Opacity vs TMS2: Expectations and Reality

[Technical Report]

Sandeep Hans, Ahmed Hassan, Roberto Palmieri, Sebastiano Peluso, and Binoy Ravindran

Virginia Tech

Abstract. Most of the popular Transactional Memory (TM) algorithms are known to be safe because they satisfy opacity, the well-known correctness criterion for TM algorithms. Recently, it has been shown that they are even more conservative, and that they satisfy TMS2, a strictly stronger property than opacity. This paper investigates the theoretical and practical implications of relaxing those algorithms in order to allow histories that are not TMS2. In particular, we present four impossibility results on TM implementations that are not TMS2 and are either opaque or strictly serializable, and one practical TM implementation that extends TL2, a high-performance state-of-the-art TM algorithm, to allow non-TMS2 histories. By matching our theoretical findings with the results of our performance evaluation, we conclude that designing and implementing TM algorithms that are not TMS2, but safe, has inherent costs that limit any possible performance gain.

1 Introduction

Transactional Memory (TM) [14] is a programming abstraction that ease the development of concurrent applications. Most of the popular TM algorithms (e.g., TL2 [4], NOrec [3], and LSA [19]) are proved to be correct because they do not violate opacity [11], the well-known criterion that requires each transaction (even a non-committed one) to *i)* read only committed values, and *ii)* behave as atomically executed at a single point between its beginning and its completion. However Doherty *et al.* [6] showed that most of the TM implementations that aim at satisfying opacity actually guarantee a strictly stronger correctness criterion, known as TMS2. In practice, TMS2 implementations reject executions that would not violate opacity while seeking for a tradeoff between the performance and complexity of the concurrency control implementation.

In this paper, we evaluate the costs and implications of having TM implementations that guarantee weaker conditions than TMS2, such as opacity. In particular, we focus on the simple execution pattern used in [6] to distinguish between TMS2 and opaque TM implementations. We name this execution pattern as *reverse-commit anti-dependency* (*RC-anti-dependency* in short). Intuitively, we say that a TM implementation allows RC-anti-dependency if it accepts the execution in Figure 1, where there is an anti-dependency between two committed update transactions T_1 and T_2 (i.e., T_2 overwrites T_1's previous read) and their

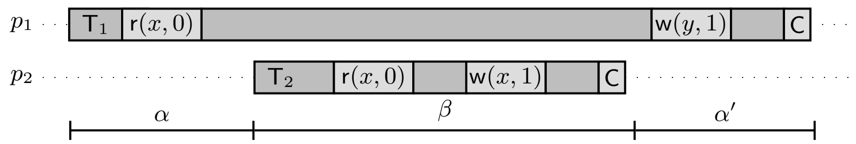


Fig. 1. An execution with RC-anti-dependency: $\mathsf{r}(x,0)$ denotes the read operation on x, $\mathsf{w}(x,1)$ denotes the write operation on x; C denotes a successful commit operation.

commit order is reversed with respect to the anti-dependency (i.e., T_2 starts its commit phase and completes it before T_1 does).

Designing a TM implementation that includes RC-anti-dependency looks appealing because, on the one hand, the execution pattern looks simple to accept, and on the other hand, performance is likely to improve thanks to the possibility of committing transactions that otherwise would abort due to a read invalidation. That is why embracing this pattern was one of the goals of previous attempts to increase the number of accepted executions, such as permissive TMs [9, 16], and TWM [5]. However, none of those attempts isolated the advantages and limitations of allowing RC-anti-dependency in a TM implementation. Specifically, permissive TMs aims at allowing all the possible schedules of execution within a certain correctness guarantee; and the latter relies on a class of input-acceptance [8] that mixes anti-dependency with multi-versioning and non-blocking read-only transactions. This paper is the first one that isolates RC-anti-dependency in order to assess the need of designing TM implementations that are not TMS2. Specifically, we provide a set of impossibility results on allowing RC-anti-dependency, and one possibility result, which is a concrete TM implementation, built on top of TL2, that allows RC-anti-dependency and confirms our theoretical claims.

We prove that in any strictly serializable [18] minimally progressive [11] TM implementation that allows RC-anti-dependency: *i)* read operations of update transactions must be visible; *ii)* either read-only transactions or the read-only prefix (i.e., all read operations before the first write) of update transactions must be visible. We also prove that if a strictly serializable TM that allows RC-anti-dependency has obstruction-free [13] read-only transactions, they must be visible. Finally, we prove that if we consider opacity rather than strict serializability, the visibility of read operations must be immediate and cannot be deferred to the commit phases of transactions. Table 1 summarizes our results. Due to space constraints, we defer the formal proofs to Appendix B.

	Consistency	**Progress**	**Impossibility**
Theorem 1	Strict Serializability	Minimal Progressiveness	Invisible Read Operations (update transactions)
Theorem 2	Strict Serializability	Minimal Progressiveness	Invisible Read-only Transactions Invisible Read Executions
Theorem 3	Strict Serializability	Obstruction-freedom (read-only transactions)	Invisible Read-only Transactions
Theorem 4	Opacity	Minimal Progressiveness	Invisible Read Executions

Table 1. Summary of the impossibility results presented in the paper.

As a possibility result, we present the design and implementation of a TM, named TL2-RCAD, which allows executions that are not TMS2, including the one in Figure 1, and limits the overhead of making visible reads by deploying specific algorithmic optimization. Evaluating TL2-RCAD we found that, contrary to expectations, the overall percentage of potential RC-anti-dependency executions is small. Thus the performance gain (if any) of allowing them is very limited in all the tested scenarios that include STAMP [17], the standard benchmark suite for TM, and even customized micro-benchmarks.

The paper is organized as follows. In Section 2, we introduce our basic model definitions. In Section 3, we present our impossibility results. TL2-RCAD's design is presented in Section 4, and its performance results are analyzed in Section 5. Section 6 overviews the related work, and Section 7 concludes the paper.

2 Preliminaries

System and Transaction Execution Model. We consider an asynchronous shared memory system composed of N processes $p_1, \ldots, p_N$ that communicate by executing transactions on 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*, or equivalently *objects*, to distinguish them from *base objects*, which are used to encapsulate any information (data and metadata) associated with transactional objects.

Each transaction is a sequence of accesses, reading from and writing to the set of objects. In particular a transaction T_j accesses objects with *operations* $\mathsf{op_{ij}} \in \{\mathsf{read}, \mathsf{write}, \mathsf{begin}, \mathsf{tryAbort}, \mathsf{tryCommit}\}$, each being a matching pair of *invocation* $Inv_{\mathsf{op_{ij}}}$ and *response* $Res_{\mathsf{op_{ij}}}$ actions. For a more compact representation of both the invocation and the response of an operation, we also use the notation $\mathsf{T_j.op_{ij}} \rightarrow \mathsf{val}$, where val is the value returned by its response action.

The specification of all possible operations of a transaction T_j is the following: $\mathsf{T_j.read(x)} \rightarrow \mathsf{val}$ is a read operation by T_j of an object x, which returns either a value in some domain V or a special value $\mathsf{aborted} \notin V$; $\mathsf{T_j.write(x,v)} \rightarrow \mathsf{val}$ is a write operation by T_j on an object x with a value $\mathsf{v} \in V$, which returns either *ok* or $\mathsf{aborted}$; $\mathsf{T_j.begin()} \rightarrow \mathsf{val}$ is the begin operation by T_j, which returns either *ok* or $\mathsf{aborted}$; $\mathsf{T_j.tryAbort()} \rightarrow \mathsf{val}$ is the request of an abort by T_j, which returns $\mathsf{aborted}$; $\mathsf{T}_j.\mathsf{tryCommit()} \rightarrow \mathsf{val}$ is the request of a commit by T_j, which returns either $\mathsf{committed} \notin V \cup \{\mathsf{aborted}\}$ or $\mathsf{aborted}$. We also use the following notation: $\mathsf{T_j.read(x)}$, to indicate $\mathsf{T_j.read(x)} \rightarrow \mathsf{val}$, with $\mathsf{val} \neq \mathsf{aborted}$; $\mathsf{T_j.write(x,v)}$, to indicate $\mathsf{T_j.write(x,v)} \rightarrow ok$; $\mathsf{T}_j.begin()$, to indicate $\mathsf{T}_j.\mathsf{begin()} \rightarrow \mathsf{ok}$; $\mathsf{T}_j.\mathsf{abort()}$, to indicate $\mathsf{T}_j.\mathsf{tryAbort()} \rightarrow \mathsf{aborted}$; and $\mathsf{T}_j.\mathsf{commit()}$, to indicate $\mathsf{T}_j.\mathsf{tryCommit()} \rightarrow \mathsf{committed}$.

Histories and implementations. A *history* $\mathcal{H}$ of a TM implementation is a (possibly infinite) sequence of invocation and response actions of operations. Let $\mathsf{tx}(\mathcal{H})$ denote the set of transactions in $\mathcal{H}$. A history $\mathcal{H}$ is *well-formed* if for all $\mathsf{T} \in \mathsf{tx}(\mathcal{H})$: *i)* T begins with an invocation of $\mathsf{begin()}$; *ii)* every invocation in T is followed by a matching response; *iii)* T has no actions after a response has returned either aborted or committed; and *iv)* T cannot invoke more than one begin operation. For simplicity, we assume that all histories are well-formed. Two histories $\mathcal{H}$ and $\mathcal{H}'$ are *equivalent* if $\mathsf{tx}(\mathcal{H}) = \mathsf{tx}(\mathcal{H}')$, and for every process p, $\mathcal{H}|p = \mathcal{H}'|p$, where $\mathcal{H}|p$ denote the projection of actions of process p in $\mathcal{H}$.

The *read-set* of a transaction T in history $\mathcal{H}$, denoted as $\mathsf{rset}(T)$, is the set of objects that T reads in $\mathcal{H}$; the *write-set* of T in history $\mathcal{H}$, denoted as $\mathsf{wset}(T)$, is the set of objects that T writes to in $\mathcal{H}$. T is an *update* transaction if $\mathsf{wset}(T) \neq \emptyset$; otherwise, T is a *read-only* transaction.

A *TM implementation* (TM for short), denoted by $\mathcal{T}$, provides processes with algorithms for implementing read, write, begin, tryCommit and tryAbort of a transaction. $\mathcal{T}$ is defined as a set of well-formed histories, which are the histories that are produced by $\mathcal{T}$. We denote by $\mathcal{H}_1 \cdot \mathcal{H}_2$, the concatenation of histories $\mathcal{H}_1$ and $\mathcal{H}_2$, $\mathsf{T}_1 \cdot \mathsf{T}_2$, the concatenation of transactions T_1 and T_2, and $op_{ij} \cdot op_{zk}$, the concatenation of operations op_{ij} and op_{zk}.

Complete histories and real-time precedence. A transaction $\mathsf{T} \in \mathsf{tx}(\mathcal{H})$ is *complete* if T ends with abort or committed in $\mathcal{H}$. A transaction T is *pending* in $\mathcal{H}$ if the last action of T is the invocation of tryCommit. A transaction T is *committed* (resp., *aborted*) in $\mathcal{H}$ if the return value of the last operation of T is committed (resp., aborted). A transaction T is *live* in $\mathcal{H}$ if it is neither pending nor completed. The history $\mathcal{H}$ is *complete* if all transactions in $\mathsf{tx}(\mathcal{H})$ are complete.

Given two transactions T_j and T_k, and two operations op_{ij} and op_{zk} in $\mathcal{H}$ by T_j and T_k, respectively, we say that op_{ij} *precedes* op_{zk} in the *real-time order* of $\mathcal{H}$, denoted $op_{ij} \prec_{\mathcal{H}} op_{zk}$, if the response action $Res_{op_{ij}}$ precedes the invocation action $Inv_{op_{zk}}$ in $\mathcal{H}$. Otherwise, if neither $op_{ij} \prec_{\mathcal{H}} op_{zk}$ nor $op_{zk} \prec_{\mathcal{H}} op_{ij}$, then op_{ij} and op_{zk} are *concurrent* in $\mathcal{H}$. We overload the notation and say, for transactions $\mathsf{T}_j, \mathsf{T}_k \in \mathsf{tx}(\mathcal{H})$, that T_j *precedes* T_k in the *real-time order* of $\mathcal{H}$, denoted $\mathsf{T}_j \prec_{\mathcal{H}} \mathsf{T}_k$, if T_j is complete in $\mathcal{H}$ and the last action of T_j precedes the first action of T_k in $\mathcal{H}$. If neither $\mathsf{T}_j \prec_{\mathcal{H}} \mathsf{T}_k$ nor $\mathsf{T}_k \prec_{\mathcal{H}} \mathsf{T}_j$, then T_j and T_k are *concurrent* in $\mathcal{H}$. A history $\mathcal{H}$ is *sequential* if there are no concurrent transactions in $\mathcal{H}$.

For simplicity, we assume that each history $\mathcal{H}$ begins with a complete transaction T_0 that writes initial values to all objects and commits before any other transaction begins in $\mathcal{H}$. A read operation $\mathsf{T.read(x)}$ in a complete and sequential history $\mathcal{H}$ is *legal* if it returns the latest value written by T, if T writes on x before $\mathsf{T.read(x)}$, otherwise it returns the latest value written by a committed transaction. A complete and sequential history $\mathcal{H}$ is *legal* if every read operation in $\mathcal{H}$, which does not return aborted, is legal.

Configuration, Step, and Execution. 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, a *step* ϕ_τ performed by a process encompasses a local computation, the application of a primitive operation (e.g., CAS) to a base object, and a possible change to the status of the process. Any step ϕ_τ is executed atomically, and it is generated by an operation op_{ij} of a transaction T_j, hence also denoting the step as $\phi_\tau^{op_{ij}}$. In general, an operation op_{ij} of a transaction can generate one or more steps $\phi_\tau^{op_{ij}}$, such that, for each op_{ij}, τ, one of the following conditions is true: $\phi_\tau^{op_{ij}}$ is the invocation $Inv_{op_{ij}}$; $\phi_\tau^{op_{ij}}$ is the response $Res_{op_{ij}}$; $\phi_\tau^{op_{ij}}$ is executed after $Inv_{op_{ij}}$ and before $Res_{op_{ij}}$.

An *execution interval* is a sequence $\Psi_\tau \cdot \phi_\tau \cdot \Psi_{\tau+1} \cdot \phi_{\tau+1} \cdot \ldots$ that alternates configurations Ψ_τ and steps ϕ_τ, where $\Psi_\tau \cdot \phi_\tau$ generates the configuration $\Psi_{\tau+1}$. A step ϕ_τ by process p_k is *legal* if its primitive operation applied to a base object follows the object's sequential specification [15]. Moreover, a configuration Ψ_τ is *quiescent* if, for any transaction T_i, Ψ_τ is not after the first operation of T_i and before its completion.

We define an *execution* E as an execution interval starting from an initial configuration Ψ_0. We also say that two executions are *indistinguishable* to a process p_k if *i)* p_k performs the same sequence of steps in both the executions, and *ii)* the steps by p_k are legal.

Since an execution E is actually a low-level history of configurations and steps 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, where $\forall i, j, k$, we remove all the configurations and the steps $\phi_k^{op_{ij}}$ such that $\phi_k^{op_{ij}} \neq Inv_{op_{ij}} \wedge \phi_k^{op_{ij}} \neq Res_{op_{ij}}$.

Invisible Transactions. Given an execution E and a set S of steps in E, we use the notation $E \setminus S$ to indicate the execution derived from E by removing only the steps in S and all the configurations generated by those steps. Also, given an execution E, and an operation op_{ij} by transaction T_j, we define $S_E^{op_{ij}}$ as the set $\{\phi_\tau^{op_{ij}} | \phi_\tau^{op_{ij}} \in E\}$. Given an execution E and a process p_k that executes a transaction T_j in E, we say that a set of operations Q of T_j is *invisible* in E if the executions E and $E \setminus (\bigcup_{op_{ij} \in Q} S_E^{op_{ij}})$ are indistinguishable to every process $p_{h \neq k}$ that takes steps in E. Therefore, we define a read-only transaction T_j as *invisible* if, for any possible execution E, the set of *all* the operations of T_j are invisible in E. Analogously, we say that a transaction has *invisible read operations* if, for any possible execution E, the set of the read operations of T_j are invisible in E. Moreover, we say that a transaction has an *invisible read execution* if it is a live transaction, and it has invisible read operations.

Consistency. Serializability [18] requires that for a history $\mathcal{H}$ to be serializable, there must exist a complete and sequential legal history S that is equivalent to all the committed transactions in $\mathcal{H}$, and strict-serializability requires S to preserve the real-time order in $\mathcal{H}$. Opacity [10, 11], informally, requires that for a history $\mathcal{H}$ to be opaque, there must exist a complete and sequential legal history S which is equivalent to a completion of $\mathcal{H}$, and preserves the real-time order in $\mathcal{H}$. *Transactional Memory Specification 2 (TMS2)* [6] is a stronger condition than opacity, and requires S to preserve the order of non-concurrent commit operations of committed update transactions. *Transactional Memory Specification 1 (TMS1)* [1, 6] is weaker than opacity, and requires $\mathcal{H}$ to be strictly serializable, and for each operation in $\mathcal{H}$, there must exist an equivalent legal history, and restricts this history to include all the committed transaction preceding the transaction of the operation in real-time order.

Progress guarantees. We say that a process p_i runs with no *step contention* in an execution interval α if α contains steps by p_i, and there are no other processes different from p_i that take steps in α. We also say that a transaction T_k (resp. op_{ik}) that is executed by a process p_i in an execution E does not encounter step contention in E if p_i runs with no *step contention* in the minimal execution interval α that contains all the steps of T_k (resp. op_{ij}) in E. A TM guarantees *obstruction-freedom* [13] if, for every execution E that is generated by the TM, a transaction T_i is forcefully aborted in E only if T_i encounters *step contention* in E. On the other hand, a TM guarantees *minimal progressiveness* [11] if, for every execution E that is generated by the TM, a transaction T_i is forcefully aborted in E only if T_i either encounters *step contention* in E or it does not start from a quiescent configuration. We assume that operations are *obstruction-free.*

3 The Cost of Reverse-Commit Anti-Dependency

We say that a TM implementation $\mathcal{T}$ allows RC-anti-dependency if the history shown in Figure 1 is a history of $\mathcal{T}$. More formally:

Definition 1. *Let* $\mathcal{H}_{rcad} = \mathcal{H}_\alpha \cdot \mathcal{H}_\beta \cdot \mathcal{H}_{\alpha'}$*, where* $\mathcal{H}_\alpha = \mathsf{T}_i.\mathsf{begin}() \cdot \mathsf{T}_i.\mathsf{read}(x) \rightarrow v_0$*,* $\mathcal{H}_\beta = \mathsf{T}_j.\mathsf{begin}() \cdot \mathsf{T}_j.\mathsf{read}(x) \rightarrow v_0 \cdot \mathsf{T}_j.\mathsf{write}(x, v_j) \cdot \mathsf{T}_j.\mathsf{commit}()$*, and* $\mathcal{H}_{\alpha'} = \mathsf{T}_i.\mathsf{write}(y, v_i) \cdot \mathsf{T}_i.\mathsf{commit}()$*, with* $i \neq j$ *and* $x \neq y$*. We say that a TM implementation* $\mathcal{T}$ *allows RC-anti-dependency iff* $\mathcal{H}_{rcad} \in \mathcal{T}$*.*

We use $\mathcal{H}_{rcad}$ to prove our impossibility results. Note that $\mathcal{H}_{rcad}$ is accepted by all known non-TMS2 implementations, e.g., TWM, as well as our TL2-RCAD algorithm. Intuitively, any TM implementation that accepts $\mathcal{H}_{rcad}$ is expected to include more histories that are not TMS2.

Our first result shows that allowing RC-anti-dependency in strictly serializable TMs implies an impossibility on having invisible read operations of update transactions.

Theorem 1. *A TM* $\mathcal{T}$ *that allows RC-anti-dependency cannot guarantee both strict serializability and minimal progressiveness if update transactions have invisible read operations.*

The intuition of the proof can be inferred from Figure 1. In order to allow RC-anti-dependency in Figure 1 and to guarantee that the history is strictly serializable, T_2 cannot read any object written by T_1. However, if read operations are invisible, it is impossible for T_1 to know whether such a read exists or not. We show that such a result holds even with a weak progress guarantee, like minimal progressiveness.

This result justifies the design of TM implementations like TWM [5], as well as our TL2-RCAD algorithm, which both allow RC-anti-dependency by making read operations of each update transaction visible at a certain point of the transaction execution. Indeed, in both implementations read operations of update transactions remain invisible until the execution of the transactions'

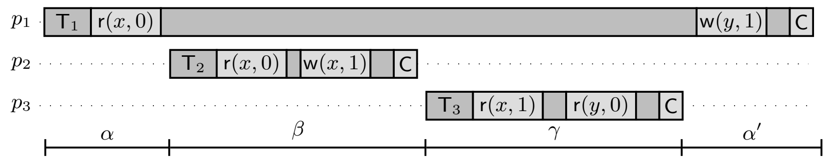


Fig. 2. For the transaction T_1, since read only transactions are invisible, this execution is indistinguishable to p_1 and p_2 from the execution in Figure 1.

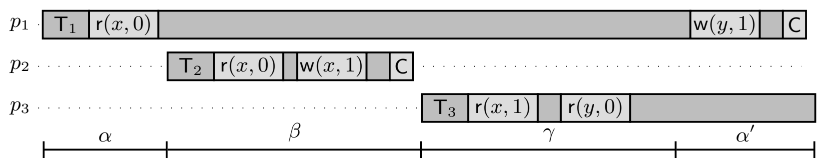


Fig. 3. For the transactions T_1 and T_2, since read operations are invisible, this execution is indistinguishable to p_1 and p_2 from the execution in Figure 1.

commit phase, and then read operations are forced to be visible during the commit phase. This means, based on our definitions, that both implementations have invisible read executions.

Having invisible read executions is weaker than having invisible read operations, because the former does not prevent a transaction from making its read operations visible after the invocation of either tryCommit or tryAbort. However, this relaxation for update transactions requires having visible read-only transactions, which is implied by our second impossibility result (Theorem 2).

Theorem 2. *A TM $\mathcal{T}$ that allows RC-anti-dependency cannot guarantee both strict serializability and minimal progressiveness if i) update transactions have invisible read executions, and ii) read-only transactions are invisible.*

The proof intuition is based on the indistinguishability of the two histories in Figures 1 and 2, due to the invisibility of the read-only transaction T_3. Specifically, if T_3 is invisible, T_1 has to behave in Figure 2 as in Figure 1. Also, the invisible read execution of T_1 gives both T_2 and T_3 the illusion of executing without concurrency, which means that they must commit due to minimal progressiveness. Therefore, the history in Figure 2, which is not strictly serializable, has to be accepted by $\mathcal{T}$.

Theorem 1 and Theorem 2 give two impossibility results mainly on update transactions. Therefore, a natural question would be: can we free the TM from any constraint on the invisibility of read operations of update transactions, and have non-TMS2 implementations that guarantee strict serializability? Theorem 3 shows that the answer is still "no", in case read-only transactions are invisible and obstruction-free.

Theorem 3. *A TM $\mathcal{T}$ that allows RC-anti-dependency cannot guarantee both strict serializability and minimal progressiveness if read-only transactions are invisible and obstruction-free.*

The intuition of the proof is also based on the indistinguishability of the two histories in Figures 1 and 2, due to the invisibility of the read-only transaction T_3. Specifically, if T_3 is invisible, both T_1 and T_2 must behave in Figure 2 as they do in Figure 1. In this case, although the read-only transaction T_3 does not have the illusion of running without concurrency, it has to commit because we assume that read-only transactions are obstruction-free, and T_3 runs without any step contention. Therefore, the history in Figure 2 has to be accepted by $\mathcal{T}$.

All the previous theorems assume strictly serializable TM implementations. Theorem 4, on the other hand, shows how the impossibility results will change if we rather aim for opacity than strict serializability. Specifically, we show that in order to have TM implementations that guarantee opacity and allow RC-anti-dependency, any read operation of any transaction (whether it is read-only or not) must be visible at the time the operation is executed. Our investigation on the relation between guaranteeing opacity and allowing RC-anti-dependency is motivated by some TM implementations that allow RC-anti-dependency and violate opacity, such as TWM and TL2-RCAD.

Theorem 4. *A TM $\mathcal{T}$ that allows RC-anti-dependency cannot guarantee both opacity and minimal progressiveness if transactions have invisible read executions.*

The intuition of the proof is based on the indistinguishability of the two histories in Figures 1 and 3, due to the invisibility of the read executions. Specifically, based on the definition of invisible read executions, T_3 is invisible since it is live and it did not execute any write operation yet. Thus, both T_1 and T_2 must behave in Figure 3 as they do in Figure 1. Furthermore, the invisible read execution of T_1 gives both T_2 and T_3 the illusion of executing without concurrency, which means that they cannot abort due to minimal progressiveness. As a result, the history in Figure 3 has to be accepted by $\mathcal{T}$, which violates opacity. Note that T_3 can abort later, in order to preserve strict serializability after T_1 commits. However, aborting T_3 does not make the history opaque.

4 TL2-RCAD: a TM implementation that allows RC-anti-dependency

In the previous section, we showed a set of impossibility results on allowing RC-anti-dependency in a TM implementation. In this section, we show a possibility result: a modified version of TL2 [4], named TL2-RCAD, that allows RC-anti-dependency and therefore deploys visible read operations. Algorithm 1 shows the main procedures of TL2-RCAD. The entire pseudo code is included in Appendix A.

TL2 uses a shared timestamp, which is atomically incremented anytime an update transaction commits and locally copied into the *start timestamp* at the beginning of a transaction execution. This timestamp is used by read operations to decide if the version available of a shared object is compliant with the transaction's history. At commit time, update transactions undergo a two-phase locking on written locations, and modifications are applied to the shared state only if all versions of read locations are still valid. Versions of locations are stored along with locks in a shared ownership record (*orec*) table.

One of the main issues in TL2 is that the live validation (i.e., the one made before returning from a read operation) is conservative: a transaction T_i aborts if the version of the *orec* to be read is greater than T_i's start timestamp. Using such a scheme to build TL2-RCAD would reduce the chance for allowing RC-anti-dependency because only few transactions with a potential RC-anti-dependency would reach the commit phase. Therefore we use a variant of TL2 that extends T_i's starting timestamp by using the technique presented in [19] (lines 17-21).

To the best of our knowledge, all TM algorithms that allow RC-anti-dependency are multi-versioning. Multi-versioning has its own practical limitations (e.g., expensive memory management) that makes it not a candidate in some real applications. TL2-RCAD is the first practical single-version TM algorithm that allows RC-anti-dependency. In Section 6, we discuss the differences between TL2-RCAD and TWM [5], a multi-versioning algorithm that allows RC-anti-dependency.

Algorithm 1 TL2-RCAD

```
1:  procedure Start()
2:      tx.start = global_timestamp
3:  end procedure
4:  procedure Read(addr)
5:      val = tx.write-set.find(addr)
6:      if val != NULL then
7:          return val
8:      orec = getOrec(addr)
9:      while true do
10:         val = *addr
11:         o = orec
12:         if o.lock then
13:             continue
14:         if o.wv ≤ tx.start then
15:             tx.readset.append(orec)
16:             return val
17:         new_start = global_timestamp
18:         for each (orec) in tx.readset do
19:             if orec.wv > tx.start then
20:                 Abort()
21:         tx.start = new_start
22: end procedure
23: procedure Write(addr, value)
24:     tx.writeset.add(addr, value)
25: end procedure

26: procedure CommitRW()
27:     LockAbortIfLockedNotByMe(tx.writeset)
28:     for each (r_orec) in tx.readset do
29:         if LockedNotByMe(r_orec) then
30:             Abort()
31:     for each (r_orec) in tx.readset do
32:         if r_orec.wv > tx.start then
33:             CheckAntiDep()
34:             break
35:     WriteBack(tx.writeset)
36:     tx.end = AtomicInc(global_timestamp)
37:     UpdateVersions()
38:     Unlock(tx.writeset)
39: end procedure

40: procedure CommitRO
41:     AntiDepHandle()
42: end procedure
```

TL2-RCAD Metadata. Based on our impossibility results, a mandatory step to allow RC-anti-dependency and guarantee at least strict serializability is to expose more metadata to make read-only transactions and the read operations of update transactions visible. TL2-RCAD exposes per-*orec* metadata for update transactions and a global flag for read-only transactions.

More in details, each *orec* is enriched with a read version, named *orec.rv*, that is modified at the commit time of only update transactions (hereafter we name TL2's original *orec* versions as write version, or *orec.wv*). As we will show later, adding the read version is enough to detect simple scenarios where there is no read-only transactions and there is only one transaction that allows RC-anti-dependency at a time, like the example in Figure 1. We also define three shared global metadata: *anti_dep_lock*, *last_ro*, and *last_anti_dep*. Those metadata are used to detect the more complicated scenarios, like the one in Figure 2, where at least two concurrent transactions attempt to commit and they both allow RC-anti-dependency, or one of them allows RC-anti-dependency and the other is read-only. Note that *last_ro*, and *last_anti_dep*, and the read versions of the *orecs* should be monotonically increasing, which is not guaranteed if transactions overwrite their values without checking the old values. Hereafter we use the term *monotonic update* to refer to the correct action that considers this requirement. The detailed pseudo code of this monotonic update is in Appendix A.

TL2-RCAD Commit Procedure. Now we show how we modify the commit procedure of TL2 to allow RC-anti-dependency. We structured the pseudocode in Algorithm 1 so that the difference between TL2 and TL2-RCAD, in addition to the new metadata, lies only in the three functions at lines 33, 37, and 41. For each function, we show how TL2 implements it, and how TL2-RCAD extends that. The detailed pseudocode for both TL2 and TL2-RCAD is in Appendix A.

The first function is *CheckAntiDep*. TL2-RCAD allows RC-anti-dependency at commit time by enriching TL2's validation procedure. In TL2 each transaction T_i iterates over all the *orecs* of its read-set and checks if the write version of each *orec* is higher than the start timestamp of T_i, which is the condition for RC-anti-dependency. At this point, TL2 conservatively aborts T_i, while TL2-RCAD tries to commit T_i if allowing RC-anti-dependency does not result in executions that violate strict serializability. To do so, the *CheckAntiDep* function in TL2-RCAD is implemented as follows: it first acquires the *anti_dep_lock* to guarantee that no other transaction will concurrently allow RC-anti-dependency. Then, it checks if T_i's start timestamp is less than: *i) last_ro*, *ii) last_anti_dep*, and *iii)* both the read versions and the write versions of all T_i's write-set entries. Interestingly, all those steps use only information saved in T_i's read-set and do not require any accurate knowledge about dependencies of other transactions.

The second function is *UpdateVersions*. In this function, TL2 only modifies the write version of the write-set entries. In addition to that, TL2-RCAD modifies the read version of the read-set entries. Also, if committing the transaction generates RC-anti-dependency (i.e., it calls *CheckAntiDep*), *last_anti_dep* is monotonically updated to be the new value of the shared timestamp (after being atomically incremented in line 36), then *anti_dep_lock* is released. The third function is *AntiDepHandle*, which is an empty function in TL2. In TL2-RCAD a read-only transaction T_r monotonically updates *last_ro* to be the current value of the shared timestamp. Then, it checks if *anti_dep_lock* is acquired or *last_anti_dep* is greater than or equal to T_r's start timestamp, which indicates a concurrent transaction that allowed or is trying to allow RC-anti-dependency. In that case, T_r conservatively aborts.

Considering Theorem 4, TL2-RCAD violates opacity because it makes read operations visible only during the commit phase. However, as we prove in Theorem 5 (proof is in Appendix B), TL2-RCAD guarantees strict serializability, and this does not contradict the other impossibility results presented in Section 3, because read operations and read-only transactions are visible in TL2-RCAD. In fact, Theorem 5 proves that TL2-RCAD guarantees TMS1 [6], which is stronger than strict serializability. Interestingly, to the best of our knowledge, TL2-RCAD is the first TM implementation that has TMS1 guarantee.

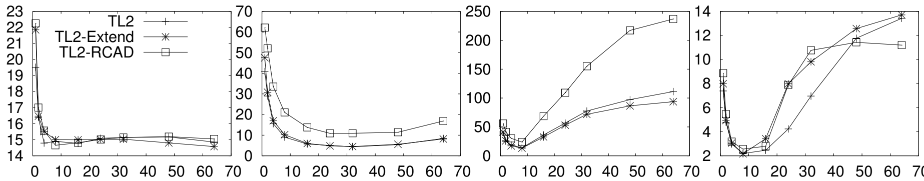
Theorem 5. *TL2-RCAD guarantees TMS1.*

5 Evaluation

In this section, we evaluate TL2-RCAD, mainly to understand how allowing RC-anti-dependency and weakening TMS2 affects performance. To do so, we compare three variants of the TL2 algorithm: the TL2 implementation (TL2), the TL2 implementation with the extension of the transaction start timestamp (TL2-Extend), and TL2-RCAD. We evaluated the algorithms using the STAMP benchmark suite [17]. The testbed consists of an AMD server equipped with 64 CPU-cores. Each datapoint is the average of 5 runs. We use two metrics during the evaluation: throughput (in Figure 4) and the commit/abort ratio normalized to the total number of executions of TL2 (in Figure 5). Due to space constraints, we show the most significant plots here and we refer to Appendix for the others.

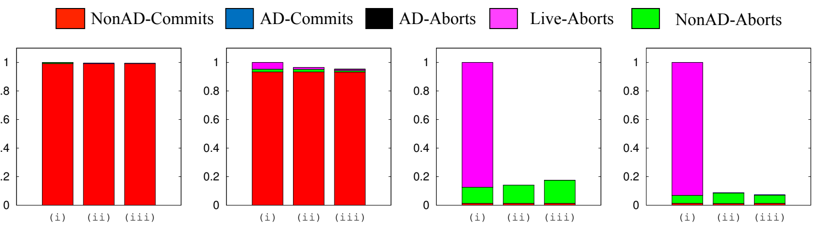
Our main observation is that aborts occur because of two main reasons other than for RC-anti-dependency: finding an inconsistent state by a live transaction, and failing in acquiring locks at commit time. That is why, the results in Figure 4 show only a marginal performance improvement in one application (`kmeans`), and a performance similar to or worse than the other versions of TL2 in all other applications. Roughly, the results split STAMP benchmarks into four categories based on the level of contention and the size of transactions read-sets. This is because contention level is an indicator of the potential gain of allowing RC-anti-dependency; and the size of the read-set indicates the overhead of allowing RC-anti-dependency in TL2-RCAD given the need of updating the read versions.

The first category includes `ssca2` and `labyrinth`. In `ssca2`, transactions are non-conflicting and have small read-sets, while in `labyrinth` transactions have dominating non-transactional work. That is why in both scenarios allowing RC-anti-dependency causes neither a degradation nor an improvement in performance. Figures 4(a) confirms that by showing similar performance for all competitors. The second category (`vacation` and `genome`) represents workloads with non-conflicting transactions and large read-sets. In this case, the overhead of allowing RC-anti-dependency increases due to large read-sets, but it does not provide any benefit because most of the transactions already commit even using the original TL2. That is why the performance of TL2-RCAD is constantly worse than other competitors, as shown in Figure 4(b), which reflects the overhead of allowing RC-anti-dependency. The third category is represented by `intruder`, which is the worst case for TL2-RCAD. Performance degradation is higher than all the other benchmarks, as shown in Figure 4(c). This is mainly because transactions have large read-sets, which adds a significant overhead that increases the overall commit time. `Kmeans` (Figure 4(d)) represents the forth category, where transactions have small read-sets but they are more conflicting than the



(a) Ssca2 (b) Vacation (c) Intruder (d) Kmeans

Fig. 4. Performance using STAMP. X-axis: number of threads; Y-axis: Time (s).



(a) Ssca2 (b) Vacation (c) Intruder (d) Kmeans

Fig. 5. Commit/abort ratios in STAMP. *(i)* TL2; *(ii)* TL2-Extended; *(iii)* TL2-RCAD.

previous applications. Due to the small read-sets, TL2-RCAD does not generally perform worse than its competitors. However, we also observe no gain from allowing RC-anti-dependency, as detailed below.

Figure 5 shows, for each version of TL2, the percentage of committed read-only and update transactions that never called *CheckAntiDep* (*NonAD-Commits*); committed update transactions that called *CheckAntiDep* (*AD-Commits*); aborted transactions that called either *CheckAntiDep* or *AntiDepHandle* (*AD-Aborts*); aborted live transactions (*Live-Aborts*); and aborted update transactions due to failing in locking an *orec* at commit (*NonAD-Aborts*). The sum of *AD-Commits* and *AD-Aborts* represents all the potential executions for RC-anti-dependency, while *Live-Aborts* and *NonAD-Aborts* represent the two reasons for aborting transactions other than observing RC-anti-dependency. Unfortunately, although we can save some *Live-Aborts* by spinning until the location is unlocked, which is the case of both TL2-Extend and TL2-RCAD (line 13), spinning at commit time in order to save some *NonAD-Aborts* can result in deadlock.

Figure 5 assesses that the overall percentage of potential RC-anti-dependency is small, which clarifies the reason of limited performance improvement (or degradation). More in details, Figures 5(a) and 5(b) confirm our analysis in `ssca2` and `vacation` by showing that *NonAD-Commits* is dominating. `Labyrinth` cannot be interpreted due to its dominating non-transactional work. In `intruder` (Figure 5(c)), *NonAD-Aborts* of TL2-RCAD are higher than TL2-Extend. This is a direct implication of holding locks for a longer time at commit time, due to the longer validation process and the time spent in writing the read versions. Finally, in `kmeans` (Figure 5(d)) the number of RC-anti-dependency observed is very limited, which results in a slight performance improvement only for the cases of 48/64 threads. Note that, although both TL2-Extend and TL2-RCAD save most of live aborts in both `kmeans` and `intruder`, the impact of that in performance is not reflected. This is mainly because preventing aborts in those cases come with an additional cost of incrementally validating the whole read-set.

We also ran micro-benchmarks (shown in Appendix) seeking for a favorable configuration that allows RC-anti-dependency. Results confirmed a limited performance improvement. As a summary, although we cannot claim that more favorable workloads do not exist, our evaluation study assesses that, even with a hand-tuned micro-benchmark, it is hard to find workloads where allowing RC-anti-dependency in a single-version TM enhances performance noticeably.

6 Related Work

We classify the previous works that allow RC-anti-dependency into two categories: permissive algorithms, and multi-versioning algorithms. Interestingly, both of them confirm our impossibility results by adopting techniques that make both read-only transactions and the reads of update transactions visible.

Permissive algorithms need to track all dependencies in the system [9, 16], and/or acquire locks for read operations [2], and both these techniques are known to have a significant negative impact on performance. That is why, unlike TL2-RCAD, all those solutions in this regard aimed at proving theoretical possibility results rather than assessing practical implications.

Multi-versioning algorithms have a major benefit: allowing read-only transactions to progress (usually non-blocking), and, generally, read operations to complete without aborting the enclosing transaction. That is easy to achieve because, thanks to the multi-versioned memory, transactions can always find a consistent version to read. That is why even multi-versioning algorithms that do not allow RC-anti-dependency, such as LSA [19] or JVSTM [7], have this positive effect. We believe that the advantages of allowing RC-anti-dependency have a limited gain compared to the potential gain of having non-blocking read-only transactions. We justify this claim by briefly analyzing TWM [5], an algorithm that uses multi-versioning and allows RC-anti-dependency. In the evaluation of TWM, all the workloads that show a significant improvement have long and mostly read-only transactions. Here TWM mainly benefits from the strong progress of read operations. Those scenarios are not favorable for TL2-RCAD because, without multi-versioning, it is hard to improve the progress of read operations, and also those workloads add significant overhead due to their long read-sets.

Theoretically, our results are inspired by a research trend that aims at identifying the cost of accepting more histories in TM. For example, both *online permissiveness* [16] and *input-acceptance* [8] introduce similar impossibility results on the visibility of read and write operations when the TM accepts some sets of histories. Our results, however, are stronger since they are based on the assumption that TM accepts only one history. Also, we selected that history as it is the one used for identifying TMS2, which allows, for the first time, understanding the cost and limitations of relaxing TMS2 while still being safe.

7 Conclusion

In this paper we investigated the inherent costs and limitations of allowing RC-anti-dependency in TM implementations. The major outcome of our findings is that, the mandatory costs of allowing RC-anti-dependency (e.g., having visible read operations) is not reflected in noticeable performance improvement.

Acknowledgements. This work is partially supported by Air Force Office of Scientific Research (AFOSR) under grant FA9550-14-1-0187.

References

1. Attiya, H., Gotsman, A., Hans, S., Rinetzky, N.: Safety of live Transactions in Transactional Memory: TMS is necessary and sufficient. In: DISC. pp. 376–390 (2014)
2. Attiya, H., Hillel, E.: A single-version STM that is multi-versioned permissive. Theory Comput. Syst. 51(4), 425–446 (2012)
3. Dalessandro, L., Spear, M.F., Scott, M.L.: NOrec: streamlining STM by abolishing ownership records. In: PPOPP. pp. 67–78 (2010)
4. Dice, D., Shalev, O., Shavit, N.: Transactional Locking II. In: DISC. pp. 194–208 (2006)
5. Diegues, N., Romano, P.: Time-warp: Lightweight abort minimization in transactional memory. In: PPoPP. pp. 167–178 (2014)
6. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Towards formally specifying and verifying Transactional Memory. Formal Aspects of Computing 25(5), 769–799 (2013)
7. Fernandes, S.M., Cachopo, J.P.: Lock-free and scalable multi-version software transactional memory. In: PPOPP. pp. 179–188 (2011)
8. Gramoli, V., Harmanci, D., Felber, P.: On the Input Acceptance of Transactional Memory. Parallel Processing Letters 20(1), 31–50 (2010)
9. Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in Transactional memories. In: DISC. pp. 305–319 (2008)
10. Guerraoui, R., Kapalka, M.: On the correctness of Transactional Memory. In: PPOPP. pp. 175–184 (2008)
11. Guerraoui, R., Kapalka, M.: Principles of Transactional Memory. Synthesis Lectures on Distributed Computing Theory, Morgan and Claypool (2011)
12. Hans, S., Hassan, A., Palmieri, R., Peluso, S., Ravindran, B.: Opacity vs TMS2: Expectations and Reality. Tech. rep., Virginia Tech (2016), http://www.ssrg.ece.vt.edu/papers/disc16-TR.pdf
13. Herlihy, M., Luchangco, V., Moir, M., III, W.N.S.: Software Transactional Memory for dynamic-sized data structures. In: PODC. pp. 92–101 (2003)
14. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: ISCA. pp. 289–300 (1993)
15. Herlihy, M., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM Transactions on Programming Languages and Systems 12(3), 463–492 (1990)
16. Keidar, I., Perelman, D.: On Avoiding Spare Aborts in Transactional Memory. In: SPAA. pp. 59–68 (2009)

17. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IISWC. pp. 35–46 (2008)
18. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26, 631–653 (October 1979)
19. Riegel, T., Felber, P., Fetzer, C.: A Lazy Snapshot Algorithm with Eager Validation. In: DISC. pp. 284–298 (2006)

A Detailed Pseudocode of TL2-RCAD

Algorithm 2 Detailed TL2-RCAD

```
procedure Start()
    tx.start = global_timestamp
end procedure

procedure Read(addr)
    val = tx.write-set.find(addr)
    if val != NULL then
        return val
    orec = getOrec(addr)
    while true do
        val = *addr
        o = orec
        if o.lock then
            continue
        if o.wv ≤ tx.start then
            tx.readset.append(orec)
            return val
        new_start = global_timestamp
        for each (orec) in tx.readset do
            if orec.wv > tx.start then
                Abort()
        tx.start = new_start
end procedure

procedure Write(addr, value)
    tx.writeset.add(addr, value)
end procedure

procedure CommitRW()
    LockAbortIfLockedNotByMe(tx.writeset)
    for each (r_orec) in tx.readset do
        if LockedNotByMe(r_orec) then
            Abort()
    for each (r_orec) in tx.readset do
        if r_orec.wv > tx.start then
            CheckAntiDep()
            break
    WriteBack(tx.writeset)
    tx.end = AtomicInc(global_timestamp)
    UpdateVersions()
    Unlock(tx.writeset)
end procedure

procedure CommitRO
    AntiDepHandle()
end procedure

procedure CheckAntiDep
    tx.anti_dep_flag = true
    anti_dep_lock.acquire()
    if last_anti_dep ≥ tx.start or last_ro ≥ tx.start then
        anti_dep_lock.release()
        Abort()
    for each (w_orec) in tx.writeset do
        if w_orec.wv geq tx.start or w_orec.rv geq tx.start then
            anti_dep_lock.release()
            Abort()
end procedure

procedure UpdateVersions
    for each (r_orec) in tx.readset do
        MonotonicUpdate(r_orec.rv, tx.)
    for each (w_orec) in tx.writeset do
        w_orec.wv = tx.end
    if tx.anti_dep_flag then
        last_anti_dep = tx.end_time
        anti_dep_lock.release()
end procedure

procedure AntiDepHandle
    MonotonicUpdate(last_ro, global_timestamp)
    if anti_dep_lock.locked or last_anti_dep ≥ tx.start then
        Abort()
end procedure

procedure MonotonicUpdate(var, new_val)
    repeat
        cur_val = var
        if cur_val ≥ new_val then
            break
    until CAS(var, cur_val, new_val) = true
end procedure
```

B Proofs

B.1 Proof of Theorem 1

Proof. Let p_1, p_2 be two processes, and x and y be two distinct data items with initial value equal to v_0. Let us also consider the following execution intervals:

- α: p_1 runs with no step contention from the initial configuration, it starts an update transaction T_1, and it executes the read operation $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$.

- β: p_2 runs with no step contention, it performs an update transaction T_2, by executing the read operation $\mathsf{T}_2.\mathsf{read}(\mathsf{y})$, the read operation $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$, and the write operation $\mathsf{T}_2.\mathsf{write}(\mathsf{x}, \mathsf{v}_2)$, and then it commits T_2.
- α': p_1 runs with no step contention, it executes the write operation $\mathsf{T}_1.\mathsf{write}(\mathsf{y}, \mathsf{v}_1)$, and it commits T_1.
- β': p_2 runs with no step contention, it performs transaction T_2, by executing the read operation $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$, the write operation $\mathsf{T}_2.\mathsf{write}(\mathsf{x}, \mathsf{v}_2)$, and then it commits T_2.

We first prove that the executions $E_1 = \alpha \cdot \beta$ and $E^\beta = \beta$ are valid executions of $\mathcal{T}$, and that $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$, $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$, and $\mathsf{T}_2.\mathsf{read}(\mathsf{y})$ return v_0 as follows. α is valid in E_1 because transactions are minimal progressive, and, in α, T_1 runs without step contention from the initial quiescent configuration. For the same reason, β is valid in E^β because T_2 runs without step contention from the initial quiescent configuration. Also, since read operations of update transactions are invisible, the executions E_1 and E^β are indistinguishable to process p_2 by definition of invisible read operations, meaning that p_2 has to take the same steps both in E_1 and E^β. Therefore, β is valid in E_1 as well. In addition, v_0 is the value of both x and y at the time α and the read operations of β are performed, and therefore $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$, $\mathsf{T}_2.\mathsf{read}(\mathsf{y})$, and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ return v_0.

At this point we prove that the valid execution E_1 can be extended by obtaining an execution that must be accepted by $\mathcal{T}$, although it violates strict serializability. In particular, let us consider the executions $E_2 = \alpha \cdot \beta \cdot \alpha'$ and $E_3 = \alpha \cdot \beta' \cdot \alpha'$. First, we notice that E_3 must be accepted by $\mathcal{T}$, since $\mathcal{T}$ has to accept RC-anti-dependencyhistories by hypothesis, and $E_3|\mathcal{H}$ is RC-anti-dependency. Then the prefix $\alpha \cdot \beta'$ of E_3 is a valid execution of $\mathcal{T}$, for the same reason why E_1 is a valid execution of $\mathcal{T}$.

However, when p_1 runs in E_2 after $\alpha \cdot \beta$ has been executed, it has to behave like in E_3, by executing α'. This is because T_2 is an update transaction in β and β', and E_2 and E_3 are indistinguishable to p_1 by definition of invisible read operations. Therefore, E_2 must be a valid execution that is accepted by $\mathcal{T}$, although the history $E_2|\mathcal{H}$ is not strict serializable. Indeed, in any possible legal sequential history of T_1, T_2, we have that one of the following statements must be true: *i)* $\mathsf{T}_2.\mathsf{read}(\mathsf{y})$ by T_2 has to return v_1; *ii)* $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ by T_1 has to return v_2. This proves the theorem.

B.2 Proof of Theorem 2

Proof. Let p_1, p_2 and p_3 be three processes, and x and y be two distinct data items with initial value equal to v_0. Let us also consider the following execution intervals:
- α: p_1 runs with no step contention from the initial configuration, it starts transaction T_1, and it executes the read operation $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ of T_1.
- β: p_2 runs with no step contention, it performs transaction T_2, by executing the read operation $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$, the write operation $\mathsf{T}_2.\mathsf{write}(\mathsf{x}, \mathsf{v}_2)$, and then it commits T_2.

- α': p_1 runs with no step contention, it executes the write operation $\mathsf{T}_1.\mathsf{write}(\mathsf{y}, \mathsf{v}_1)$ of T_1, and it commits T_1.
- γ: p_3 runs with no step contention, and it executes and commits the read-only transaction T_3, with the read operation $\mathsf{T}_3.\mathsf{read}(\mathsf{x})$ and the read operation $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$.

We first prove that the executions $E_1 = \alpha \cdot \beta$ and $E^{\beta} = \beta$ are valid executions of $\mathcal{T}$, and that $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ return v_0 in E_1 and E_2 as follows. α is valid in E_1 because transactions are minimal progressive, and, in α, T_1 runs without step contention from the initial quiescent configuration. For the same reason, β is valid in E^{β} because T_2 runs without step contention from the initial quiescent configuration. Moreover, T_1 is invisible in E_1, since T_1 is live in E_1. If so, the executions E_1 and E^{β} are indistinguishable to process p_2 by the definition of invisible read executions, meaning that p_2 has to take the same steps both in E_1 and E^{β}. Therefore, β is valid in E_1 as well. In addition, v_0 is the value of x at the time α and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ are performed, and therefore $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ return v_0.

Now we are going to extend the execution E_1 with the execution interval γ, and we are going to show that the resulting execution is valid. In particular, let us consider the executions $E_2 = \alpha \cdot \beta \cdot \gamma$ and $E_3 = \beta \cdot \gamma$. E_3 is a valid execution of $\mathcal{T}$ since both T_2 and T_3 (in β and γ, respectively) run without step contention from a quiescent state, and they cannot abort according to minimal progressiveness. Also, the execution E_2 is a valid execution of $\mathcal{T}$ since $E_1 = \alpha \cdot \beta$ is valid, and the executions $E_2 = \alpha \cdot \beta \cdot \gamma$ and $E_3 = \beta \cdot \gamma$ are indistinguishable to process p_3, due to the invisibility of transaction T_1 (as it is in E_1). Indeed, after the execution interval $\alpha \cdot \beta$, p_3 has to behave in E_2 like it does in E_3, since T_1 is invisible.

It is also straightforward showing that, $\mathsf{T}_3.\mathsf{read}(\mathsf{x})$ has to return v_0 in E_2 and E_3, for the same reason why $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ returns v_0 in E_1. Also, since T_3 follows T_2 according to the real-time time that is induced by E_2 and E_3, and $\mathcal{T}$ guarantees strict-serializability, then $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$ has to return v_2 in both E_2 and E_3.

At this point we prove that the valid execution E_2 can be extended by obtaining an execution that must be accepted by $\mathcal{T}$, although it violates strict serializability. In particular, let us consider the executions $E_4 = \alpha \cdot \beta \cdot \gamma \cdot \alpha'$ and $E_5 = \alpha \cdot \beta \cdot \alpha'$. First, we notice that E_5 must be an execution of $\mathcal{T}$, since $\mathcal{T}$ has to accept RC-anti-dependencyhistories by hypothesis, and $E_5|\mathcal{H}$ is RC-anti-dependency. Then the prefix $\alpha \cdot \beta \cdot \gamma$ of E_5 is a valid execution of $\mathcal{T}$, since it is equal to E_2. However, when p_1 runs in E_4 after $\alpha \cdot \beta \cdot \gamma$ has been executed, it has to behave like in E_5, by executing α'. This is because T_3 is a read-only transaction in γ, and E_4 and E_5 are indistinguishable to p_1. Therefore, E_4 must be a valid execution of $\mathcal{T}$, although the history $E_4|\mathcal{H}$ is not strict serializable. Indeed, in any possible legal sequential history of T_1, T_2, and T_3, which does not violate the real-time order of T_2 and T_3, we have that one of the following statements must be true: *i)* $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$ by T_3 has to return v_1; *ii)* $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ by T_1 has to return v_2. This proves the theorem.

B.3 Proof of Theorem 3

Proof. Let p_1, p_2 and p_3 be three processes, and x and y be two distinct data items with initial value equal to v_0. Let us also consider the following execution intervals:

- α: p_1 runs with no step contention from the initial configuration, it starts transaction T_1, and it executes the read operation $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ of T_1.
- β: p_2 runs with no step contention, it performs transaction T_2, by executing the read operation $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$, the write operation $\mathsf{T}_2.\mathsf{write}(\mathsf{x}, \mathsf{v}_2)$, and then it commits T_2.
- α': p_1 runs with no step contention, it executes the write operation $\mathsf{T}_1.\mathsf{write}(\mathsf{y}, \mathsf{v}_1)$ of T_1, and it commits T_1.
- γ: p_3 runs with no step contention, and it executes and commits the read-only transaction T_3, with the read operation $\mathsf{T}_3.\mathsf{read}(\mathsf{x})$ and the read operation $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$.

Let us also consider the executions $E_1 = \alpha \cdot \beta \cdot \gamma \cdot \alpha'$ and $E_2 = \alpha \cdot \beta \cdot \alpha'$.

First, we notice that E_2 must be an execution of $\mathcal{T}$, since $\mathcal{T}$ has to accept RC-anti-dependencyhistories by hypothesis, and $E_2|\mathcal{H}$ is RC-anti-dependency. Therefore the prefix $\alpha \cdot \beta$ in both E_1 and E_2 is a valid execution of $\mathcal{T}$. Furthermore, the prefix $\alpha \cdot \beta \cdot \gamma$ of E_1 is a valid execution of $\mathcal{T}$, since read-only transactions are obstruction-free, and T_3 is a read-only transaction executing with no step contention in γ.

About the return values of read operations, we notice that v_0 is the value of x at the time α and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ are performed, and therefore $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ return v_0 in both E_1 and E_2. For the same reason, $\mathsf{T}_3.\mathsf{read}(\mathsf{x})$ has to return v_0 in E_1. Also, since T_3 follows T_2 according to the real-time time that is induced by E_1, and $\mathcal{T}$ guarantees strict serializability, then $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$ has to return v_2 in E_1.

However, when p_1 runs in E_1 after $\alpha \cdot \beta \cdot \gamma$ has been executed, it has to behave like in E_2, by executing α'. This is because T_3 is a read-only transaction in γ, and E_1 and E_2 are indistinguishable to p_1 due to the invisibility of read-only transactions. Therefore, E_1 must be a valid execution of $\mathcal{T}$, although the history $E_1|\mathcal{H}$ is not strict serializable. Indeed, in any possible legal sequential history of T_1, T_2, and T_3, which does not violate the real-time order of T_2 and T_3, we have that one of the following statements must be true: *i)* $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$ by T_3 has to return v_1; *ii)* $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ by T_1 has to return v_2. This proves the theorem.

B.4 Proof of Theorem 4

Proof. Let p_1, p_2 and p_3 be three processes, and x and y be two distinct data items with initial value equal to v_0. Let us also consider the following execution intervals:

- α: p_1 runs with no step contention from the initial configuration, it starts transaction T_1, and it executes the read operation $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ of T_1.
- β: p_2 runs with no step contention, it performs transaction T_2, by executing the read operation $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$, the write operation $\mathsf{T}_2.\mathsf{write}(\mathsf{x}, \mathsf{v}_2)$, and then it commits T_2.

- α': p_1 runs with no step contention, it executes the write operation $\mathsf{T}_1.\mathsf{write}(\mathsf{y}, \mathsf{v}_1)$ of T_1, and it commits T_1.
- γ: p_3 runs with no step contention, it starts a transaction T_3, and it executes two read operations $\mathsf{T}_3.\mathsf{read}(\mathsf{x})$ and $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$.

We first prove that the executions $E_1 = \alpha \cdot \beta$ and $E^\beta = \beta$ are valid executions of $\mathcal{T}$, and that $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ return v_0 in E_1 and E_2 as follows. α is valid in E_1 because transactions are minimal progressive, and, in α, T_1 runs without step contention from the initial quiescent configuration. For the same reason, β is valid in E^β because T_2 runs without step contention from the initial quiescent configuration. Furthermore, since T_1 is live in α, and it does not execute any write operation, it is invisible in E_1 due to the invisible read execution assumption.

Therefore, the executions E_1 and E^β are indistinguishable to process p_2, meaning that p_2 has to take the same steps both in E_1 and E^β, and hence β is valid in E_1 as well. In addition, v_0 is the value of x at the time α and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ are performed, and therefore $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ and $\mathsf{T}_2.\mathsf{read}(\mathsf{x})$ return v_0.

Now we are going to extend the execution E_1 with the execution interval γ, and we are going to show that the resulting execution is valid. In particular, let us consider the executions $E_2 = \alpha \cdot \beta \cdot \gamma$ and $E_3 = \beta \cdot \gamma$. E_3 is a valid execution of $\mathcal{T}$ since both T_2 and T_3 (in β and γ, respectively) run without step contention from a quiescent state, and they cannot abort according to minimal progressiveness. Also, the execution E_2 is a valid execution of $\mathcal{T}$ since $E_1 = \alpha \cdot \beta$ is valid, and the executions $E_2 = \alpha \cdot \beta \cdot \gamma$ and $E_3 = \beta \cdot \gamma$ are indistinguishable to process p_3, due to the invisibility of transaction T_1 (as it is in E_1). Indeed, after the execution interval $\alpha \cdot \beta$, p_3 has to behave in E_2 like it does in E_3, since T_1 is invisible.

It is also straightforward showing that, $\mathsf{T}_3.\mathsf{read}(\mathsf{x})$ has to return v_0 in E_2 and E_3, for the same reason why $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ returns v_0 in E_1. Also, since T_3 follows T_2 according to the real-time time that is induced by E_2 and E_3, and $\mathcal{T}$ guarantees opacity, then $\mathsf{T}_3.\mathsf{read}(\mathsf{y})$ has to return v_2 in both E_2 and E_3.

At this point we prove that the valid execution E_2 can be extended by obtaining an execution that must be accepted by $\mathcal{T}$, although it violates opacity. In particular, let us consider the executions $E_4 = \alpha \cdot \beta \cdot \gamma \cdot \alpha'$ and $E_5 = \alpha \cdot \beta \cdot \alpha'$. First, we notice that E_5 must be an execution of $\mathcal{T}$, since $\mathcal{T}$ has to accept RC-anti-dependency histories by hypothesis, and $E_5|\mathcal{H}$ is RC-anti-dependency. Then the prefix $\alpha \cdot \beta \cdot \gamma$ of E_5 is a valid execution of $\mathcal{T}$, since it is equal to E_2. However, when p_1 runs in E_4 after $\alpha \cdot \beta \cdot \gamma$ has been executed, it has to behave like in E_5, by executing α'. This is because T_3 is live and it does not invoke any write operation in γ, and hence it is invisible in E_4. Therefore E_4 and E_5 are indistinguishable to p_1, and E_4 must be a valid execution of $\mathcal{T}$ as well as E_5 is

However the history $E_4|\mathcal{H}$ violates opacity. Indeed, in any possible legal sequential history of T_1, T_2, and the completion T'_3 of the live transaction T_3, which does not violate the real-time order of T_2 and T'_3, we have that one of the following statements must be true: *i)* $\mathsf{T}'_3.\mathsf{read}(\mathsf{y})$ by T'_3 has to return v_1; *ii)* $\mathsf{T}_1.\mathsf{read}(\mathsf{x})$ by T_1 has to return v_2. This proves the theorem.

B.5 Proof of Theorem 5

Proof. We prove that TL2-RCAD guarantees TMS1 in two steps. First, proving that it is strict serializable. Second, proving that the response of each operation, including those invoked by live or pending transactions, is justified by a legal sequential history that: includes all transactions committed before it; excludes all transactions aborted before it, as well as all live transactions; and either includes or excludes any of the other concurrent transactions.

We first prove that $\mathcal{H}$ is strict serializable. Assume $\mathcal{T}_{TL2-RCAD}$ is the set of all histories generated by TL2-RCAD. Let $\mathcal{H} \in \mathcal{T}_{TL2-RCAD}$ be any arbitrary history, and let E be an execution that generates $\mathcal{H}$. We have three cases for E: *i)* no transaction in E calls *CheckAntiDependency* and commits; *ii)* only one transaction in E calls *CheckAntiDependency* and commits; or *iii)* more than one transaction in E call *CheckAntiDependency* and commit. We will show that in all the three cases, $\mathcal{H}$ is TMS1.

The first case is the simplest case. If no transaction calls *CheckAntiDependency* and commits, TL2-RCAD behaves as TL2-Extend, and thus $\mathcal{H}$ is also accepted by TL2-Extend. Since TL2-Extend guarantees opacity, it is guaranteed that $\mathcal{H}$ is opaque, and thus it is TMS1 (since opacity is stronger).

It is easy to show that the third case, where $\mathcal{H}$ has more than one transaction that calls *CheckAntiDep* and commits, can be reduced to the second case, where there is only one such transaction. This is because any two transactions that call *CheckAntidep* and commit cannot be concurrent (because of the way *anti_dep_lock* and *last_anti_dep* are checked and updated).

Regarding the second case, let us assume that $\mathcal{H}$ has only one transaction that calls *CheckAntiDep* and commits. We call this transaction T_{AD}. This means that we have one or more transactions that overwrite some reads of T_{AD} and commit before T_{AD}. For simplicity, assume that there is only one such transaction, called T_{owr}. T_{AD} has to be serialized before T_{owr} because of the anti-dependency. We show that we can find a serialization of all committed transactions where T_{AD} is serialized before T_{owr}.

First, it is guaranteed that neither the reads nor the writes of T_{owr} intersect with the writes of T_{AD}, because otherwise *CheckAntiDep* would fail and T_{AD} would abort. Also, since their write-sets do not intersect, the case where a write by T_{AD} overwrites a write on the same variable cannot happen, which also means that the final values in the shared memory are always legal. This means that both T_{AD} and T_{owr} comply with that serialization.

Now, Let T_{cmd} be any committed transaction in $\mathcal{H}$ other than T_{AD} and T_{owr}. We have four possible cases for T_{cmd}:

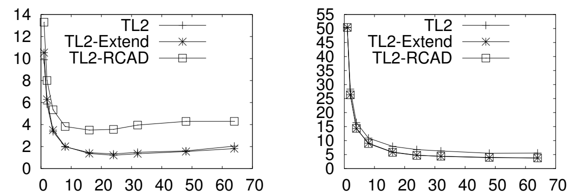
- $\mathsf{T}_{AD} \prec_{\mathcal{H}} \mathsf{T}_{cmd}$: This implies that $\mathsf{T}_{owr} \prec_{\mathcal{H}} \mathsf{T}_{cmd}$, which means that T_{cmd} would certainly observe all the writes of T_{AD} and T_{owr}.
- $\mathsf{T}_{cmd} \prec_{\mathcal{H}} \mathsf{T}_{AD}$: In this case, T_{cmd} cannot be enforced to be serialized after T_{owr}. This is because *i)* it is impossible that $\mathsf{T}_{owr} \prec_{\mathcal{H}} \mathsf{T}_{cmd}$ because T_{owr} commits after T_{AD} starts, *ii)* T_{owr} does not publish any write before the commit phase, which is after T_{AD} begins, and thus T_{cmd} did not read any value written by T_{owr}.

- T_{cmd} is a read-only transaction concurrent with T_{AD}: This case cannot happen because the way *last_ro* and *last_anti_dep* are checked and updated prevents two concurrent transactions from committing if one of them is read-only and the other calls *CheckAntiDep* and commits.
- T_{cmd} is an update transaction concurrent with T_{AD}: T_{cmd} cannot observe a partial set of T_{AD}'s or T_{owr}'s writes, because of the two phase locking approach used at commit. Thus, in order to forbid T_{AD} from serializing before T_{owr}, T_{cmd} should have read some value written by T_{owr} and missed some values written by T_{AD}. We prove that this is impossible as follows. Here, we have two cases. First, T_{cmd} commits before T_{AD}. In this case, T_{AD} should have observed that there is a transaction that reads from its writes during *CheckAntiDep*, and should have aborted in that case, which contradicts our assumption that T_{AD} commits. Second, T_{cmd} commits after T_{AD}. This means that T_{cmd} also observes anti-dependency and calls *CheckAntiDep*, which contradicts our assumption that only one transaction accepts anti-dependency and commits.

The next step is to prove that $\mathcal{H}$ is TMS1. Let T_{live} be any live, pending, or aborted transaction in $\mathcal{H}$. We prove that the return value of each operation in T_{live} is justified by a legal sequential history. Here we have three cases for T_{live}.

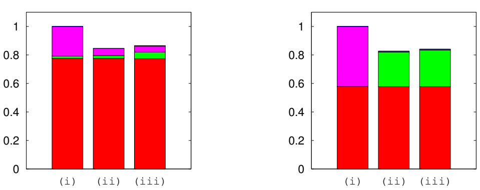
- $\mathsf{T}_{live} \prec_{\mathcal{H}} \mathsf{T}_{AD}$: This is impossible because we assume that T_{AD} is already committed.
- $\mathsf{T}_{AD} \prec_{\mathcal{H}} \mathsf{T}_{live}$: Like committed transactions, T_{live} would have no issue in serializing T_{AD} before T_{owr} in this case.
- T_{live} is concurrent with T_{AD}: Again, the only dangerous case is when T_{live} reads some value written by T_{owr} and misses some value written by T_{AD}. However, this case does not violate TMS1 because T_{live} and T_{AD} are concurrent, so T_{live} has the opportunity to exclude T_{AD} from the legal serialization that justifies the retunr values of its reads. It is worth to note that in this case, T_{live} will eventually abort (whether it is read-only or update transaction) because of RC-anti-dependencyit observes, as we show above for T_{cmd}. Such eventual abort preserves strict serializability in the whole system.

C Additional Evaluation Plots



(a) Genome (b) Labyrinth

Fig. 6. Throughput in STAMP benchmarks. X-axis: number of threads; Y-axis: Time (s).



(a) Genome (b) Labyrinth

Fig. 7. Commit/abort ratios in STAMP. X-axis: *(i)* is TL2; *(ii)* is TL2-Extended; and *(iii)* is TL2-RCAD.