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

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 describes 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

\(^1\)Opacity also requires serializability of all (including pending and aborted) transactions.

\(PODC'15\), July 21–23, 2015, Donostia-San Sebastián, Spain.
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 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 \( T_1 \) (as in the history \( H_{WRTO} \) 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 track 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 \( \Omega(k \cdot N_v) \), where \( k \) is the number of versions a transactional object can have, and \( N_v \) is the total number of transactional objects; and ii) a space complexity equal to \( \Omega(min(N_v, N_r)) \) for the meta data associated with each transactional object version, where \( N_r \) 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 \( N_v \) 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 denote 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 \( H \) over a set of transactions \( \{ T_1, \ldots, T_n \} \) is a partial order \( \prec_H \) defined over the set of operations \( OP = \bigcup_{i=1}^{n} OP_i \) such that \( \prec_H \) preserves the ordering of the operations of each transaction \( T_i \) (\( \prec_H \supseteq \bigcup_{i=1}^{n} OP_i \)); and for any two conflicting operations \( op_i, op_j \in H \), either \( op_i \prec_H op_j \) or \( op_j \prec_H op_i \). \( H \) 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 \( DSG(H) \) on a history \( H \) (as in [1, 7]) as a direct graph with a vertex for each transaction \( T_i \) in \( H \) and a directed edge from \( T_i \) to \( T_j \), with \( i \neq j \), if there exist two operations \( op_i, op_j \in OP \) such that \( op_i \prec_H op_j \) or \( op_j \prec_H op_i \). 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 \( R_j(x_j) \) are in \( H \). We say that \( T_j \) directly read-depends on \( T_i \) and we use the notation \( T_i \rightarrow_{rw} T_j \). 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 \( H \) and \( x_i \) immediately follows \( x_j \) in the total order defined by \( \ll \) on \( x \). We say that \( T_i \) directly write-depends on \( T_j \) and we use the notation \( T_i \rightarrow_{wr} T_j \). iii) Direct anti-dependence edge, if there exists a transactional object \( x \) and a committed transaction \( T_j \) in \( H \), with \( k \neq i \) and \( k \neq j \), such that both \( R_i(x_i) \) and \( W_j(x_j) \) are in \( H \) and \( x_i \) immediately follows \( x_j \) 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 \( T_i \rightarrow_{tw} T_j \).

**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 \( \Psi_{1} \phi_{1} \Psi_{2} \phi_{2} \Psi_{3} \phi_{3} \cdots \) where \( \Psi_k \) alternates configurations and events and are denoted by \( \Psi_{k+1} \) and events \( \phi_k \) where the sequence \( \Psi_{k}, \phi_{k} \) generates the configuration \( \Psi_{k+1} \) and \( \phi_{k} \) is said to be pending when configuration \( \Psi_{k+1} \) has not been generated yet. We also refer an execution \( E \) to as an execution interval starting from an initial configuration \( \Psi_{0} \), and we say that two executions are indistinguishable to a process \( p \) if \( p \) 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 \( H \), we may also want to derive a history \( H \) from one of the possible executions, say \( E \). In particular, given an execution \( E \), we define \( E/H \) 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 \( H \) executed by the TM, a transaction \( T_i \in H \) 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. 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 WFIRO; also, if such a read-only transaction also provides invisible reads, then we name it WFIRO (i.e., the union of WFIRO 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(H) graph built on the history H', where H' is derived from H by removing aborted and executing transactions. Consistent View prevents the G-single anomaly, hence it avoids that DSG(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(H \(H'_{G}^{RTO}\)) graph built on the committed write transactions in H plus a generic transaction \(T_k\) in 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 \(T_1\) belonging to one among the following categories: aborted, executing or read-only transactions. EUS admits non-serializable histories, as illustrated in history \(H_{NO-RTO}\) of Figure 1, in which two read-only/executing/aborted transactions (e.g., \(T_3\) and \(T_1\) in \(H_{NO-RTO}\)) are allowed to observe in different orders the commits of non-conflicting write transactions (e.g., \(T_2\) and \(T_3\) in \(H_{NO-RTO}\)).

Real-Time Order (RTO) is a partial order defined over a transaction history H, noted \(\prec_{RTO}\), which reflects the happened-before relations among transactions in a history. A transaction \(T_q\) is ordered before \(T_k\) in RTO, \(T_q \prec_{RTO} T_k\), if the commit operation \(\phi_q\) of \(T_q\) precedes the begin operation \(\phi_k\) of \(T_k\) in \(H\). 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, \(T_q \prec_{WRTO} T_k\) if \(T_q \prec_{RTO} T_k\), and \(T_q\) and \(T_k\) are connected by a direct dependence edge in DSG(H).

A history H preserves RTO (respectively WRTO) if, after having included in DSG(H) a direct edge \(\forall T_q, T_k\) in H, such that \(T_q \prec_{RTO} T_k\) (respectively \(T_q \prec_{WRTO} T_k\)), 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 \(H_{WRTO}\) 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 \(T_3 \prec_{RTO} T_2\) holds. RTO is not ensured because \(T_4\) observes the committed versions of \(T_2\) but not those of \(T_3\), and hence DSG(H \(H_{RTO}\)) plus the RTO edge from \(T_1\) to \(T_2\) contains a cycle that includes \(T_1\) and \(T_2\). On the contrary, a history that does not ensure WRTO and RTO is the history \(H_{NO-RTO}\) in Figure 1. Transaction \(T_1\) runs sequentially after transaction \(T_2\), and the two transactions conflict, hence \(T_2 \prec_{WRTO} T_1\) holds. However, \(T_1\) does not observe the version of \(x\) committed by \(T_2\), and thus DSG(H \(H_{NO-RTO}\)) 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 \(H_{WRTO}\), where the RTO between the two non-conflicting transactions \(T_2\) and \(T_3\) is violated due to the return values of \(T_1\)'s read operations. To show that \(H_{WRTO}\) 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 \(H_{WRTO}\). 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 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 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 relies on Lemma 2 to show why the read-only transaction \(T_4\) in \(H_{WRTO}\) 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.

**Proof.** By the definition of WDAP TM, two concurrent disjoint-access transactions \(T_i, T_j\) cannot concurrently contend on a common base object. We need to show that \(T_i\) and \(T_j\) cannot contend even non-concurrently. To do so, let us assume by contradiction that they contend non-concurrently on a base object \(o\). We denote with \(\phi_1, \phi_2\) the first two pairs of events processed by \(T_i\) and \(T_j\), respectively, that contend on \(o\). It follows that \(\phi_1\) and \(\phi_2\) are non-concurrent events — otherwise, \(T_i\) and \(T_j\) would concurrently contend, which is prohibited by WDAP.

Figure 2(a) depicts an execution of \(T_i\) and \(T_j\), where we have denoted with \(\alpha\) the set of events processed by \(T_i\) before \(\phi_1\), and with \(\alpha'\) those processed by \(T_j\) before \(\phi_2\). We suppose that \(\alpha\) and \(\alpha'\) do not overlap in time. Now let us consider, in Figure 2(b), an execution in which we have postponed the processing of \(\phi_2\) by \(T_j\) to take place concurrently with \(\phi_1\). By construction, the execution interval \(\alpha\alpha'\) of the execution in
Figure 2: Undistinguishable execution intervals in WDAP.

Figure 2(a) and the execution interval $\alpha \alpha'$ of the execution in Figure 2(b) are indistinguishable to both $p_i$ and $p_j$ since they are the same execution interval. Therefore, $p_i$ and $p_j$ are about to perform respectively $\phi_i$ and $\phi_j$ after $\alpha \alpha'$ in both the executions and, since $\phi_i$ and $\phi_j$ contend on a common base object in Figure 2(a), they contend on the same base object in Figure 2(b) as well. In the execution of Figure 2(b), however, $\phi_i$ and $\phi_j$ contend concurrently, which is impossible as we assume that the TM is WDAP. Therefore, we obtain a contradiction and the proof holds.

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 $H$ generated by a WDAP TM. Assume that there exists only one path connecting $T_i$ and $T_j$ in DSG($H$), 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 $\phi_i$, and $op_j$ generates $\phi_j$. Further, assume that $T_i$ executes $\phi_i$ 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 $t' \leq t$.

**Proof.** Assume without loss of generality that $T_i$ and $T_j$ are two transactions executed by process $p_i$ and process $p_j$, respectively. Let $t'$ be the time at which event $\phi_i$ is processed by $T_i$, hence the conflict between $T_i$ and $T_j$ occurs at time $t'$. We distinguish two cases depending on whether $T_i$ and $T_j$ are concurrent or not.

In case $T_i$ and $T_j$ are concurrent, since $T_i$ executes $\phi_i$ after $T_j$ commits, we can decompose their execution intervals in three phases, as depicted in Figure 3(a). The possible execution intervals of $T_i$ can be expressed by means of i) the execution interval $\alpha$, which comprises all the events generated by $T_i$ since its beginning and prior to the execution of $\phi_i$ (if any); ii) the event $\phi_i$; and iii) the execution interval $\beta$, which comprises all the events generated by $T_i$ after $\phi_i$ (if any). Analogously, transaction $T_j$ is constituted by the event $\phi_j$ that is processed in between two execution intervals $\alpha'$ and $\beta'$. Note that since we assume that $\phi_i$ is executed by $T_i$ after $T_j$’s commit and that the two transactions are concurrent, it follows that $\alpha$ and the execution interval of $T_j$ are concurrent.

Since the operations issued by a transaction are not known a priori, at any time $t'' < t' \ p_i$ is not able to distinguish whether it is processing the execution interval $\alpha$ of transaction $T_i$ or the execution interval $\alpha$ of a transaction $T''$ that completes without executing $\phi_i \cdot \beta$, like the one of Figure 3(b). Therefore if at time $t''$, $\alpha$ of $T_i$ contended on a base object accessed by $T_j$, $\alpha$ of $T''$ would have to do the same. However, since $T''$ and $T_j$ are two concurrent and disjoint-access transactions, they cannot either concurrently or non-concurrently contend on a common base object due to the definition of WDAP and Lemma 1 respectively. Consequently, $T_i$ and $T_j$ can only contend on a common base object at time $t \geq t'$.

If $T_i$ and $T_j$ are not concurrent, by the assumption that $T_i$ executes $\phi_i$ after $T_j$ commits, we can depict their executions as in Figure 4(a). Transaction $T_i$ is constituted by an execution interval $\alpha$, followed by the event $\phi_i$, and another execution interval $\beta$, and such that $\alpha$ is not concurrent with any execution interval of $T_j$. Transaction $T_j$ is constituted by the event $\phi_j$ that is processed in between two execution intervals $\alpha'$ and $\beta'$. As for the previous case, since the operations executed by a transaction are not known a priori, at time $t'' < t' \ p_i$ is not able to distinguish whether it is processing the execution interval $\alpha$ of transaction $T_i$ or the execution interval $\alpha$ of a transaction $T''$ that completes without executing the interval $\phi_i \cdot \beta$, as depicted in Figure 4(b).

Therefore, if at time $t''$, $\alpha$ of $T_i$ contended on a base object accessed by $T_j$, $\alpha$ of $T''$ would do the same. However, $T''$ and $T_j$ cannot contend on a common base object by the result of Lemma 2 in [4]. In fact, $T''$ and $T_j$ are two consecutive and non-conflicting transactions executed by a WDAP TM starting from a quiescent configuration (i.e., a configuration with no pending event), and there is no path connecting $T''$.
and $T_j$ in the conflict graph of the minimal execution interval containing $T^*$ and $T_j$ by hypothesis. Consequently, $T_i$ and $T_j$ cannot contend on a common base object at time $t'' < t'$, meaning that $T_i$ and $T_j$ can contend on a common base object only at time $t > t'$.

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 $W_j(x)$.

**Proof.** We assume transactions $T_i$ and $T_j$ as the ones defined in the hypothesis. Therefore $T_i$ executes a read operation on object $x$, i.e., $R_i(x)$, and $T_j$ is the last transaction that commits by writing version $x_j$ on $x$ (i.e., $T_j$ executes $W_j(x_j)$), and it is the last one to commit a version on $x$ before $R_i(x)$ starts. Further, the only path connecting $T_i$ and $T_j$ in the DSG is the edge associated with the conflict of $R_i(x)$ and $W_j(x)$. We distinguish two cases depending on whether $T_i$ and $T_j$ are concurrent or not.

If $T_i$ and $T_j$ are consecutive (i.e., not concurrent) and $R_i(x)$ follows the commit of $T_j$, then the begin of $T_i$ too follows the commit of $T_j$. Therefore we have $T_i \prec_{\text{WRTO}}^\text{WRTO} T_j$, because $T_i$ and $T_j$ are conflicting and $T_j \succ_{\text{RTO}}^\text{RTO} T_i$. Consequently, $R_i(x)$ has to return a version, meaning that it cannot abort due to the hypothesis of WFRO, and that version must be $x_j$. In fact, if that was not the case and $T_i$ returned $x_i < x_j$, then the DSG on the history containing $T_i$ and $T_j$ augmented with WRTO edges would contain both an edge reflecting $T_i \prec_{\text{WRTO}}^\text{WRTO} T_j$ and a path from $T_i$ to $T_j$ reflecting the fact that $T_i$ missed version $x_j$, by violating the assumption of a TM that guarantees WRTO.

If $T_i$ and $T_j$ are concurrent, we can assume both $T_i$ and $T_j$ as depicted in Figure 3(a) where $\phi_i$ and $\phi_j$ are the events generated by the operations $R_i(x)$ and $W_j(x)$ respectively. In this scenario, the hypothesis of Lemma 2 are verified, and therefore $T_i$ and $T_j$ cannot contend on a common base object till $T_i$ executes $\phi_i$ and hence $R_i(x)$. This means that $\alpha$ and all the execution intervals of $T_j$ cannot contend on a common base object, and therefore $p_i$ and $p_j$ have to perform the same steps in all the aforementioned executions up to the execution of $\phi_i$ and hence of $R_i(x)$. More in detail, the following execution intervals are indistinguishable to $p_i$: i) $\alpha$ concurrent to $\alpha' \cdot \phi_j \cdot \beta'$; ii) $\alpha' \cdot \phi_i \cdot \beta'$; and iii) $\alpha' \cdot \phi_j \cdot \beta' \cdot \alpha$.

As a consequence, $T_j$ cannot abort because of the hypothesis of WFRO, and if we assumed that $R_i(x)$ returned a version $x_v \ll x_j$ in the executions of Figure 3(a), where $\alpha$ is concurrent to $\alpha' \cdot \phi_j \cdot \beta'$, then $R_i(x)$ should return $x_v$ in the execution of Figure 4(a) as well, which is the case of $\alpha' \cdot \phi_j \cdot \beta' \cdot \alpha$. However, since in the last execution we have $T_j \prec_{\text{WRTO}}^\text{WRTO} T_i$ because $T_i$ and $T_j$ are conflicting and $T_j \succ_{\text{RTO}}^\text{RTO} T_i$, returning $x_v \ll x_j$ violates WRTO as proved in the first case (i.e., when $T_i$ and $T_j$ are assumed to be consecutive). Therefore, $R_i(x)$ returns $x_j$ also when $T_i$ and $T_j$ are concurrent.

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 $\forall H$ accepted by the TM, $H$ preserves RTO, and hence WFRO. Consider the history $H_{\text{WRTO}}$ 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. $T_3$ 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 $H_{\text{WRTO}}$ 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 $\text{DSG}(H_{\text{WRTO}})$ 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 $H_{\text{WRTO}}$ does not preserve RTO because: i) $T_3 \prec_{\text{RTO}}^\text{WRTO} T_2$; and ii) there exists the oriented path $T_2 \xrightarrow{\text{wr}} T_4 \xrightarrow{\text{wr}} \cdots \cdots \cdots \xrightarrow{\text{wr}} T_3$ from $T_2$ to $T_3$ in $\text{DSG}(H_{\text{WRTO}})$.

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 $H_{\text{WRTO}}$).
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) 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.

**Proof.** As in the proof of Theorem 1, also this 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, weakly progressive write transactions, WFIRO and such that $\forall H$ accepted by the TM, $H$ preserves RTO, and hence WRTO. Consider the history $H_{WRTO}$ as executed in Figure 1. In this proof we do not require $T_2$ and $T_3$ to run solo. Therefore, $T_3$ is a write transaction that writes the version $y_3$ of $y$ and then commits. $T_2$ begins after $T_3$, 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 $T_3$'s begin, 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 $H_{WRTO}$ because the TM is weakly progressive, namely they are allowed to abort only if they encounter a conflict. In fact, even though $T_2$ and $T_3$ conflict only with $T_4$, they should commit in an analogous execution where $T_4$ does not take any step, and that execution is indistinguishable from the one in Figure 1 to the processes executing $T_2$ and $T_3$ because $T_4$ is invisible.

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 the conflict graph of DSG 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 leveraging the result of Lemma 3 when running. That is because the assumed WDAP TM provides WFIRO and WRTO, which is implied by RTO (assumed by contradiction).

However, history $H_{WRTO}$ does not preserve RTO because:

i) $T_3 \prec_{H_{WRTO}} T_2$; and ii) there exists the oriented path $T_2 \xrightarrow{wr} T_4 \xrightarrow{...} T_3$ from $T_2$ to $T_3$ in $DSG(H_{WRTO})$.

Therefore we have showed that a WDAP, weakly progressive TM, that guarantees WFIRO and WRTO necessarily violates RTO (as the TM cannot reject the history $H_{WRTO}$). 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, weakly progressive TM that guarantees WFIRO and 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.

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, $tc_i$ that stores the timestamp associated with the last commit of process $p_i$; and a vector clock of size $N_p$, max$S$, where max$S[k]$ records the maximum value of process $p_i$’s $tc$ as known by $p_i$. Intuitively, at each step of the execution of a transaction $T$ processed by $p_i$, max$S$ identifies the commit events of the transactions serialized before $T$. Further, each transaction $T$ maintains a vector clock, upper$S$, 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 \( \text{cid} \)-th entry of vector \( S \) associated with a version \( \text{ver} \) is the value of \( \text{tc} \) at the time \( \text{ver} \) was committed by \( \text{p}_{\text{cid}} \). Intuitively, the remaining entries \( S[j] \), with \( j \neq \text{cid} \), capture the causal dependencies developed by the transaction that committed \( \text{ver} \) with the transactions executed by the other processes.

**Read logic.** The version visibility logic relies on both \( \text{upperS} \) and \( \text{maxS} \) 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 \( \text{maxS} \) 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 \( \text{upperS} \) 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 5 a graphical representation of the meaning of \( \text{maxS} \) and \( \text{upperS} \) 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 \( \text{maxS} \), 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 \( \text{upperS} \), \( T \) cannot observe the commit events with \( S \in \{ \{0,5,3\}, \{0,4,3\} \} \), as their vector clocks \( S \) are concurrent with \( T \)'s \( \text{upperS} \). 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 \( \text{ver} \) is not visible, the transaction creating \( \text{ver} \) is serialized after \( T \), therefore \( T \) moves its \( \text{upperS} \) backwards so as to exclude the vector clock \( S \) of \( \text{ver} \). Otherwise, that transaction creating \( \text{ver} \) has to be serialized before \( T \). The latter case will trigger an advancement of \( \text{maxS} \) only when \( T \) completes, in order to include the vector clock \( S \) of \( \text{ver} \) in the visible snapshot of the transactions executing after \( T \) on the same process.

In order to determine the visibility of an uncertain version \( \text{ver} \), if any exists, the transaction read-set is re-validated to verify whether it is safe to serialize the transaction that committed \( \text{ver} \) before the reading transaction. An uncertain version \( \text{ver} \) of object \( o \) is non-visible to transaction \( T \), if \( T \) has read a version \( \text{ver}_1 \) of \( o \), and: i) a version \( \text{ver}_2 \) of \( o \) exists that is more recent than \( \text{ver}_1 \); ii) the commit of \( \text{ver} \) causally depends on the commit of \( \text{ver}_2 \) (the vector clock \( S \) of \( \text{ver} \) is greater than or equal to the vector clock \( S \) of \( \text{ver}_2 \)). If the two conditions above are met, and \( \text{ver} \) had been serialized before \( T \) since its beginning, then \( T \) should have observed \( \text{ver}_2 \) (and not \( \text{ver}_1 \)) 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.

### 4.2 Algorithm Details

We now provide all the details of the algorithm.

In the following we use the binary relation \( \leq \) to define an order for both scalar values and vector clock values. In case of scalar values the relation is the standard \( \text{less-than-or-equal} \) 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 \( v_1[i] \leq v_2[i] \). If there exists also an index \( j \) such that \( v_1[j] \geq v_2[j] \), where \( < \) is the standard \( \text{less-than} \) relation defined for natural numbers, then \( v_1 < v_2 \) holds.

**Initialization.** \( \text{tc}, \text{maxS} \) and \( \text{upperS} \) are initialized respectively with \( 0, \{0,...,8\} \) and \( \{-1,...,-1\} \). \( \text{upperS} \) is reset each time a transaction starts.

**Write and read operations.** The write operation simply stores 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.

The read operation on \( v \) of transaction \( T \) processed by \( p_i \) (see Algorithm 1) checks first if \( v \) has been previously written by \( T \). In this case it returns the written value (lines 2-3). Otherwise, the versions' chain associated with \( v \) is traversed from the most recently committed version to the oldest one (lines 4 and 11-17). Specifically for each version \( \text{ver} \), \( \text{ver} \)'s snapshot \( S \) is compared to the vector clocks \( \text{upperS} \) of \( T \) and \( \text{maxS} \) of \( p_i \) (functions \( \text{nonVisible} \) and \( \text{unsafe} \) at lines 13, 23-26 and 28-35). If \( \text{maxS} \geq S \) (the condition at line 29 is false), which means that the process that is executing \( T \) has already observed a snapshot at least as recent as \( S \), then \( \text{ver} \) can be observed by \( T \). This corresponds to the case in which the returned version is in the past of \( T \).

Otherwise there are two scenarios in which the current version could not be readable by \( T \): when \( \text{upperS} < S \) on the significant entries (i.e., those different from -1) meaning that the condition at line 24 is true, or when \( \text{maxS} \) is less than or not comparable with \( S \) (the condition at line 29 is true). In these scenarios, in fact, reading the version could lead to a history that violates EUS.
Algorithm 1 Read operation of transaction $T$ on process $p_i$

1: Val read(Tx T, Var v)
2: if ∃ val': $< v, \text{val}' > \in T.writeSet then
3: return val'
4: [Val result, bool mostRecent] $\leftarrow$ doRead(T, v)
5: if $T.writeSet \neq \emptyset \land \text{mostRecent} = \bot$ then
6: $p, \text{max}S \leftarrow \max(p, \text{max}S, p, \text{max}S)$
7: abort(T)
8: else return
9: return result
10: doRead(Tx T, Var v)
11: Version ver $\leftarrow v.\text{mostRecentVers}$
12: while nonVisible(T, ver) $\lor$ unsafe(T, ver, v) do
13: if $T.upperS[\text{ver.cid}] \geq \text{ver.S[\text{ver.cid}]} \lor T.upperS[\text{ver.cid}] = -1$ then
14: $T.upperS[\text{ver.cid}] \leftarrow \text{ver.S[\text{ver.cid}]} - 1$
15: $\text{ver.ver.prev} \leftarrow T.upperS[\text{ver.cid}] - v, \text{ver} >$
16: return [\text{ver.value}, \text{ver} = v.\text{mostRecentVers}]
17: end while
18: $p, \text{max}S \leftarrow \max(p, \text{max}S, p, \text{max}S)$
19: $\text{T.readSet} \leftarrow T.\text{readSet} \cup < v, \text{ver} >$
20: return $\{\text{ver.value}, \text{ver} = v.\text{mostRecentVers}\}$
21: nonVisible(Tx T, Version ver)
22: if $\exists k: T.upperS[k] \neq -1 \land T.upperS[k] < \text{ver.S[k]}$ then
23: return $\top$
24: end if
25: unsafe(Tx T, Version ver, Var v)
26: if $\exists k: p, \text{max}S[k] < \text{ver.S[k]}$ then
27: return $\top$
28: end if
29: overwritten(Tx T, Var readV, Version readVer, Version target)
30: $\text{Version curr} \leftarrow \text{readV.mostRecentVers}$
31: if $\text{curr} \neq \text{readVer.do}$ then
32: if curr.$\leq$ target.S then
33: return $\top$
34: $\text{curr} \leftarrow \text{curr.prev}$
35: end while
36: return $\bot$

In the former scenario, $T$ cannot read ver because it belongs to a snapshot already skipped by $T$ due to a previous read, which has serialized $T$ before the transaction that committed ver. This is the case in which the snapshot identified by $S$ is in the future of $T$.

In the latter scenario, ver is uncertain for $T$. So $T$ has to check if by reading version ver, which implies serializing $T$ after the transaction $T'$ that committed this version, it is still possible to serialize all the reads already performed by the transaction after $T'$. For this reason, a re-validation of the read-set of $T$ is needed (lines 32-34 and 37-44) to check if there exists a version ver' that has been committed after any version in $T$'s read-set, and the snapshot that contains ver' is serialized before the snapshot that contains ver (which can be determined by comparing the vector clocks of their snapshots $S$ at line 40). In other words, $T$ can observe a version ver if the commit of ver does not causally depend on a commit that invalidated (i.e., had overwritten) a snapshot already observed by $T$ in its previous reads. In this aspect, the proposed algorithm shares similarities with LSA [29], 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.

There exists another additional case in which transaction $T$ has to discard a version in order to not violate wait-freedom for read-only transactions. In particular, that’s happen if an uncertain version ver is found and ver is not marked as committed while $v$ is still locked (lines 30-31). As it will be clearer in the description of the commit phase, that is the case in which the transaction committing ver, e.g., $T'$, has not completely finalized its commit phase yet. In that case, in fact, allowing $T$ to observe ver would imply forcing $T$ to possibly wait, upon a subsequent read, for the commit of another version written by $T'$.

After each read operation, an auxiliary vector clock associated with the current process, i.e., $\text{max}S$, is updated by computing the maximum between the snapshot $S$ of the returned version and $\text{max}S$. This implies a possible shrinking of the unexplored area for all the transactions they will follow the execution of $T$ on the same process. On the contrary, as soon as ver is discarded by a read operation by $T$ (lines 14-16), $\text{upperS}$ is changed accordingly by possibly moving the commit of ver from the uncertain area to the future of $T$.

Finally a transaction tracks the read versions in its read-set (line 19), and during a read operation only write transactions can be aborted if they cannot access the newest version of a read object (lines 5-8). This optimization allows to early abort update transactions whose validation would fail at commit time.

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 (lines 3-8). This means that read-only transactions always commit without any additional step. This is safe because, as required by EUS, they always observe a state produced by a serializable history of update transactions.

Next, a 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 (lines 11-14). Only after the successful acquisition of all the requested locks,
Algorithm 2 Commit operation of transaction $T$ on process $p_i$

1: $\text{commit}(T_x T)$
2: $p_i.maxS \leftarrow \max(p_i, maxS, p_i, maxS)$
3: if $T.writeSet \neq \emptyset$ then
4: VectorClock $newS \leftarrow p_i, maxS$
5: for $<v, val > \in T.writeSet$ do
6:     $bool\ locked \leftarrow tryLock(v)$
7:     if locked = $\top$ then
8:         abort($T$)
9:     else
10:         $newS \leftarrow \max(newS, v.mostRecentVers.S)$
11:     for $<v, version > \in T.readSet$ do
12:         $bool\ locked \leftarrow trySharedLock(v)$
13:     if locked = $\top$ then
14:         abort($T$)
15:     for $<v, version > \in T.writeSet$ do
16:         if version $\neq v.mostRecentVers$ then
17:             abort($T$)
18:     else
19:         $newS \leftarrow \max(newS, v.mostRecentVers.S)$
20:     $p_i.tc \leftarrow p_i.tc + 1$
21:     $newS[i] \leftarrow p_i, tc$
22:     $p_i.maxS \leftarrow \max(p_i, maxS, newS)$
23:     for $<v, val > \in T.writeSet$ do
24:         Version newVersion
25:         newVersion.value $\leftarrow$ val
26:         newVersion.S $\leftarrow$ newS
27:         newVersion.cid $\leftarrow i$
28:         $newS, v.mostRecentVers$ $\leftarrow$ newVersion
29:         $v.mostRecentVers.committed \leftarrow 1$
30:     $v.mostRecentVers$ $\leftarrow$ newVersion
31:     for $<v, val > \in T.writeSet$ do
32:         $v.mostRecentVers.committed $\leftarrow 1$
33:     for $<v, committed > \in T.readSet \cup T.writeSet$ do
34:         unlock($v$)

the transaction validates its read-set, by checking that the read versions are the last committed ones (lines 15-17), and flushes the write-set (lines 20-34).

Note that the same outcome may be obtained by avoiding the acquisition of locks on the objects in the transactions’ read-set, as implemented by other lock-based TMs, e.g., TL2 [12]. But the reason why we acquire locks on the read-sets of a transaction $T$, is to prevent a concurrent transaction $T'$ from committing a new version of an object read by $T$ and just after $T$ successfully validated that object. In this case, in fact, the commit of $T'$ could possibly complete even before the completion of the commit of $T$ even though $T'$ has to appear as serialized after $T$. The correct execution on a run like that is to force an abort of the transactions that try to read an object still locked by $T$ (e.g., TL2 [12]). However, our TM has to ensure wait-free read-only transactions, hence reads executed on locked objects can generate neither aborts nor delays.

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 (lines 10, 19 and 26). Further, the cid of those versions is equal to the identifier of the process committing the transaction (line 27).

After having completed the updates on the versions’ chains, a transaction marks the new versions as committed (lines 31-32) and it releases the previously acquired locks (lines 33-34).

4.3 Correctness Arguments

In the following we provide arguments on why the proposed algorithm ensures EUS and WRTO. We also discuss what mechanisms it employs to ensure liveness.

Regarding EUS, the acquisition of locks on read-set and write-set plus the validation of the read-set is used to serialize conflicting update transactions by preventing the possibility of cycles in the DSG of the history of the committed update transactions $H_{upc}$. Read-only transactions are guaranteed to observe (possibly different) linearizations of the partial order defined by DSG($H_{upc}$), since the necessity to advance $maxS$, which is used to determine the most recent version visible by a transaction, triggers the re-validation of the transaction read-set. This re-validation ensures that the previous reads executed by the transaction would have returned the same value, if they had been executed using the new value of $maxS$.

Intuitively, the algorithm guarantees WRTO, since, given two transactions $T_1, T_2 \in H$ such that $T_1 \prec^{WRTO} T_2$, a read operation by $T_2$ on an object written by $T_1$ never returns a committed version $ver$ such that $ver$ is serialized before $T_1$.

Concerning liveness, the proposed algorithm ensures wait-free read-only transactions and strong progressive update transactions. The former follows trivially from that we never block or abort a read-only transaction (and we are assuming parasitic-free histories). As for update transactions, if we exclude shared locks since they can be simultaneously acquired by read operations at commit time, they achieve strong progressiveness as the commit scheme that they adopt follows the lock-based scheme implemented in TL2 [12], which has already been proved to guarantee strong progressiveness in [17].

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 [20]). 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 $\Psi_k$, a read-only transaction $T_{ro}$ live in $\Psi_k$, and a version $x_i$ that exists in $\Psi_k$, CheckVersion($\Psi_k$, $T_{ro}$, $x_i$) 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: $T_i \xrightarrow{rw} T_{ro} \xrightarrow{ro} T_i$. 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 WRTOS.

To determine this, the function CheckVersion($\Psi_k$, $T_{ro}$, $x_i$) 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($\Psi_k$, $T_{ro}$, $x_i$) 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, WRTOS, and other 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, WRTOS, 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 WRTOS, these operations return the last available versions (the ones committed by $T_i$). $E_3$: 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 $x_j \leq x_i$ 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. $T_i$ 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 $p_i$ since $T_{ro}$ is invisible. Regarding the correctness guarantee, the history $E[H]$ does not violate Consistent View since the graph $DSG(E[H])$ 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 $T_i \xrightarrow{rw} T_{ro} \xrightarrow{ro} T_i$ and $T_i \xrightarrow{rw} T_i$.

We define the configuration $\Psi_k$ as the result of the processing of $E_1$, $E_2$, $E_3$. Now, by contradiction we assume that the number of steps for executing CheckVersion($\Psi_k$, $T_{ro}$, $x_i$) is $O(N_o)$. Our assumption means that, when $T_{ro}$ executes the read operation on $x$ in $E_4$, $p_2$ executes $O(N_o)$ steps by applying trivial primitives on $O(N_o)$ 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 $O(N_o)$ base objects among the ones accessed by $T_{ro}$ to guarantee the correct execution of CheckVersion($\Psi_k$, $T_{ro}$, $x_i$). In fact, if that is not the case and $T_i$’s execution applies non-trivial primitives on $N_o - O(N_o)$ base objects among the ones that are not accessed by $T_{ro}$, then $E[H]$ can violate Consistent View because CheckVersion($\Psi_k$, $T_{ro}$, $x_i$) can return true by erroneously applying trivial primitives on only $O(N_o)$ base objects.

We now show that our assumption leads us to a contradiction since either the worst case of $O(N_o)$ steps is not enough to preserve Consistent View or WDAP is not guaranteed. To do that, let us consider an execution $E'' = E_1 \cdot E_2 \cdot E_3 \cdot E_4$, where $E_3$ differs from $E_3$ only because $Q_{wr}$ is equal to $Q_1 \cup \{z\}$ in $E_3$, 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 $E_3$, but it writes on $z$, it has to apply non-trivial primitives on the $O(N_o)$ 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 $N_o - O(N_o)$ base objects among the ones that are not accessed by $T_{ro}$, then $E''[H]$ can violate Consistent View because Check-
VERSION($\Psi_k,T_{ro},x_1$) can return true by erroneously applying trivial primitives on only $O(N_o)$ base objects.

However, if that is the case, we can build $O(N_o)$ 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 $O(N_o)$ base objects among the ones accessed by $T_{ro}$. Since the executions are $O(N_o)$ and the base objects accessed by $T_{ro}$ are only $O(N_o)$, 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 $O(N_o)$ 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 $O(N_o)$ 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 between 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 $O(N_o)$ previously accessed by $T_{ro}$, thus violating WDAP due to Lemma 1.

Therefore, since the assumption on the cost of $O(N_o)$ for CHECKVERSION($\Psi_k,T_{ro},x_1$) entails a TM that violates either WDAP or Consistent View, then CHECKVERSION($\Psi_k,T_{ro},x_1$) 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 extract-
transmitted any information to \( T_j \) on the execution of its read on \( x_q \). Further, no other transaction could have notified \( T_\alpha \) of the existence of such anti-dependence.

Note that if \( T_\alpha \) ignored the possibility of having developed new anti-dependences when determining the visibility of \( d^{i+1}_j \), it could miss the existence of the cycle \( C \), and violate Consistent View. It follows that \( T_\alpha \) has to first validate its current read-set, which comprises only \( x_q \). This allows \( T_\alpha \) to detect the anti-dependence with \( T_j \), and poses \( T_\alpha \) with the problem to determine whether there exists an oriented path from \( T_j \) to \( T_\alpha \) (in which case \( d^{i+1}_j \) should not be observed). Note that since we assume a SDAP TM, \( T_\alpha \) needs to detect the existence of a path of direct dependencies from \( T_j \) to \( T_\alpha \), without however being able to apply trivial operations to any of the base objects that the transactions \( T_1, \ldots, T_{s-1} \) accessed (as \( T_\alpha \) 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 \( T_i \xrightarrow{\text{mod}} T_{i+1} \in \text{DSG}(H) \) developed by a pair of write/read operations on version \( d^{i+1}_j \) of transactional object \( d^{i+1}_j \), we can associate the events of send, respectively receive, of a message \( m_{i+1} \) 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 \( m_{i+1} \) can only be tagged with the information that \( T_i \) had stored in the base object associated with version \( d^{i+1}_j \) at the time in which \( T_i \) created it. Analogously for the direct anti-dependence edge \( T_{\alpha} \xleftarrow{\text{mod}} T_j \in \text{DSG}(H) \) developed by the operations \( R(x_i) \) and \( W(x_j) \), we can associate the events of send, respectively receive, of a message \( m_{\alpha,ro} \) in DS from \( p_{\alpha} \), respectively to \( p_{j} \). What triggers the sending of this message in this history is the fact that \( T_\alpha \) needs 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 \( T_\alpha \) in \( \text{DSG}(H) \) based on the information available to \( T_\alpha \), to the problem of having the process \( p_{\alpha} \) (that executes transaction \( T_\alpha \)) in DS to determine if the two messages \( m_{j,ro} \) and \( m_{\alpha,ro} \) are causally ordered [23], namely \( m_{j,ro} <_{\text{DS}} m_{\alpha,ro} \). Thanks to the result in [24], in a distributed system of \( N_p \) processes such as DS, given two events \( e \) and \( e' \), \( e <_{\text{DS}} e' \) iff \( \forall(e) < \forall(e') \), where \( \forall(e) \) (respectively \( \forall(e') \)) is the vector clock of size \( N_p \) associated to \( e \) (respectively \( e' \)). Hence the base objects need to have a space capacity equal to \( 10 \) 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 WFIRO and Strict Serializability, that is not sufficient in the case of obstruction-free transactions because it can be 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