC[j]$, where p_j is the process executing T_k, as proved for the previous case.
  As a consequence $T_q.commitVC < T_k.commitVC$.
- $T_q \overset{rw}{\dashrightarrow} T_k$. In this case there exists an object x such that $W_k(x_k) \in \mathcal{H}^{upc}$, $R_q(x_h) \in \mathcal{H}^{upc}$ and $x_h \ll x_k$. T_q has completed its commit before the finalization of the commit of T_k otherwise we would

have that: *i)* either T_q aborts due to a failed shared lock acquisition on x or validation of x, or *ii)* T_k aborts due to a failed exclusive lock acquisition on x. As in the first case, T_q and T_k are two conflicting update transactions where the commit of T_k follows the commit of T_q. Therefore $T_q.commitVC < T_k.commitVC$.

As a consequence if the cycle C exists we would have $T_q.commitVC < T_q.commitVC$, where T_q is a vertex in C, that is clearly impossible. Therefore the $DSG(\mathcal{H}^{upc})$ graph does not contain any oriented cycle. □

Lemma 3 *For each history $\mathcal{H}$ accepted by $\mathcal{A}$, the $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed write transactions in $\mathcal{H}$ plus committed read-only transaction T_k in $\mathcal{H}$ does not contain any oriented cycle.*

Proof. The proof follows by contradiction. In particular we prove that if such a cycle exists, this violates the total order property on natural numbers. Therefore we suppose that the $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed transactions in $\mathcal{H}$ plus committed read-only transaction T_k in $\mathcal{H}$ contains an oriented cycle C.

By the result of Lemma 2, C must involve the read-only transaction T_k because the $DSG(\mathcal{H}^{upc})$ is acyclic. In addition, following the proof of Lemma 2, for each vertex T_q in C, we associate: *i)* the vector clock $T_q.commitVC$ that is the $newS$ vector clock used by T_q to commit new versions (if any); *ii)* the vector clock $T_q.readVC$ that is the T_q's $maxS$ vector clock at the time T_q starts the commit phase.

As a consequence we prove that:

- for each committed update transaction T_q such that $T_q \xrightarrow{wr} T_k$ is in C, $T_q.commitVC \leq T_k.readVC$;
- for each committed update transaction T_q such that $T_k \overset{rw}{\dashrightarrow} T_q$ is in C, there exists an index j such that $T_k.readVC[j] < T_q.commitVC[j]$.

The former is verified because there exists an object x such that $W_q(x_q) \in \mathcal{H}^{upc}_{T_k}$, $R_k(x_q) \in \mathcal{H}^{upc}_{T_k}$ and the status of $T_k.readVC$ right after the execution of $R_k(x_q)$ is at least equals to $T_q.commitVC$ since $T_k.readVC$ is updated by means of $max(T_k.readVC, T_q.commitVC)$.

The latter is verified because there exists an object x such that $W_q(x_q) \in \mathcal{H}^{upc}$, and T_k skips version x_q when it executes $R_k(x_h) \in \mathcal{H}^{upc}$, where $x_h \ll x_q$.

Afterwards by following the proof of Lemma 2, for each dependence or anti-dependence $T_q \rightarrow T_h$, we have $T_q.commitVC < T_h.commitVC$ if both T_q and T_h are committed write transactions.

As a consequence there exists an index j such that $T_q.commitVC[j] < T_q.commitVC[j]$, for each committed write transaction in C, and $T_k.readVC[j] < T_k.readVC[j]$, which are both impossible.

Therefore we have proved that the $DSG(\mathcal{H}^{upc}_{T_k})$ does not contain any oriented cycle. □

Theorem 4 *For each history $\mathcal{H}$ accepted by $\mathcal{A}$, $\mathcal{H}$ does not violate EUS.*

Proof. By Lemma 2 the history $\mathcal{H}^{upc}$ derived from $\mathcal{H}$ by removing aborted, executing and read-only transactions does not contain any oriented cycle. Since a cycle of all dependence edges cannot involve a read-only transaction, then the $DSG(\mathcal{H}^{c})$ on $\mathcal{H}^{c}$ derived from $\mathcal{H}$ by removing aborted and executing transactions does not contain any oriented cycle of all dependence edges. In this way we have proved that $\mathcal{H}$ cannot generates anomaly G1c.

On the other side $\mathcal{H}$ cannot generates anomaly Extended G-update for the following reasons:

- The $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed write transactions in $\mathcal{H}$ plus transaction T_k in $\mathcal{H}$ does not contain any oriented cycle with one or more anti-dependence edges if T_k is a committed write transaction. This follows by the Lemma 2, because in this case $\mathcal{H}^{upc}_{T_k} = \mathcal{H}^{upc}$ and the $DSG(\mathcal{H}^{upc})$ graph does not contain any oriented cycle.
- The $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed transactions in $\mathcal{H}$ plus a committed read-only transaction T_k does not contain any oriented cycle by the result of Lemma 3 and therefore it does not contain any oriented cycle with one or more anti-dependence edges.
- The $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed transactions in $\mathcal{H}$ plus an aborted or executing T_k does not contain any oriented cycle by the result of Lemma 3 and by considering that an executing or an aborted transaction at time t can be treated as a committed read-only transaction constituted by its prefix

at time t that contains all its read operations performed until time t, except the read operation which has triggered an abort (if any). This is an admissible reduction since write operations are buffered during the execution of a transaction and they are externalized (i.e. the updates are applied) only upon a successfully completed commit phase.

Therefore the $DSG(\mathcal{H}^{upc}_{T_k})$ graph does not contain any oriented cycle with one or more anti-dependence edges.

As a consequence we have proved that for each history $\mathcal{H}$ accepted by $\mathcal{A}$, $\mathcal{H}$ does not violate EUS. □

Theorem 5 *$\forall\mathcal{H}$ accepted by $\mathcal{A}$, $\mathcal{H}$ preserves WRTO.*

Proof.

We suppose that $\exists\mathcal{H}$ accepted by $\mathcal{A}$ such that $\mathcal{H}$ does not preserve the real-time order among conflicting transactions in $\mathcal{H}$. Therefore there are two conflicting transactions T_q and T_k such that $T_q \prec_{\mathcal{H}} T_k$ and at least one of the following condition is verified:

1. $\exists x$ such that both $W_q(x_q)$ and $W_k(x_k)$ are in $\mathcal{H}$ and $x_k \ll x_q$.
2. $\exists x$ such that both $W_q(x_q)$ and $R_k(x_h)$ are in $\mathcal{H}$ and $x_h \ll x_q$.

Condition *1.* is impossible because:

- By nature of transactions if $T_q \prec_{\mathcal{H}} T_k$ then $c_q \prec_{\mathcal{H}} c_k$.
- $\mathcal{A}$ applies write operations at commit time and only if a transaction successfully commits.
- In $\mathcal{A}$ the total order on the versions of an object is defined by the total order of the commits on that object. Therefore $x_q \ll x_k$ in $\mathcal{H}$ that contradicts condition *1.*

Condition *2.* is impossible because:

- $\mathcal{A}$ has forced the read of x_h because T_k could not read x_q in $\mathcal{H}$. This happens if there exists a version y_r in T_k's read-set and a version y_t, where $y_r \ll y_t$, such that the snapshot vector clock of y_t is less than or equals to the snapshot vector clock of x_q, i.e. $y_t.snapshot \leq x_q.snapshot$. Without loss of generality we suppose that the read of y_r, i.e. $R_k(y_r)$, is the first read of T_k and the read of x_h, i.e. $R_k(x_h)$ is the second one.

  If that is the case the write operation on y_t is executed after the write operation on x_q because at the time T_k executed its first read operation $R_k(y_r)$ it returned the most recent version of y and it didn't find version y_t. In addition, at that time transaction T_q had already committed since $T_q \prec_{\mathcal{H}} T_k$. Therefore we have the contradiction such that the write on y_t is executed after the write on x_q and $y_t.snapshot \leq x_q.snapshot$. In fact, since $\mathcal{A}$ can be simulated by the distributed system $DS_{\mathcal{A}}$ (Lemma 1), we have that $e^q_{W_q(x_q)} \prec_{DS} e^t_{W_t(y_t)}$ and $y_t.snapshot > x_q.snapshot$.

Since we have proved that both conditions are impossible it must be that $\forall\mathcal{H}$ accepted by $\mathcal{A}$, $\mathcal{H}$ preserves WRTO.

□