Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations

Sebastiano Peluso
ECE Department
Virginia Tech
Blacksburg VA, USA
peluso@vt.edu

Roberto Palmieri
ECE Department
Virginia Tech
Blacksburg VA, USA
robertop@vt.edu

Paolo Romano
IST & INESC-ID
Universidade de Lisboa
Lisbon, Portugal
romano@inesc-id.pt

Binoy Ravindran
ECE Department
Virginia Tech
Blacksburg VA, USA
binoy@vt.edu

Francesco Quaglia
DIAG
Sapienza University of Rome
Rome, Italy
quaglia@dis.uniroma1.it

ABSTRACT

Disjoint-Access Parallelism (DAP) is considered one of the most desirable properties to maximize the scalability of Transactional Memory (TM). This paper investigates the possibility and inherent cost of implementing a DAP TM that ensures two properties that are regarded as important to maximize efficiency in read-dominated workloads, namely having invisible and wait-free read-only transactions. We first prove that relaxing Real-Time Order (RTO) is necessary to implement such a TM. This result motivates us to introduce *Witnessable Real-Time Order* (WRTO), a weaker variant of RTO that demands enforcing RTO only between directly conflicting transactions. Then we show that adopting WRTO makes it possible to design a strictly DAP TM with invisible and wait-free read-only transactions, while preserving strong progressiveness for write transactions and an isolation level known in literature as Extended Update Serializability. Finally, we shed light on the inherent inefficiency of DAP TM implementations that have invisible and wait-free read-only transactions, by establishing lower bounds on the time and space complexity of such TMs.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming—*distributed programming, parallel programming*; D.4.1 [**Operating Systems**]: Process Management—*concurrency, synchronization*; F.1.2 [**Computation by Abstract Devices**]: Modes of Computation—*parallelism and concurrency*

General Terms

Algorithms, Theory

PODC'15, July 21–23, 2015, Donostia-San Sebastián, Spain.
.

Keywords

Disjoint-access parallelism; Wait-freedom; Real-time order; Transactional memory; Invisible reads; Impossibility results

1. INTRODUCTION

Over the last decade, Transactional Memory (TM) [30] has emerged as an attractive paradigm for simplifying parallel programming. Further, the recent integration of TM support in hardware by major chip vendors (e.g., Intel, IBM), along with the development of dedicated GCC extensions for TM (i.e., GCC-4.7), have significantly increased TM's traction, paving the way for its mainstream adoption.

Disjoint-access parallelism (or DAP) [22] is a long studied property that assesses the ability of a TM implementation to avoid any contention on shared objects (also called *base objects*), between transactions that access disjoint data sets. Roughly speaking, DAP prescribes that non-conflicting transactions should not contend on the same memory locations, thus minimizing expensive inter-processor coordination and enhancing scalability [5, 11]. Several consistency criteria have been proposed for TM, such as *opacity* [16], *virtual world consistency* [21], and *TMS1* [13], which are widely adopted because they all prevent erroneous computations that can occur as a consequence of observing arbitrarily stale snapshots during the transaction execution.

All the above correctness criteria require serializability of committed transactions[1], but the existing TM literature has established that having DAP and serializable committed transactions in the same TM implementation is impossible under certain conditions for read-only transactions or different progress guarantees [4, 28, 15, 14, 8]. In particular, Attiya et al. [4] proved that a TM cannot be weak DAP (a weaker version of the original DAP [22]), while ensuring minimal progressiveness [18] for write transactions (a progress condition weaker than obstruction-freedom [20]), and providing invisible and wait-free read-only transactions if the correctness guarantee is (Strict) Serializability [7] or Snapshot Isolation [6] (and hence opacity, virtual world consistency, or TMS1). More recently, Bushkov et al. [8] proved

[1]Opacity also requires serializability of all (including pending and aborted) transactions.

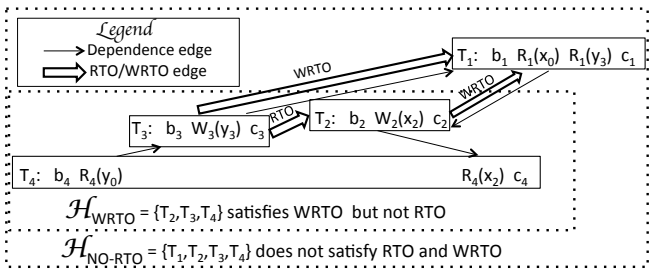

Figure 1: Histories accepted or rejected by WRTO.

the impossibility of implementing a strict DAP [15] (a stronger version of the original DAP [22]) TM that guarantees obstruction-freedom and Weak Adaptive Consistency, which is weaker than opacity, virtual world consistency, and TMS1.

However, the invisibility of read-only transactions as well as their ability to always commit in a finite number of steps (wait-freedom), are desirable requirements for enhancing the performance of TM in case of read-dominated workloads.

For this reason, in this paper we seek an answer to the following questions: *what are the correctness and progress guarantees that a DAP TM algorithm can ensure while having invisible and wait-free read-only transactions?* And for those guarantees, *which are the unavoidable costs to consider in terms of both time and space complexity?*

Along the path that will lead us to answer the above questions, we present two impossibility results and lower bounds that should be taken into account while designing a DAP TM. Specifically, in Section 3 we show that if one defines as target correctness criterion *any* criterion that includes Real-Time Order (RTO) then, independently of the particular isolation level ensured among concurrent transactions, it is impossible to ensure wait-free read-only transactions, obstruction-free write transactions, and the weakest form of DAP. We also show that, even when assuming weakly progressive write transactions [17] we still have an impossibility result if we enforce read-only transactions to be invisible.

These results highlight the need for relaxing RTO to implement a DAP TM that guarantees wait-free and invisible read-only transactions. Such an observation leads us to introduce a weaker variant of RTO, namely *Witnessable Real-Time Order* (WRTO), which demands that the RTO relation between two transactions is enforced only if they exhibit a direct conflict or they are executed by the same process. WRTO preserves some desirable properties of the classic RTO, including that if a transaction T running solo issues a read on a transactional object x, T is guaranteed to return the version of x produced by the last transaction that committed before T's beginning and updated x. On the other hand, WRTO admits schedules in which a set of sequential transactions (e.g., T_2, T_3 in Figure 1) accessing disjoint data sets can be observed in an arbitrary order by a concurrent transaction T_4 (as in the history $\mathcal{H}_{WRTO}$ of Figure 1), which possibly contradicts their RTO relations.

The WRTO property is indeed designed to cope with DAP TM implementations, as it demands that RTO is enforced only among conflicting (and hence non-disjoint) transactions, or trivially among transactions executed by the same process. In fact, these transactions can use shared base-objects involved in a conflict and thread-local base objects as witnesses to truck RTO relations.

In Section 4 we show that, by adopting WRTO and by restricting serializability to committed write transactions, it is indeed possible to design a TM with wait-free and invisible read-only transactions, which guarantees the strongest variant of DAP, strong progressiveness for write transactions, and a consistency criterion strictly weaker than opacity, i.e., Extended Update Serializability [1] (EUS). Informally, EUS guarantees serializability of committed write transactions, and it ensures that each transaction observes a state produced by a serializable schedule. Despite the theoretical relevance of this algorithm, we note that it comes with overheads, which can hinder its practical relevance. This observation motivates us to investigate whether such costs could be avoided, or are inherently implied by any DAP design. In Section 5 we prove that these costs are actually necessary, by deriving two lower bounds on the space and time complexity of any DAP TM that guarantees wait-free and invisible read-only transactions, WRTO, and obstruction-freedom or weak progressiveness for write transactions, when considering a consistency criterion strictly weaker than EUS, i.e., Consistent View [1]. Informally, Consistent View allows a specific type of non-serializable schedules, but it ensures that transactions read from a causally consistent snapshot, and prevents observing the effects of aborted transactions. In particular, such a TM has: *i)* a read-only time complexity equal to $\Omega(k \cdot N_o)$, where k is the number of versions a transactional object can have, and N_o is the total number of transactional objects; and *ii)* a space complexity equal to $\Omega(min(N_o, N_p))$ for the meta data associated with each transactional object version, where N_p is the maximum number of concurrent processes executing.

These lower bounds have implications of both theoretical and practical nature. From a theoretical perspective, they prove the optimality of the space and time complexity of the algorithm that we present. From a practical perspective, they provide useful guidelines to aim future research in the area by suggesting that, in order to allow the implementation of efficient DAP TM, lowering the correctness property is not enough (although remaining as strong as Consistent View), rather either WFRO or IRO should be sacrificed.

2. FORMALISM AND ASSUMPTIONS

System and Transaction Execution Model. We consider an asynchronous shared memory system composed of N_p processes $p_1, \ldots, p_{N_p}$ that communicate by executing transactions on N_o shared objects, which we call transactional objects, and may be faulty by crashing (i.e., slow down or block indefinitely). We use the term transactional objects to distinguish them from base objects, which are used to encapsulate any information (data and meta data) associated with a transactional object, and are manipulated via primitive operations (*primitives*, hereafter). We say that a primitive is non-trivial if it may change the value of a base object (e.g., *Compare-And-Swap* or *Store*), otherwise it is trivial (e.g., *Load*). Also primitives are atomic and wait-free.

A transaction starts with a *begin* operation, and can be followed by a sequence of *read* and *write* operations on transactional objects not known a priori, and finally completed by either a *commit* or an *abort* operation. A transaction only performing read operations is named read-only; otherwise it is called write. We denote with x_i the version of the transactional object x committed by transaction T_i, where i is an index that univocally identifies a transaction. We de-

note by op_i an operation issued by T_i, and by OP_i the set of operations issued by T_i, which is assumed to be totally ordered. We denote a write operation of T_i on a transactional object x as $W_i(x_i)$; and we use the notation $R_i(x_j)$ to indicate that a transaction T_i has read a version x_j of x written by transaction T_j. We say that two operations op_i and op_j are *conflicting* if they access a common transactional 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 $\prec_{\mathcal{H}}$ preserves the ordering of the operations of each transaction T_i ($\prec_{\mathcal{H}} \supseteq \bigcup_{i=1}^{n} OP_i$); and for any two conflicting operations op_i, $op_j \in \mathcal{H}$, either $op_i \prec_{\mathcal{H}} op_j$ or $op_j \prec_{\mathcal{H}} op_i$. $\mathcal{H}$ implicitly induces a total order $\ll$ on the committed versions of a given transactional object [1]. Specifically, the order of the versions associated with that object follows the order of the write operations successfully executed on it.

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, with $i \neq j$, if there exist two operations op_i, $op_j \in OP_{\mathcal{H}}$ such that $op_i \prec_{\mathcal{H}} op_j$ or $op_j \prec_{\mathcal{H}} op_i$. We distinguish three types of edges depending on the type of conflicts between T_i and T_j: *i) Direct read-dependence edge*, if there exists a transactional 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 $T_i \xrightarrow{wr} T_j$. *ii) Direct write-dependence edge*, if there exists a transactional object x such that both $W_i(x_i)$ and $W_j(x_j)$ are in $\mathcal{H}$ and x_j immediately follows x_i in the total order defined by $\ll$ on x. We say that T_j *directly write-depends* on T_i and we use the notation $T_i \xrightarrow{ww} T_j$. *iii) Direct anti-dependence edge*, if there exists a transactional 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 $\ll$ on x. We say that T_j *directly anti-depends* on T_i and we use the notation $T_i \xrightarrow{rw} T_j$.

Configurations, Events and Executions. A *configuration* is a tuple characterizing the status of each process, and of the base objects at some point in time. On the other hand, an *event* is a step performed by a process in a configuration, which encompasses the execution of a local computation, the application of a primitive on a base object, and a change to the status of the process. An event is generated by an operation of a transaction and in general an operation of a transaction can generate one or more events. An *execution interval* is a sequence $\Psi_i \cdot \phi_i \cdot \Psi_{i+1} \cdot \phi_{i+1} \cdot \Psi_{i+2} \cdot \phi_{i+2} \cdot \ldots$ that alternates configurations Ψ_k and events ϕ_k, and where the sequence $\Psi_k \cdot \phi_k$ generates the configuration Ψ_{k+1}. An event ϕ_k is said to be pending when configuration Ψ_{k+1} has not been generated yet. We also refer an *execution* E to as an execution interval starting from an initial configuration Ψ_0, and we say that two executions are *indistinguishable* to a process p_i if p_i observes the same sequence of configurations and steps in both the executions.

Since an execution E is actually a low-level history of configurations and events that are generated by operations of transactions in a history $\mathcal{H}$, we may also want to derive a history $\mathcal{H}$ from one of the possible executions, say E. In particular, given an execution E, we define $E|\mathcal{H}$ as the history derived from E by removing all the events and configurations, and where each sequence of events (which may possibly contain one event) is replaced by the corresponding operation that generated it.

Two events contend on a base object o if they both access o, and at least one of them applies a non-trivial primitive to o. Two processes concurrently contend on a base object o if they have pending events at the same configuration, which contend on o.

DAP and Invisible Reads. We consider 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 contend on a common base object only if they access some common transactional object; while a TM ensures WDAP if two concurrent transactions concurrently contend on a common base object only if they are not *disjoint-access*. Two transactions are not *disjoint-access*, according to the definition provided in [4], if there is a path that connects the two transactions in the undirected version of the DSG. A read operation R_i performed by T_i is called *invisible* if it does not apply any non-trivial primitive on any base object. Otherwise it is called *visible*. A read-only transaction performing only invisible reads is also called invisible (or IRO).

Progress Guarantees. A TM is *strongly progressive* [17] if *i)* transactions 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 *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 by Attiya et al. [4]: a TM is *wait-free* [19] if any transaction executed by a non-faulty process eventually commits in a finite number of steps despite the behavior of concurrent transactions[2]. We consider processes to be non-parasitic, i.e., they eventually request the commit of every transaction that they start unless they crash or the transaction is aborted by the TM [9]. Hereafter, we refer to a read-only transaction that guarantees wait-freedom as WFRO; also, if such a read-only transaction also provides invisible reads, then we name it WFIRO (i.e., the union of WFRO and IRO).

Correctness guarantees. Consistent View proscribes the anomalies *G1a*, *G1b*, *G1c*, and *G-single* as defined in [1]. In particular, proscribing *G1a* means that values created by transactions that abort cannot be observed. Proscribing *G1b* does not allow the observation of intermediate non-committed values. *G1c* is not allowed when there is no 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 transactions. Consistent

[2]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*, the reader can refer to the work in [9].

View prevents the G-single anomaly, hence it avoids that $DSG(\mathcal{H})$ contains an oriented cycle with exactly one anti-dependence edge.

Extended Update Serializability (EUS) [27] is stronger than Consistent View, and it can be seen as the union of Consistent View and Serializability of committed write transactions. EUS proscribes the anomalies *G1a*, *G1b*, and *G1c*, as well as the anomaly *Extended G-update*, such that the $DSG(\mathcal{H}^{upc}_{T_k})$ graph built on the committed write transactions in $\mathcal{H}$ plus a generic transaction T_k in $\mathcal{H}$ contains an oriented cycle with one or more anti-dependence edges.

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 $\mathcal{H}_{NO-RTO}$ of Figure 1, in which two read-only/executing/aborted transactions (e.g., T_4 and T_1 in $\mathcal{H}_{NO-RTO}$) are allowed to observe in different orders the commits of non-conflicting write transactions (e.g., T_2 and T_3 in $\mathcal{H}_{NO-RTO}$).

Real-Time Order (RTO) is a partial order defined over a transaction history $\mathcal{H}$, noted $\prec^{RTO}_{\mathcal{H}}$, which reflects the happened-before relations among transactions in a history. A transaction T_q is ordered before T_k in RTO, $T_q \prec^{RTO}_{\mathcal{H}} T_k$, 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 and transactions executed by the same process. Formally $T_q \prec^{WRTO}_{\mathcal{H}} T_k$ if $T_q \prec^{RTO}_{\mathcal{H}} T_k$, and T_q and T_k are connected by a direct dependence edge in $DSG(\mathcal{H})$. 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 $T_q \prec^{RTO}_{\mathcal{H}} T_k$ (respectively $T_q \prec^{WRTO}_{\mathcal{H}} T_k$), then the resulting graph does not contain cycles involving T_q and T_k.

An example history ensuring WRTO but not RTO is the history $\mathcal{H}_{WRTO}$ shown in Figure 1, where transaction T_4 runs concurrently with two write transactions T_3 and T_2, and T_2 runs sequentially after T_3. Since T_2 and T_3 neither conflict nor are executed by the same process, then $T_3 \prec^{RTO}_{\mathcal{H}} T_2$ holds. RTO is not ensured because T_4 observes the committed versions of T_2 but not those of T_3, and hence DSG($\mathcal{H}_{WRTO}$) plus the RTO edge from T_3 to T_2 contains a cycle that includes T_3 and T_2. On the contrary, a history that does not ensure WRTO and RTO is the history $\mathcal{H}_{NO-RTO}$ in Figure 1. Transaction T_1 runs sequentially after transaction T_2, and the two transactions conflict, hence $T_2 \prec^{WRTO}_{\mathcal{H}} T_1$ holds. However, T_1 does not observe the version of x committed by T_2, and thus DSG($\mathcal{H}_{NO-RTO}$) plus the WRTO edge from T_2 to T_1 contains a cycle that includes T_2 and T_1.

3. IMPOSSIBILITY RESULTS ON DAP AND RTO

In this section we seek an answer to the following question: *consider a DAP TM that guarantees RTO; can such a TM provide WFRO and IRO?* We do so by proving in Theorem 1 that a DAP TM cannot guarantee both RTO and WFRO if write transactions are obstruction-free. The result assumes only the RTO relation as the correctness guarantee and it is independent of the visibility of read-only transactions. In Theorem 2 we show that the impossibility of combining RTO and WFRO holds if the progress requirement of write transactions is weak progressiveness and read-only transactions are invisible (IRO). The intuition to prove these two results is the following: we show that any DAP TM that guarantees WRTO must accept a history that violates RTO. This history is illustrated in Figure 1 through $\mathcal{H}_{WRTO}$, where the RTO between the two non-conflicting transactions T_2 and T_3 is violated due to the return values of T_4's read operations. To show that $\mathcal{H}_{WRTO}$ is accepted by any DAP TM that guarantees WRTO and WFRO, we prove that such a TM must produce the return values of the read operations in $\mathcal{H}_{WRTO}$.

The proofs of the theorems rely on the following lemmas. In Lemma 1 we prove that in a WDAP TM two concurrent disjoint-access transactions cannot (even non-concurrently) contend on a common base object. It is worth noting that this result is not directly entailed by the definition of WDAP, because WDAP only precludes such transactions to *concurrently* contend on a common base object. The proof of Lemma 1 (which is provided in [26] due to space constraints) is based on the indistinguishability of detecting if the two *concurrent* disjoint-access transactions are *concurrently* contending on a common base object. In addition, this result differs from the result of Lemma 2 in [4], which proves that in a WDAP TM two disjoint-access transactions cannot contend on a common base object if they are *non-concurrent*.

Then Lemma 2 (whose proof is still provided in [26] due to space constraints) uses the result of Lemma 1 to prove that in a WDAP TM two conflicting transactions cannot contend on a common base object before the conflict actually occurs. Finally Lemma 3 (whose proof is still provided in [26]) relies on Lemma 2 to show why the read-only transaction T_4 in $\mathcal{H}_{WRTO}$ must execute as shown in Figure 1 in a WDAP TM that guarantees WRTO and WFRO. An interesting study on the relation between the impossibility results proved in this section and the existing impossibility results in literature is reported in Section 6.

LEMMA 1. *In a WDAP TM two concurrent disjoint-access transactions cannot contend on a common base object.*

As said, Lemma 2 asserts that in a WDAP TM, two transactions that conflict (either directly or transitively) cannot contend on a common base object before the time in which a path connecting the two transactions in the conflict graph is created. The proof is based on the indistinguishability of the following two scenarios: one where a conflict actually occurs; and the other where one of the two operations generating the conflict never occurs.

LEMMA 2. *Let T_i and T_j be two transactions in a history $\mathcal{H}$ generated by a WDAP TM. Assume that there exists only one path connecting T_i and T_j in DSG($\mathcal{H}$), which consists of a direct edge connecting T_i and T_j corresponding to two conflicting operations op_i and op_j, issued respectively by T_i and T_j. The operation op_i generates the event ϕ_i, and op_j generates ϕ_j. Further, assume that T_i executes ϕ_i after the commit of T_j. Then, if T_i and T_j contend on a common base object at time t, the conflict between them must occur at time $t' \leq t$.*

Lemma 3 asserts that a read operation of a wait-free read-only transaction has to always return at least the latest version of an object committed at the time the read operation on that object started. As before, the proof is based on indistinguishability arguments. Its core argument is that, in a WDAP TM a read operation of a read-only transaction T_i, which reads a version preceding the latest committed version x_j, is not able to distinguish whether x_j has been committed before T_i began or not.

LEMMA 3. *Given a WDAP TM that guarantees WRTO and WFRO, for each read-only transaction T_i executed by the TM, a read operation $R_i(x)$ by T_i returns the last version x_j committed on x at the time $R_i(x)$ starts its execution, if x_j has been committed by a transaction T_j that conflicts with T_i only due to the operations $R_i(x)$ and $W_j(x)$.*

We now prove Theorem 1, which states that a DAP TM cannot guarantee both RTO and WFRO if the progress requirement of write transactions is obstruction-freedom.

THEOREM 1. *No WDAP, obstruction-free TM that guarantees WFRO can ensure RTO.*

PROOF. The proof follows by contradiction and we assume that two different transactions are executed by two distinct processes. By contradiction, we assume that there exists a TM providing WDAP, obstruction-free write transactions, WFRO and such that $\forall \mathcal{H}$ accepted by the TM, $\mathcal{H}$ preserves RTO, and hence WRTO. Consider the history $\mathcal{H}_{WRTO}$ as executed in Figure 1. T_3 is a write transaction that runs solo and writes the version y_3 of y and then commits. T_2 begins after T_3, runs solo, writes x_2 of x, and then commits. T_4 is a read-only transaction that reads the initial version of y, i.e., y_0, before T_3 begins, and executes a read operation $R_4(x)$ on x, after T_2's commit. Both T_2 and T_3 cannot be aborted in $\mathcal{H}_{WRTO}$ because the TM is obstruction-free.

Since x_2 is the last version of x committed before the execution of $R_4(x)$, and the only path connecting T_2 and T_4 in DSG($\mathcal{H}_{WRTO}$) is the edge associated with the conflict of $R_4(x)$ and $W_2(x)$, then $R_4(x)$ has to return the version x_2 of T_2 by the result of Lemma 3. That is because the assumed WDAP TM provides WFRO and WRTO, which is implied by RTO (assumed by contradiction). Note that, even assuming the visibility of T_4's reads, T_3 can neither abort due to the detection of the conflict with $R_4(y_0)$, which is now visible, (as T_3 runs solo), nor wait for the completion of T_4 (as T_3 could be indefinitely blocked if the process executing T_4 crashed).

However, 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 $DSG(\mathcal{H}_{WRTO})$.

Therefore we have shown that a WDAP, obstruction-free TM, that guarantees WFRO and WRTO necessarily violates RTO (as the TM cannot reject the history $\mathcal{H}_{WRTO}$). But WRTO is strictly weaker than RTO, which means that to guarantee RTO the TM should guarantee WRTO, and thus there cannot exist a WDAP, obstruction-free TM that guarantees WFRO and RTO. □

Next we question the possibility to ensure RTO in a WDAP TM when considering weak progressiveness as the progress guarantee for write transactions. The answer is still negative (as we show in Theorem 2, whose formal proof is still in [26]) if we require WFIRO, which includes invisible read-only transactions. The proof is analogous to the one of Theorem 1, but further takes into account that write transactions cannot detect a conflict with read-only transactions due to the invisibility of the latter. To do so, we rely on the indistinguishability between the execution in Figure 1 and an analogous execution where T_4 does not take any step.

THEOREM 2. *No WDAP, weakly progressive TM that guarantees WFIRO can ensure RTO.*

It is interesting to relate the above results to at least two existing DAP TM algorithms, namely TLC [5] and the SDAP TM proposed in [3]. The latter TM guarantees opacity (and hence RTO) and 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. For this reason, write transactions are not obstruction-free and, hence, the TM in [3] does not contradict Theorem 1. TLC [5] is another SDAP TM implementation that guarantees invisible read-only and strongly progressive transactions. However, TLC does not guarantee WFRO, thus again one of the hypothesis of Theorem 2 is not met.

4. A SDAP TM WITH WITNESSABLE REAL-TIME ORDER

In Section 3 we have shown that RTO cannot be guaranteed by a WDAP TM that provides WFIRO. Here we provide a possibility result by adopting WRTO: it is possible to implement a WFIRO TM that guarantees the strongest variant of DAP, strong progressiveness for write transactions, and a correctness criterion whose semantics is very close to those provided by opacity or virtual world consistency. This consistency criterion, known in the literature as Extended Update Serializability (EUS) [1, 27], guarantees the serializability of the history of committed write transactions — hence ensuring that the state of the TM is updated without anomalies. Further, analogously to opacity, virtual world consistency, and TMS1, EUS provides Consistent View.

The details of such a TM are shown in the technical report [26] and are omitted from this paper for space constraints, although we provide an overview in the next Section 4.1. However intuitively the above properties are achieved as follows. Committed write transactions can be guaranteed to be serializable without sharing any global information but just leveraging meta data associated with transactional objects, by adopting a scheme similar to the DAP version of TL2 [5]. On the other hand, Consistent View and WRTO can be ensured by combining a multi-version scheme that allows a read operation by a transaction T to never incur wait conditions, and to always return the right version v such that v was not committed by a transaction T' that causally follows T, i.e., v is a correct state observable by T. This is achievable without sharing any global information (e.g., a global clock, which would violate DAP) and without applying any modification during the execution of a read operation (invisible read-only transactions) by just exploiting meta data associated with the committed versions, in order to detect the causal dependencies of commit events.

4.1 Algorithm Overview

Data structures. The proposed algorithm relies on vector clocks as identifiers of the snapshots committed and as references to determine which of the available object versions should be returned by read operations.

Each process p_i maintains a scalar, tc, that stores the timestamp associated with the last commit of process p_i; and a vector clock of size N_p, $maxS$, where $maxS[k]$ records the maximum value of process p_k's tc as known by p_i. Intuitively, at each step of the execution of a transaction T processed by p_i, $maxS$ identifies the commit events of the transactions serialized before T. Further, each transaction T maintains a vector clock, $upperS$, which identifies the commit events of the transactions serialized after T.

Commit events. In order to univocally identify the commit events in a totally decentralized way, the versions created upon commit are associated with two identifiers, i.e., cid and S. The former is the identifier of the process having committed those versions, while the latter is the vector clock that identifies the committed snapshot containing the versions created by this commit event. Therefore the cid-th entry of vector S associated with a version ver is the value of tc at the time ver was committed by p_{cid}. Intuitively, the remaining entries $S[j]$, with $j \neq cid$, capture the causal dependencies developed by the transaction that committed ver with the transactions executed by the other processes.

Read logic. The version visibility logic relies on both $upperS$ and $maxS$ to determine whether a commit event should be visible by a transaction T. As already mentioned, the commit events that precede or are equal to $maxS$ correspond to transactions that were already serialized before T, and are hence considered as visible by T; the commit events that do not precede or are equal to $upperS$ correspond to transactions that were already serialized after T, and are hence non-visible by T. Finally, the remaining commit events can be potentially visible by T and are said to be *uncertain*.

As an example, we provide in Figure 2 a graphical representation of the meaning of $maxS$ and $upperS$ for a transaction T at a certain point of its execution by process p_0 in a configuration of three processes p_0, p_1 and p_2. Given the current value of $maxS$, transaction T can determine that the transactions that generated the commit events having $S \in$ { {1,0,0}, {0,2,0}, {0,1,2} } are serialized before T. They are therefore in the *past* of T. On the contrary, due to the value of $upperS$, T cannot observe the commit events with $S \in$ { {0,5,3}, {0,4,3} }, as their vector clocks S are concurrent with T's $upperS$. This is because, as a result of the values observed by T's read operations so far, T has been serialized before the transactions that generated these commit events. These transactions are therefore in T's future.

Upon each read operation on a transactional object, the corresponding set of versions is retrieved and the uncertain ones are analyzed to determine whether the corresponding commit is visible to T, i.e., serialized before T. If an uncertain version ver is not visible, the transaction creating ver is serialized after T, therefore T moves its $upperS$ backwards so as to exclude the vector clock S of ver. Otherwise, that transaction creating ver has to be serialized before T. The latter case will trigger an advancement of $maxS$ only when T completes, in order to include the vector clock S of ver in the visible snapshot of the transactions executing after T on the same process.

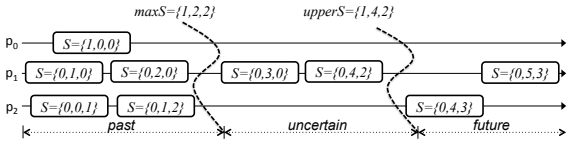

Figure 2: Example of usage of vectors $maxS$ and $upperS$ for a transaction T. A rectangle at process p_i identifies a commit event by process p_i and is tagged with the corresponding vector clock S.

In order to determine the visibility of an uncertain version ver, if any exists, the transaction read-set is re-validated to verify whether it is safe to serialize the transaction that committed ver before the reading transaction. An uncertain version ver of object o is non-visible to transaction T, if T has read a version ver_1 of object o', and: *i)* a version ver_2 of o' exists that is more recent than ver_1; *ii)* the commit of ver causally depends on the commit of ver_2 (the vector clock S of ver is greater than or equal to the vector clock S of ver_2). If the two conditions above are met, and ver had been serialized before T since its beginning, then T should have observed ver_2 (and not ver_1) at the time it issued a read on o'.

Finally, the version returned by a read operation is the most recent of the visible versions.

Commit phase. The algorithm allows a write transaction T to commit only if *i)* it is able to acquire all locks on the objects read and written by T; and *ii)* T's read-set has not been invalidated by a concurrent transaction, i.e., for each version read by T, that version is still the last committed version of the corresponding object. This is sufficient to guarantee the serializability of the history of the write transactions, as required by EUS. Furthermore, the S vector clock associated with a snapshot committed by T is set to be greater than any of the S vector clocks of the objects read and overwritten by T. This allows to track causality dependencies among transactions, which is necessary to guarantee the correct execution of read operations.

5. TIME AND SPACE COMPLEXITY OF DAP TM IMPLEMENTATIONS

Despite the theoretical relevance of the algorithm in Section 4, we note that it comes with a non-negligible overhead: each version of a transactional object has to keep a vector clock as big as the maximum number of concurrent processes N_p; a read operation of a read-only transaction may accomplish $k \cdot N_o$ steps in order to return the correct value, where k is the number of versions of a transactional object, and N_o is the total number of transactional objects. In the following we prove that these costs are necessary even considering only Consistent View and WRTO.

5.1 Time Complexity

In this section we investigate the costs a WDAP TM has to pay if it guarantees WFIRO when the correctness guarantee is Consistent View. Roughly, this means that every read operation does not observe any incorrect state as long as write transactions produce correct states. Note that the

latter is needed because Consistent View does not enforce write transactions to always produce correct states (unlike EUS). We also suppose that the TM guarantees WRTO, in order to rule out any trivial implementation of WFIRO, in which the read-only transactions only observe the initial version of the transactional objects.

The intuition behind the result is that such a TM cannot always ensure a constant cost for handling any read operation because a read cannot rely on any shared timestamp to determine the correct state to be observed (as for TL2 [12] or LSA [29]). In fact, if that was the case, then the TM would trivially violate DAP. In particular, what we show is that, the maximum number of steps performed by any read operation of read-only transactions for determining whether a version can be observed without violating Consistent View is $\Omega(N_o)$. Let us introduce the definition of read-only time complexity of a TM.

DEFINITION 1. *The read-only time complexity of a TM is the maximum number of steps performed by any read operation of any read-only transaction executed by the TM.*

Given the previous definition, we show that the read-only time complexity of the above TM is $\Omega(k \cdot N_o)$, where k is the maximum number of versions associated with any transactional object. For convenience in the proof we rely on the function CHECKVERSION, which determines whether a live read-only transaction that executes a read operation on an object x is allowed to observe the version x_i of x. More formally, given a configuration Ψ_k, a read-only transaction T_{ro} live in Ψ_k, and a version x_i that exists in Ψ_k, CHECKVERSION(Ψ_k,T_{ro},x_i) returns true if T_{ro} can return x_i by executing a read operation on x without violating Consistent View; it returns false otherwise.

The proof is based on the following scenario: a read-only transaction T_{ro} whose read-set has been invalidated by a write transaction T_i and such that it executes a read operation on an object x after the invalidation. Here, we focus on how to determine the correct version of x to be observed. Clearly, T_{ro} cannot observe the version x_i if x_i has been written by T_i because the resulting DSG would contain the following cycle: $T_{ro} \overset{rw}{\dashrightarrow} T_i \overset{wr}{\longrightarrow} T_{ro}$. In addition, T_{ro} cannot arbitrarily decide not to observe x_i because it could have been written by a transaction committed before T_{ro}'s begin, hence violating WRTO.

To determine this, the function CHECKVERSION(Ψ_k,T_{ro},x_i) executed by a WDAP TM has to apply trivial primitives on the base objects associated with the objects that T_{ro} already read (i.e., the validation of T_{ro}'s read-set). Let us assume that at the time T_{ro} executes its read operation on x, it has already executed read operations on all the other objects, then the number of steps performed by CHECKVERSION(Ψ_k,T_{ro},x_i) is $\Omega(N_o)$. Now, if we suppose that there are k versions associated with any transactional object, the number of steps performed by the T_{ro}'s read operation on x is equal to $\Omega(k \cdot N_o)$, which is also the read-only time complexity of the TM. In the following proof we assume either obstruction-free or weakly progressive write transactions.

THEOREM 3. *Given a WDAP multi-version TM that guarantees WFIRO, Consistent View, WRTO, and either obstruction-freedom or weak progressiveness for write transactions, the read-only time complexity of the TM is $\Omega(k \cdot N_o)$, where k is the number of versions a transactional object can have, and N_o is the total number of transactional objects.*

PROOF. By hypothesis we consider a WDAP multi-version TM that guarantees WFIRO, Consistent View, WRTO, and either weakly progressive or obstruction-free write transactions. Also, let $E = E_1 \cdot E_2 \cdot E_3 \cdot E_4$ be the following execution of the TM in which only the processes p_1 and p_2 take steps. E_1: transaction T_1, executed by process p_1, writes to all the N_o shared objects some values and then it commits. E_2: transaction T_{ro}, executed by process p_2, performs $\Theta(N_o)$ read operations on $\Theta(N_o)$ different objects from some set Q_{RO}. Due to WRTO, these operations return the last available versions (the ones committed by T_1). E_3: transaction T_i, executed by process p_1, performs $\Theta(N_o)$ write operations on $\Theta(N_o)$ different objects from some set $Q_{WR} = Q_1 \cup \{y\}$, such that $Q_1 \cap Q_{RO} = \emptyset$ and $y \in Q_{RO}$. Then T_i commits. E_4: transaction T_{ro} performs a read operation on object $x \in Q_1$ (thus $x \notin Q_{RO}$) and it returns the version x_j such that $x_j \ll x_i$ and x_i is the version of x committed by T_i. We also assume that the process running in an execution interval (i.e., E_1, E_2, E_3, E_4) runs solo in that interval.

E is an admissible execution for the assumed TM because no transaction in E violates neither the liveness nor the correctness properties guaranteed by the TM. Regarding the former, no read operation of T_{ro} can generate an abort since read-only transactions are wait-free; and both the write transactions T_1 and T_i have to commit in E because of the following. T_1 has to commit in E because it runs solo (as required by obstruction-freedom) and it does not encounter any conflict (as required by weak progressiveness); T_i has to commit under obstruction-freedom because it runs solo, and it has to commit under weak progressiveness (even though it encounters a conflict with T_{ro}) for the following reason: T_i has to commit also in an execution $E' = E_1 \cdot E_3$, where T_{ro} is not processed, and E' and E are indistinguishable to p_1 since T_{ro} is invisible. Regarding the correctness guarantee, the history $E|\mathcal{H}$ does not violate Consistent View since the graph $DSG(E|\mathcal{H})$ does not contain any oriented cycle (including those with exactly one anti-dependence edge) and transactions only observe committed values. The graph is the union of the paths $T_1 \overset{wr}{\longrightarrow} T_{ro} \overset{rw}{\dashrightarrow} T_i$ and $T_1 \overset{ww}{\longrightarrow} T_i$.

We define the configuration Ψ_k as the result of the processing of $E_1 \cdot E_2 \cdot E_3$. Now, by contradiction we assume that the number of steps for executing CHECKVERSION(Ψ_k,T_{ro},x_i) is $\mathcal{O}(N_o)$. Our assumption means that, when T_{ro} executes the read operation on x in E_4, p_2 executes $\mathcal{O}(N_o)$ steps by applying trivial primitives on $\mathcal{O}(N_o)$ base objects, in order to determine that x_i cannot be visible. In fact, if that read operation returned x_i, then Consistent View would be violated. Our assumption also entails that T_i has to apply non-trivial primitives on $\mathcal{O}(N_o)$ base objects among the ones accessed by T_{ro} to guarantee the correct execution of CHECKVERSION(Ψ_k,T_{ro},x_i) in E. In fact, if that is not the case and T_i's execution applies non-trivial primitives on $N_o - \mathcal{O}(N_o)$ base objects among the ones that are not accessed by T_{ro}, then $E|\mathcal{H}$ can violate Consistent View because CHECKVERSION(Ψ_k,T_{ro},x_i) can return true by erroneously applying trivial primitives on only $\mathcal{O}(N_o)$ base objects.

We now show that our assumption leads us to a contradiction since either the worst case of $\mathcal{O}(N_o)$ steps is not enough to preserve Consistent View or WDAP is not guaranteed. To do that, let us consider an execution $E'' = E_1 \cdot E_2 \cdot \bar{E}_3 \cdot E_4$,

where $\bar{E_3}$ differs from E_3 only because Q_{WR} is equal to $Q_1 \cup \{z\}$ in $\bar{E_3}$, where $z \in Q_{RO}$. First, we have to notice that E'' is admissible for the assumed TM because of the same arguments provided for E. Then, since T_i does not write on y in $\bar{E_3}$, but it writes on z, it has to apply non-trivial primitives on the $\mathcal{O}(N_o)$ base objects which T_{ro} applies trivial primitives on. In fact, if that is not the case and T_i's execution applies non-trivial primitives on $N_o - \mathcal{O}(N_o)$ base objects among the ones that are not accessed by T_{ro}, then $E''|\mathcal{H}$ can violate Consistent View because CHECKVERSION(Ψ_k,T_{ro},x_i) can return true by erroneously applying trivial primitives on only $\mathcal{O}(N_o)$ base objects.

However, if that is the case, we can build $\Theta(N_o)$ executions like E'', i.e., E^*, where z is replaced by an object among the ones in Q_{RO}, and T_i's execution in E^* performs its write operations by applying non-trivial primitives on $\mathcal{O}(N_o)$ base objects among the ones accessed by T_{ro}. Since the executions are $\Theta(N_o)$ and the base objects accessed by T_{ro} are only $\mathcal{O}(N_o)$, there exist at least two executions of them where the transactions T_i in the two executions access at least one common base object among the $\mathcal{O}(N_o)$ previously accessed by T_{ro}. Without loss of generality we assume that those two executions are E and E''.

Therefore T_i in E and T_i in E'' access at least one common base object among the $\mathcal{O}(N_o)$ previously accessed by T_{ro}, and the access to that base object depends on the write operations on y and z respectively, in order to be distinguishable to the process executing T_{ro} from an execution where T_i does not write any object in Q_{RO} (T_i does not invalidate T_{ro}'s read-set). Consequently, such a TM must accept an execution with two concurrent non-conflicting transactions T_i^* and T_i^+, one writing only y and the other writing only z, which both access at least one common base object among the $\mathcal{O}(N_o)$ previously accessed by T_{ro}, thus violating WDAP due to Lemma 1.

Therefore, since the assumption on the cost of $\mathcal{O}(N_o)$ for CHECKVERSION(Ψ_k,T_{ro},x_i) entails a TM that violates either WDAP or Consistent View, then CHECKVERSION(Ψ_k,T_{ro},x_i) must have a cost equal to $\Omega(N_o)$. Thus the maximum number of steps performed by any read operation of read-only transactions for determining whether a version can be observed without violating Consistent View is $\Omega(N_o)$. If we suppose that there are k versions of any transactional object, then the read-only time complexity of the assumed TM is equal to $\Omega(k \cdot N_o)$. □

5.2 Space Complexity

We now investigate the space complexity of a SDAP multi-version TM that guarantees WFIRO, namely the necessary meta data associated with each version of a transactional object. We prove a lower bound that holds assuming Consistent View as correctness guarantee, WRTO, and assuming either weak progressiveness or obstruction-freedom for write transactions. Note that in this lower bound we consider SDAP, which matches also the variant of DAP ensured by the algorithm in Section 4, rather than WDAP as before.

As in the proof of the time complexity, we assume that the TM also guarantees WRTO to rule out any trivial implementation of WFIRO, in which the read-only transactions only observe the initial transactional state. In order to derive an implementation-independent proof, we use an innovative technique, which shows an equivalence between the problem of detecting cycles containing exactly one anti-dependence edge 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 with the current history. Due to the existence of the SDAP requirement, this check has to be performed without indiscriminately accessing all the information associated with the conflict graph, but only extracting this information via the base objects associated with the accessed transactional objects.

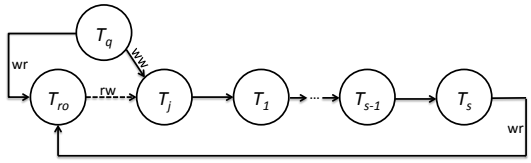

Figure 3: T_{ro} creating a cycle C with exactly one anti-dependence edge in $DSG(\mathcal{H})$.

THEOREM 4. *Given a SDAP TM that guarantees WFIRO, Consistent View, WRTO and either obstruction-freedom or weak progressiveness for the write transactions, then the space complexity for each version of a transactional object 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 transactional object 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 3, in which the first transaction to execute in absence of concurrency is T_q, which commits the version x_q. As we assume obstruction-freedom or weak progressiveness, the TM cannot refuse T_q's commit.

After T_q's commit a read-only transaction T_{ro} issues a read on object x. As we assume WFRO, the read operation of T_{ro} must eventually return some value, and by WRTO, the value returned has to be x_q. Before T_{ro} takes any other step, 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 write transactions $\mathcal{T}$=$\{T_1,\ldots,T_i,\ldots,T_{s-1},T_s\}$ is executed sequentially. Each transaction $T_i \in \mathcal{T}$ issues the following operations in $\mathcal{H}$: T_i starts, reads an 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 assume that transaction T_1 and the transactions in $\mathcal{T}$ run solo, they must commit if we assume obstruction-freedom. On the other hand, if we assume weak progressiveness, T_j may abort due to the presence of an anti-dependency on T_{ro}. However, since we assume invisible read-only transactions, T_j cannot detect the occurrence of this conflict and, also in this case, it cannot abort.

Now, let us 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 or not. Note that, since T_{ro} has already developed an anti-dependence towards T_j, 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 assume 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 a write transaction that commits before T''s begin. Also, in the assumed history, T_{ro} cannot be aware of having developed an anti-dependence towards T_j, as we assume 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 the existence of the 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 assume 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 apply trivial operations to 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).

We now assume a total number of transactional objects greater than the maximum number of concurrent processes (i.e., $N_o \geq N_p$). In a SDAP TM, 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 $T_i \xrightarrow{wr} T_{i+1} \in \text{DSG}(\mathcal{H})$ 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, respectively receive, of a message $m_{i,i+1}$ in DS from p_i, respectively 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 $T_{ro} \overset{rw}{\dashrightarrow} T_j \in \text{DSG}(\mathcal{H})$ developed by the operations $R(x_q)$ and $W(x_j)$, we can associate the events of send, respectively receive, of a message $m_{j,ro}$ in DS from p_j, respectively 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 transformed the problem of determining if 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 if the two messages $m_{j,ro}$ and $m_{s,ro}$ are causally ordered [23], namely $m_{j,ro} \prec_{DS} m_{s,ro}$. Thanks to the result in [24], in a distributed system of N_p processes such as DS, given two events e and e', $e \prec_{DS} e'$ iff $\mathcal{V}(e) < \mathcal{V}(e')$, where $\mathcal{V}(e)$ (respectively $\mathcal{V}(e')$) is the vector clock of size N_p associated to e (respectively e'). Hence the base objects need to have a space capacity equal [10] to $\Omega(N_p)$.

We now go over the case where the total number of transactional objects is lesser than the maximum number of concurrent processes (i.e., $N_o < N_p$). In this case, each version encodes the most recent state of the TM at the time the version is created. By doing so, a transaction has all the sufficient information to decide whether a version can be read or not. Note that, storing the most recent state of the whole TM in each version is also (trivially) necessary in this case because keeping the entire state but one transactional object could let the TM miss a dependency on that object, thus generating a violation of Consistent View. For instance, a TM that follows this intuition has been provided by Ardekani et al. in [2].

We showed that a lower bound on the space complexity of each version of a SDAP TM that guarantees Consistent View, WFIRO, WRTO and obstruction-freedom (or weak progressiveness), is $\Omega(m)$, where $m = min(N_o, N_p)$. □

6. RELATED RESULTS

Theorem 1 has relations with a lower bound [4] defined on the number of non-trivial primitives applied by read-only transactions in a Strict Serializable and DAP TM (where DAP here is the one defined in [22]) that ensures minimal progressiveness for write transactions. This lower bound defines a necessary condition for the visibility of the read-only transactions to ensure WFRO and Strict Serializability, that is not sufficient in the case of obstruction-free write transactions because it is superseded by Theorem 1 in that case (since Strict Serializability demands RTO [25, 16]).

On the other hand, that necessary condition is also a sufficient condition in case write transactions are weakly progressive. As an evidence of that, Attiya and Hillel proposed such a TM in [3], which is even able to guarantee opacity.

If we suppose a parasitic-free TM, the impossibility presented by Perelman et al. [28] uses the same assumptions of Theorem 2, and proves that such a TM cannot guarantee Strict Serializability. In fact, it is easy to prove that, in a parasitic-free TM, MV-permissiveness may imply wait-free and invisible read-only transactions, and weakly progressive write transactions. However, the result is weaker than the one of Theorem 2, since Strict Serializability demands RTO.

The impossibility result by Guerraoui and Kapalka [15] rules out the possibility to combine a SDAP TM and obstruction-free transactions if the target correctness guarantee is Serializability. The authors prove the result by only considering write transactions, and therefore we can consider their impossibility still valid if the target correctness guarantee is EUS. Later this same impossibility result [15] has been superseded by the results of the PCL theorem [8], which relaxes the correctness guarantee assumed in [15]. Our algorithm overcomes those two results by assuming a different progress guarantee for write transactions.

Our lower bound on the time complexity is related to the lower bound presented in [16] where the authors prove that the maximum number of steps performed by any operation of any transaction executed by a progressive (which is stronger than weakly progressive) single-version TM is $\Omega(N_o)$ if the TM guarantees opacity.

7. CONCLUSIONS

In this paper we enriched the literature on impossibility, possibility, and lower bound results of DAP TM. We

presented two impossibility results ruling DAP TM that guarantees real-time order relations and a set of desirable progress properties. Furthermore, we provided a possibility result: an algorithm that circumvents the existing DAP limitations by providing a strong correctness property (i.e., EUS), a variant of the real-time order we introduced in this paper (i.e., *Witnessable Real-Time Order*), and the same set of progress properties as before. We also provided two lower bounds on the space and time complexity of such a DAP TM, which highlight that efficient DAP TM implementations should necessarily sacrifice either invisible reads or wait-freedom for read-only transactions because lowering the correctness level is not enough.

8. ACKNOWLEDGMENTS

The authors thank Rachid Guerraoui and Srivatsan Ravi for the useful feedbacks. Peluso, Palmieri and Ravindran are supported in part by US Air Force Office of Scientific Research under grants FA9550-14-1-0143 and FA9550-14-1-0187, and US National Science Foundation under grant CNS-1217385. Romano is supported in part by FCT projects UID/CEC/50021/2013 and EXPL/EEI-ESS/0361/2013.

9. REFERENCES

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

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

[3] H. Attiya and E. Hillel. A single-version STM that is multi-versioned permissive. *Theory Comput. Syst.*, 51(4):425–446, 2012.

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

[5] H. Avni and N. Shavit. Maintaining Consistent Transactional States Without a Global Clock. In *SIROCCO*, pages 131–140, 2008.

[6] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A Critique of ANSI SQL Isolation Levels. In *SIGMOD*, pages 1–10, 1995.

[7] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[8] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. The PCL Theorem. Transactions cannot be Parallel, Consistent and Live. In *SPAA*, pages 178–187, 2014.

[9] V. Bushkov, R. Guerraoui, and M. Kapalka. On the liveness of transactional memory. In *PODC*, pages 9–18, 2012.

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

[11] T. David, R. Guerraoui, and V. Trigonakis. Everything you always wanted to know about synchronization but were afraid to ask. In *SOSP*, pages 33–48, 2013.

[12] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *DISC*, pages 194–208, 2006.

[13] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.

[14] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *PODC*, pages 115–124, 2012.

[15] R. Guerraoui and M. Kapalka. On Obstruction-free Transactions. In *SPAA*, pages 304–313, 2008.

[16] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, 2008.

[17] R. Guerraoui and M. Kapalka. The Semantics of Progress in Lock-based Transactional Memory. In *POPL*, 2009.

[18] R. Guerraoui and P. Romano. *Transactional Memory. Foundations, Algorithms, Tools, and Applications*. Springer, 2015.

[19] M. Herlihy. Wait-free synchronization. *ACM TOPLAS*, 13(1), 1991.

[20] M. Herlihy, V. Luchangco, and M. Moir. Obstruction-Free Synchronization: Double-Ended Queues As an Example. In *ICDCS*, pages 522–529, 2003.

[21] D. Imbs and M. 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.

[22] A. Israeli and L. Rappoport. Disjoint-access-parallel Implementations of Strong Shared Memory Primitives. In *PODC*, pages 151–160, 1994.

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

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

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

[26] S. Peluso, R. Palmieri, P. Romano, B. Ravindran, and F. Quaglia. Disjoint-Access Parallelism: Impossibility, Possibility, and Cost of Transactional Memory Implementations. Technical report, Virginia Tech, 2015.

[27] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When Scalability Meets Consistency: Genuine Multiversion Update-Serializable Partial Data Replication. In *ICDCS*, pages 455–465, 2012.

[28] D. Perelman, R. Fan, and I. Keidar. On maintaining multiple versions in STM. In *PODC*, pages 16–25, 2010.

[29] T. Riegel, P. Felber, and C. Fetzer. A Lazy Snapshot Algorithm with Eager Validation. In *DISC*, pages 284–298, 2006.

[30] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.