Staleness and Local Progress in Transactional Memory

Hagit Attiya[1], Panagiota Fatourou[2], Sandeep Hans[3], and Eleni Kanellou[4](✉)

[1] Technion, Haifa, Israel
hagit@cs.technion.ac.il
[2] ICS-FORTH and University of Crete, Heraklion, Greece
faturu@ics.forth.gr
[3] IBM Research India, New Delhi, India
shans001@in.ibm.com
[4] ICS-FORTH, Heraklion, Greece
kanellou@ics.forth.gr

Abstract. A key goal in the design of *Transactional Memory* (TM) systems is ensuring liveness. *Local progress* is a liveness condition which ensures that a process successfully completes every transaction it initiates, if it continually re-invokes it each time it aborts. In order to facilitate this, several state-of-the-art TM systems keep multiple versions of data items. However, this method can lead to high space-related overheads in the TM implementation. Therefore, it is desirable to strike a balance between the progress that a TM can provide and its practicality, while ensuring correctness. A consistency property that limits the number of previous versions a TM may rely on, is *k-staleness*. It is a condition derivative of snapshot isolation, in which a transaction is not allowed to access more than k previous versions of a data item. This facilitates implementations that can take advantage of multi-versioning, while at the same time, contributing to the restriction of the space overhead introduced by the TM.

In this paper, we prove that no TM can ensure both local progress and *k-staleness*, if it is unaware of the transaction's accesses and can only keep a bounded number of versions for each data item.

Keywords: Transactional memory · Progress · Consistency · Impossibility

1 Introduction

Transactional memory (TM) [13] is an important programming paradigm, which offers synchronization of processes by providing the abstraction of the *transaction* to the programmer. A transaction contains several read and/or write accesses to shared memory, determined by a piece of sequential code, which the transaction encapsulates, in order to ensure that its execution is safe when it is concurrent with other transactions. The *data items* accessed by a transaction form its *data* *set*. If the execution of the transaction does not violate consistency, then it can terminate successfully (*commit*) and its writes to data items take effect atomically. Otherwise, all the updates of the transaction are discarded and the transaction *aborts*, i.e. it appears as if it had never taken place.

The possibility of aborting a transaction is an important feature that helps ensure consistency. At the same time, however, it can hinder liveness given that if a process finds itself in a situation where its transactions repeatedly aborts (and possibly have to be restarted), it spends computation time and resources without advancing its computational task.

Thus, it is desirable for a TM implementation to provide liveness guarantees that avoid such scenarios. *Local progress* (LP) [5] is such a desirable property. A TM implementation that ensures LP guarantees that even a transaction that is aborted will have to be restarted and re-executed a finite number of times before it finally commits. However, it was shown in [5] that LP cannot be achieved if the TM implementation has to provide *strict serializability* [17], traditionally implemented by database systems. This means that a TM implementation that ensures the stronger consistency property of *opacity* [11] cannot guarantee LP.

Snapshot isolation (SI) [2] is a consistency property weaker than opacity and strict serializability. While strict serializability requires that a single serialization point $*_T$ be found for each committed transaction T, so that T appears as if it had been atomically executed at $*_T$, snapshot isolation allows two serialization points, i.e. $*_{T|r}$, a read serialization point, and $*_{T|w}$, a write serialization point, for each committed transaction T, so that T's reads on data items appear as if they had atomically occurred at point $*_{T|r}$, while its writes appear as if they had atomically occurred at $*_{T|w}$. We define a condition that is derivative of snapshot isolation, called *k-staleness* (k-SL), where a read operation of some transaction may read one of the k last values that the data item has had.

Multi-version TMs that keep an unbounded number of previous versions may end up in implementations with high space complexity. Even more so, in practice, data items have shared representation, which results in even higher space overheads in real-life implementations. k-SL restricts the number of previous versions that a transaction may access in order to be consistent, i.e., in order to make it possible to assign it serialization points. In practice, then, this can lead to implementations that are more parsimonious in the use of space.

We extend the impossibility result of [5] to k-staleness, by showing that even a TM implementation that provides only k-SL cannot guarantee LP. This result concerns TM implementations where there is the underlying assumption that transactions may be unaware of each others' data sets and where a transaction T may not execute a read or write to some data item on behalf of some other transaction T'. This means that in such TMs, one transaction does not have access to the code executed by another transaction. We remark that this assumption is kind of standard in TM computing (and is needed also for the result of [3]).

To prove our results we present a comprehensive set of formal definitions, some introducing new concepts and others formalizing existing ones that are

Table 1. Properties of popular TM implementations.

STM	Consistency	Progress	Version	Other
DSTM [12]	Opacity	Obstruction free	1	–
TL2 [7]	Opacity	Minimal progressiveness	1	Invisible reads
NoREC [6]	Opacity	Lock-Free	1	–
PermiSTM	Opacity	Wait-free (RO)	1	–
Pessimistic LE	Opacity	Wait-free (RO)	1	–
SI-STM [21]	Snapshot isolation	Obstruction free	k	No revalidation
SI-TM [15]	Snapshot isolation	Wait-free (RO)	k	–

often met only in an informal way and, thus, they are mostly understood intuitively. We consider this as one of the contributions of the paper and we believe that Sect. 2 is interesting on its own.

For instance, the notion of data item *versioning* plays an important role in some theoretical results about transactional memory [18,19] and in several TM implementations [1,10,14,16,20,21]. These works only give informal descriptions of the term or rely on the intuitive understanding by the reader. Occasionally, the term is even used in order to refer to past values of a data item or to intermediate values that are used for local bookkeeping by an implementation.

To provide a clear model for our results, we present a formal definition of the concept of versions in TM, which reflects the way versioning is used in some of the prior theoretical results. For example, the limitations of keeping multiple versions for TM implementations are examined in [19]. The authors use a design principle by which a new version of a t-object is produced by an update transaction that has the t-object in its write set and commits, similar to our concept of past committed transaction. Similarly, in [18], reads on t-objects are considered to access values installed by transactions that have committed. Our definition is also compatible with existing k-version implementations. Table 1 summarizes some well-known TM implementations, presenting their properties according to the parameters we consider. Some of our definitions follow those in [4,8].

The rest of the paper is organized as follows: Sect. 2 provides the model on which we base our results, while Sect. 3 outlines the impossibility result. Finally, Sect. 4 summarizes our result and discusses its implication and context.

2 Definitions

2.1 Basic TM Concepts

We assume a system of n asynchronous processes that communicate through a shared memory. The shared memory is modeled as a collection of *base objects*, provided by hardware, which can be accessed by executing *primitives*, such as read, write, or `CAS`, on them.

A *transactional memory* (*TM*) supports the execution of pieces of sequential code in a concurrent setting through the use of *transactions*. Transactions contain read and write accesses to pieces of data, referred to as *data items*. Data items may be accessed simultaneously by multiple processes in a concurrent setting. A data item has a *shared representation*, also called *t-object*, out of base objects. A transaction T may *commit* or *abort*. If it commits, its updates on t-objects take effect, whereas if it aborts, its updates are discarded.

A *TM implementation* provides, for each process, the implementation of a set of routines, also called *t-operations*, which are invoked in order to execute transactions. Common such routines are listed in Table 2. *BeginTx* is called in order to start the execution of a transaction and *CommitTx* is called in order to attempt to commit a transaction. T-objects are accessed by calling t-operations *Read* and *Write*. When a transaction initiates the execution of a t-operation, we say that it *invokes* it, and a *response* is returned to the transaction when the t-operation execution terminates. Invocations and responses are referred to as *actions*.

Table 2. Invocations and possible responses of t-operations by a transaction T.

t-operation	Invocation	Valid response	Description
BeginTx	T.BeginTx	T.ACK	Initiates transaction T
CommitTx	T.CommitTx	T.committed or T.aborted	Attempts to terminate T successfully
Read	T.Read(x)	value v in some domain V or T.aborted	Reads the value of t-object x
Write	T.Write(x, v)	T.ACK or T.aborted	Writes value v to t-object x

In the following, $Read(x, v)$ denotes an instance of a *Read* t-operation executed by some transaction. It accesses t-object x and receives response v. Furthermore, $Write(x, v)$ denotes an instance of a *Write* t-operation that writes v to t-object x. We say that a transaction *reads* a data item when it invokes an instance of *Read* on the t-object of the data item, and that it *writes* to a data item when it invokes an instance of *Write* on the t-object of the data item. (In such cases, we may abuse terminology and say that a transaction reads or writes a t-object, respectively.) The *read set* of a transaction T, denoted rset(T), is the set of data items that T reads, while its *write set*, denoted wset(T), is the set of data items that T writes to. The union of read set and write set is the *data set* of T.[1]

[1] The definitions of read set, write set, and data set are formulated under the implicit assumption that transactions only execute their own code and do not perform reads

A *history* is a (possibly infinite) sequence of invocations and responses of t-operations. For a history H, $H|p$ denotes the subsequence of all those actions pertaining to process p. Similarly, $H|T$ denotes the subsequence of all those actions pertaining to transaction T. We remark that any of those subsequences may be empty. We denote by λ the empty sequence.

If $H|T$ is not empty, then T *is in* H. We denote by $\mathsf{txns}(H)$ the set of all transactions in H. Two histories H and H' are *equivalent*, denoted $H \equiv H'$, if $\mathsf{txns}(H) = \mathsf{txns}(H')$ and for every transaction $T \in \mathsf{txns}(H)$ and every process p, it holds that $H|T = H'|T$ and $H|p = H'|p$.

A history H is *well-formed* if for every transaction T in $\mathsf{txns}(H)$, $H|T$ is an alternating sequence of invocations of t-operations and their valid responses, starting with $T.BeginTx$, such that (i) no further invocation follows a $T.\mathsf{committed}$ or $T.\mathsf{aborted}$ response in $H|T$, and (ii) given another transaction $T' \in \mathsf{txns}(H)$ executed by the same process as T, either the last action of $H|T$ is $T.\mathsf{committed}$ or $T.\mathsf{aborted}$ and precedes the first action of $H|T'$ in H or the last action of $H|T'$ is $T'.\mathsf{committed}$ or $T'.\mathsf{aborted}$ and precedes the first action of $H|T$ in H. We only consider well-formed histories.

T is *committed* in H, if $H|T$ ends with $T.\mathsf{committed}$. It is *aborted* in H, if $H|T$ ends with $T.\mathsf{aborted}$. T is *completed* in H, if it is either committed or aborted in H; otherwise, it is *live*. If $H|T$ ends with an invocation of $T.\mathsf{CommitTx}$, then T is *commit-pending* in H. A history H is *complete* if all transactions in $\mathsf{txns}(H)$ are completed. Let $H|\mathsf{com}$ be the projection of H on actions performed by the committed transactions in H. A *completion* of a finite history H is a (well-formed) complete history H' such that $H' = HH''$, where H'' is a sequence of actions where any action is either $T.\mathsf{committed}$ or $T.\mathsf{aborted}$, for every transaction T that is commit-pending in H. The set of completions of H is denoted $\mathsf{comp}(H)$.

A history H imposes a partial order, called *real-time order*, on t-operations: For two t-operations o_i, o_j in H, we say that o_i *precedes* o_j in H, denoted $o_i \prec^o_H o_j$, if the response of o_i occurs before the invocation of o_j in H. A history H is *operation-wise sequential* if for every pair of t-operations o_i, o_j in H, either $o_i \prec^o_H o_j$ or $o_j \prec^o_H o_i$. A history H further imposes a partial (real-time) order on transactions in it. For two transactions $T_i, T_j \in \mathsf{txns}(H)$, we say that T_i *precedes* T_j in H, denoted $T_i \prec^T_H T_j$, if T_i is complete in H and the last action of $H|T_i$ appears in H before the first action of $H|T_j$. A history H is *sequential* if for every pair of transactions $T_i, T_j \in \mathsf{txns}(H)$, either $T_i \prec^T_H T_j$ or $T_j \prec^T_H T_i$.

A $Read(x, v)$ t-operation r executed by transaction T in a *sequential* history S is *legal* if either (i) T contains a $Write(x, v)$ t-operation w which precedes r; or in case (i) does not hold, if (ii) $\mathsf{txns}(S)$ contains a committed transaction T', which executes a $Write(x, v)$ t-operation w', and w' is the last such t-operation by a committed transaction that precedes T; or in case neither (i) nor (ii) hold, if (iii) v is the initial value of x. A transaction T in S is *legal* if all its *Read* t-operations that do not receive $T.\mathsf{aborted}$ as a response, are legal in S. A complete sequential history S is *legal* if every committed transaction T in S is legal in S.

or writes by executing code that pertains to other transactions or by other forms of light-weight helping.

We define a *configuration* of the system as a vector that contains the state of each process and the state of each base object. This vector describes the system at some point in time. In an initial configuration all processes are in initial states and all base objects hold initial values. A *step* by some process p consists of the application of a primitive on a base object by p, or of the invocation or the response of a t-operation by a transaction executed by p; the step may also contain some local computation by p which cannot cause changes to the state of the base objects but it may change local variables used by p.

An *execution* is a (finite or infinite) sequence of steps. We use $\alpha\beta$ to denote the execution α immediately followed by the execution β. An execution α may also contain a $stop(p)$ event, for each process $p \in P$, which indicates that, after that point, process p is faulty (i.e. it does not take any further steps in α). Denote by $F(\alpha)$ the set of faulty processes in α, i.e. for each process $p \in F(\alpha)$, there is a $stop(p)$ event in α.

An execution α of a TM implementation is *feasible*, starting from a configuration C, if the sequence of steps performed by each process follows the algorithm for that process (starting from its state in C) and, for each base object, the responses to the primitives performed on the object are in accordance with its specification (and the value stored in the object at configuration C). Let $H(\alpha)$ be the subsequence of α consisting only of the invocations and responses of t-operations in α. We refer to $H(\alpha)$ as the history of α.

2.2 TM Consistency

Commonly used consistency conditions for transactional memory include *strict serializability* [17] and *opacity* [11]. Roughly speaking, some history H is strictly serializable if it is possible to assign a *linearization point* between the invocation and the response of each transaction in $H|\mathsf{com}$ and possibly of some of the commit-pending transaction in H such that the sequential history resulting from executing the transactions in the order defined by their linearization points, is legal. Opacity is a consistency condition stronger than strict serializability, which further restricts the responses of t-operations obtained by live transactions.

Assigning a single linearization point for each transaction T provides an atomicity guarantee for all the accesses (reads and writes) to data items by T. However, in order to avoid the performance overhead that is usually incurred to ensure these guarantees, weaker consistency conditions are often employed. A way of relaxing the strict requirements imposed by the aforementioned conditions, is that of assigning two linearization points per transaction, one to (a subset of its) *Read* t-operations and another to the rest of its t-operations. *Snapshot isolation* [2] is a weaker consistency condition which employs this strategy. Roughly speaking, the effect that the two linearization points per transaction T have, is that of making T appear to be split into two subtransactions, where one of the subtransactions contains the *global Read* t-operations that T performs on data items (i.e. those t-operations that read data items which are never written to by T), while the second subtransaction contains all $Write$ t-operations and all remaining *Read* t-operations performed by T. This practice is reminiscent of taking a "snapshot" of the values of the data items in T's read set (that are not written by T) at some point in the beginning of the transaction and of reading the data item values from that snapshot whenever necessary, hence the name of the consistency condition. This use of two linearization points allows for more flexibility, because when it comes to finding an equivalent legal sequential history, the two subtransactions can be treated as separate entities that do not have to be serialized together. Instead, they can be interleaved with the linearization points of other transactions. This allows a wider collection of histories to be considered correct under snapshot isolation. In the following, we formalize the intuitive notion of treating one transaction as split into two subtransactions and use this formalism in order to provide a definition for snapshot isolation.

Given a history H, a $Read(x)$ t-operation r invoked by some transaction $T \in txns(H)$ is *global*, if T did not invoke a $Write(x, v)$ before invoking r. Let $T|r_g$ be the longest subsequence of $H|T$ consisting only of global read invocations and their corresponding responses. Let $T|o$ be the subsequence of $H|T$ consisting of all $Read$ and $Write$ t-operations in $H|T$ other than those in $T|r_g$. Recall that λ is the empty sequence. For each committed transaction T, let $\mathsf{readTx}_g(T)$ and $\mathsf{other}(T)$ be the following histories:

- if $T|r_g = \lambda$ then $\mathsf{readTx}_g(T) = \lambda$, otherwise $\mathsf{readTx}_g(T) = T.\mathsf{BeginTx}, T.\mathsf{ACK}, T|r_g, T.\mathsf{CommitTx}, T.\mathsf{committed}$.
- if $T|o = \lambda$ then $\mathsf{other}(T) = \lambda$, otherwise $\mathsf{other}(T) = T.\mathsf{BeginTx}, T.\mathsf{ACK}, T|o, T.\mathsf{CommitTx}, T.\mathsf{committed}$.

Definition 1. *A history H satisfies* **snapshot isolation**, *if there exists a history $H' \in \mathsf{comp}(H)$, such that for every committed transaction T in H' it is possible to insert a read point $*_{T,r}$ and a write point $*_{T,w}$ such that*

(i) $*_{T,r}$ precedes $*_{T,w}$,
(ii) both $*_{T,r}$ and $*_{T,w}$ are inserted after the first action of T in H' and before the last action of T in H', and
(iii) if $\sigma_{H'}$ is the sequence defined by these points, in order, and S is the history obtained by replacing each $*_{T,r}$ with $\mathsf{readTx}_g(T)$ and each $*_{T,w}$ with $\mathsf{other}(T)$ in $\sigma_{H'}$, then S is legal.

Snapshot isolation is weaker than strict serializability, i.e. all histories that are strictly serializable satisfy snapshot isolation as well. Definition 1 provides a weaker form of snapshot isolation in comparison to standard previous definitions provided in the literature [2,9,21]. This is so because, in addition to ensuring the conditions of Definition 1, the definitions in [2,9,21] impose the extra constraint that from any two concurrent transactions writing to the same data item, only one can commit. Note also that Definition 1 does not impose any restriction on the value returned by a $Read$ t-operation on some data item by a transaction, if the transaction has written to the data item before invoking this $Read$ t-operation.

Figure 1 shows a history H which satisfies snapshot isolation but not strict serializability. H contains two transactions, T_1 and T_2, which each perform a

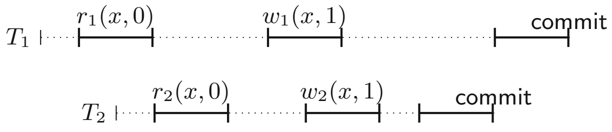

Fig. 1. An SI history which does not satisfy serializability.

Read and a *Write* t-operation on x. Both transactions read the value 0 for x and subsequently, write the value 1 to x. In order for this history to be strictly serializable, it should be possible to assign a single linearization point between the invocation and the response of T_1 and a single linearization point between the invocation and the response of T_2, so that an equivalent and legal sequential history can be constructed based on the order of these linearization points. Since the executions of T_1 and T_2 are overlapping, by assigning linearization points, we end up either with equivalent sequential history $S = T_1T_2$ or with equivalent sequential history $S' = T_2T_1$. Notice, however, that neither of those histories is legal, since for example, in S, the *Read* t-operation of T_2, r_2, returns the value 0 for x, although T_1, which writes the value 1 to x, has committed before T_2 in S. Conversely also for S'. Therefore, it isn't possible, by assigning a single linearization point to each transaction, to get an equivalent, legal sequential history, and therefore H is not strictly serializable.

On the contrary, it is possible to insert read points $*_{T_1,r}$, $*_{T_2,r}$ and write points $*_{T_1,w}$, $*_{T_2,w}$, for example in the order $*_{T_1,r}$, $*_{T_2,r}$, $*_{T_1,w}$, $*_{T_2,w}$, so that, by replacing $*_{T_1,r}$ with $\mathsf{readTx}_g(T_1)$, $*_{T_2,r}$ with $\mathsf{readTx}_g(T_2)$, $*_{T_1,w}$ with $\mathsf{other}(T_1)$, and $*_{T_2,w}$ with $\mathsf{other}(T_2)$, then, the equivalent sequential history that results, namely $S = \mathsf{readTx}_g(T_1)\mathsf{readTx}_g(T_2)\mathsf{other}(T_1)\mathsf{other}(T_2)$, is legal, given that in that case, both $\mathsf{readTx}_g(T_1)$ and $\mathsf{readTx}_g(T_2)$ contain a $Read(x)$ t-operation which in either case, legally returns the value 0 for x, since both those transactions commit in S before the invocation of transactions $\mathsf{other}(T_1)\mathsf{other}(T_2)$, which are the ones containing $Write(x, 1)$ t-operations, modifying the value of x.

2.3 Progress Conditions

A pair $\langle \alpha, F \rangle$ of an execution α produced by a TM implementation I and a set of processes $F \subset P$, is *fair*, if for each process $p \in P \setminus F$, the following holds:

– If α is finite, then p does not have a live transaction at the end of $H(\alpha)$ and p's last transaction in $H(\alpha)$ (if any) is not aborted,
– If α is infinite, then α contains either infinitely many steps by p or infinitely many configurations in which p does not have a live transaction.

For each TM implementation I, let $HF(I) = \{\langle H(\alpha), F(\alpha)\rangle | \forall \alpha$ produced by I s.t. $\langle \alpha, F(\alpha)\rangle$ is fair$\}$.

Local progress is a set $\mathcal{LP}$ of pairs s.t. for each pair $\langle H, F\rangle \in \mathcal{LP}$, H is a well-formed history and $F \subset P$ is a set of processes for which the following hold:

- If H is finite, then for each process $p \in P \setminus F$, p's last transaction in H (if any) is committed.
- If H is infinite, then for each process $p \in P \setminus F$, H contains either infinitely many *commit* events for p or there are infinitely many prefixes of H such that for each such prefix H' the last transaction (if any) executed by p in H' is committed (i.e. p does not have a live transaction at the end of H').

We say that a TM implementation I satisfies local progress (LP) if $HF(I) \subseteq \mathcal{LP}$. Intuitively, local progress guarantees that the transactions of any process not only terminate, but furthermore, that every non-faulty process eventually receives a commit response for each transaction it initiates, independently of the actions of the other processes in the system. This implies that, should a process decide to restart an aborted transaction, then this transaction will not indefinitely terminate by aborting.

3 Impossibility Result

In this section, we provide definitions regarding the staleness of values of data items in TM and use those to formally define k-staleness. Then, we use this definition in order to prove that it is not possible to come up with a TM system that can ensure local progress and k-staleness while tolerating failures, i.e. the existence of faulty processes.

3.1 Stale Values in TM

Consider an operation-wise sequential history H and a *Read* t-operation r on data item x by transaction T in H. Let T_{pw} be a committed transaction which writes x and its $CommitTx$ t-operation c_{pw} is such that $c_{pw} \prec_H^o r$. Then, we say that T_{pw} is a *past committed write transaction* for r. We define the *last committed write transaction* for r as the past committed write transaction T_{lw} for r for which the following holds: if c_{lw} is the $CommitTx$ t-operation of T_{lw}, then there is no other past committed transaction T' for r such that, if c' is the commit t-operation of T', then $c_{lw} \prec_H^o c'$.

Let Seq_r be the sequence of all past committed write transactions of r, defined by the order of their $CommitTx$ t-operations. The last transaction in this sequence is T_{lw}. Let S_r^k be the set that contains those transactions that are determined by the k last transactions in Seq_r, if $|Seq_r| > k$, and the set that contains all transactions in Seq_r, otherwise. We refer to S_r^k as the set of the k *last committed transactions* for r. Each of the values written for x during the last $Write$ performed for x by each of the transactions in S_r^k is referred to as a *previous value* of x. We denote by V_r^k the set of all these values. If $|S_r^k| < k$, let V_r^k contain also the initial value for x.

A *Read*(x) t-operation invoked by a transaction T in H is called *global* if T did not invoke a $Write$ for x before invoking this *Read*. An operation-wise sequential history H is *k-value* if for every global *Read*(x, v) executed by a transaction T

in H, it holds that $v \in V_r^k$. A TM algorithm is *k-value* if every operation-wise sequential history that it produces is k-value. Notice that a TM implementation is *single-value* if in each operation-wise sequential history H that it produces, for every global $Read(x, v)$ t-operation r, v is the value written by the last write for x performed by the last committed write transaction for r; if such a transaction does not exist, then v is the initial value of x.

Definition 2. *An operation-wise sequential history H satisfies k-staleness, if it satisfies snapshot isolation and it is additionally k-value.*

A TM implementation satisfies k-staleness if every operation-wise sequential history it produces satisfies k-staleness. We remark that k-staleness is a weak property that does not provide any consistency guarantee for histories produced by the implementation that are not operation-wise sequential. This makes our impossibility result stronger.

3.2 Impossibility of *k*-staleness and Local Progress

In order to prove the following theorem, we construct a fair history based on the use of a transaction T_0 which reads two distinct data items x and y. We construct the history so that the *Read* t-operations of T_0 are interleaved with *Write* t-operations to x and y, and argue that T_0 can not commit.

$$\begin{tikzcd}
C^0 \arrow[r, dotted, "\alpha^1\alpha^2\ldots\alpha^{i-1}"] & C^{i-1} \arrow[r, "\alpha_0^i", "T_0.r_1"'] & C_0^i \arrow[r, dotted] & C_{j-1}^i \arrow[r, "\alpha_j^i", "T_j"'] & C_j^i \arrow[r, dotted] & C_k^i \arrow[r, "\alpha_{k+1}^i", "T_0.r_2"'] & C^i
\end{tikzcd}$$

Fig. 2. Configurations in the proof of Theorem 1.

Theorem 1. *There is no TM implementation I that ensures both k-staleness and local progress and tolerates one process failure.*

Proof. The proof is by contradiction. Consider a TM implementation that ensures k-SL and LP, and assume that it tolerates one process failure. We will construct a troublesome history H in which a transaction T_0 never commits. H will be constructed to be an infinite fair history in which process p_0, which executes T_0, takes infinitely many steps. To construct H, we employ an instance of the following transaction (which, as we prove, repeatedly aborts forever in H):

– $T_0 = r_1(x)r_2(y)$, executed by p_0, where x and y are two distinct data items.

We also employ an infinite number of instances of the following k transactions, executed by a different process p_1:

– for every j, $1 \leq j \leq k$, $T_j = w_{j,1}(x, v_j^i), w_{j,2}(y, v_j^i)$, executed by p_1, where for every integer $i > 0$, v_j^i is a distinct value other than 0, used by the ith instance of T_j.

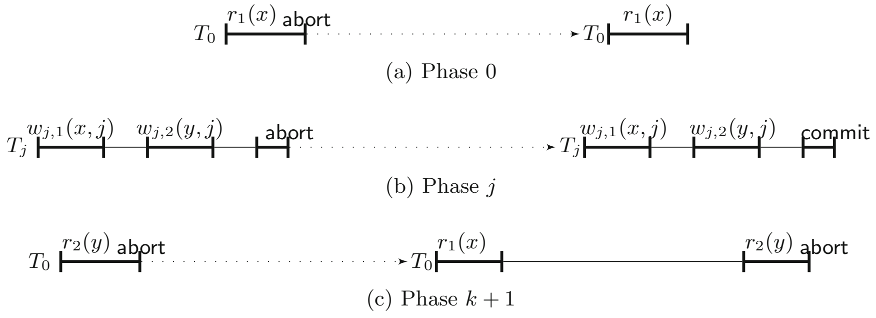

Fig. 3. An illustration of the phases performed in the proof of Theorem 1

For simplicity, we have omitted the invocations of BeginTx and CommitTx when describing transactions $T_0, \ldots, T_k$ above.

Let the initial values of x and y be 0. An adversary constructs history H as described below:

Phase 0**:** Process p_0 starts executing solo from the initial configuration to perform transaction T_0 and invokes r_1 on x. As long as r_1 returns T_0.aborted, phase 0 is repeated until r_1 returns a value (we later prove that this indeed occurs). Then, Phase 1 starts.

Phases $j = 1$ **to** k**:** These phases are constructed inductively on j as follows. Fix j, $1 \leq j \leq k$, and assume that phases $1, \ldots, j-1$ have been constructed. Let C_{j-1} be the configuration at the end of phase $j-1$. Phase j starts from C_{j-1}. In phase j, process p_1 does the following: It starts executing transaction T_j. As long as the execution of T_j completes with T_j.aborted, p_1 restarts the execution of T_j from the resulting configuration. If T_j commits, Phase j ends. We later prove that T_j must indeed eventually commit, and we denote by C_j the resulting configuration.

Phase $k+1$**:** Process p_0 resumes executing solo from C_k to continue performing transaction T_0 and invokes r_2 on y. As long as r_2 returns T_0.aborted, the adversary repeats all phases from the resulting configuration, starting from phase 0. We later prove that r_2 must always return T_0.aborted. Therefore, the result is an infinite, fair history H. This history violates local progress since T_0 never commits.

Figure 3 illustrates the phases described above. Figure 4 illustrates the adversary's strategy for the case $k = 1$, namely, for a single-version TM.

The next claim shows that the adversary can indeed follow the strategy described above and that the resulting history has the required properties. We denote by C^0 the initial configuration.

Claim 1. *For each integer $i > 0$, the TM implementation I has a feasible execution α^i, starting from configuration C^{i-1}, such that $\alpha^i = \alpha^i_0 \alpha^i_1 \ldots \alpha^i_k \alpha^i_{k+1}$, where:*

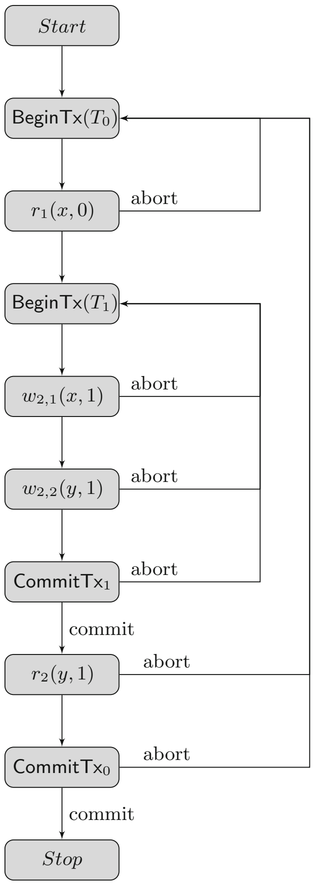

Fig. 4. Flowchart of the adversary's strategy for $k = 1$.

– *α_0^i is a solo execution by p_0,*
– *α_j^i is a solo execution by p_1, for $1 \leq j \leq k$,*
– *α_{k+1}^i is a solo execution by p_0,*

so that:

1. *α_0^i is a finite execution in which p_0, starting from C^{i-1}, repeatedly executes T_0 until r_1 returns a value other than T_0.*aborted*; let C_0^i be the resulting configuration.*
2. *α_j^i, $\forall j$, $1 \leq j \leq k$, is a finite execution starting from C_{j-1}^i, in which T_j writes value v_j^i to x and y and commits; let C_j^i be the resulting configuration.*
3. *α_{k+1}^i is a finite execution by p_0 starting from configuration C_k^i such that T_0 is aborted in α_{k+1}^i; let C^i be the resulting configuration.*

Proof. The proof is by induction on i. Fix any $i > 0$ and assume that we have constructed $\alpha^1, \ldots, \alpha^{i-1}$; let C^{i-1} be the configuration we reach when $\alpha^1, \ldots, \alpha^{i-1}$

is applied from C^0. We prove that the claim holds for i. Figure 2 shows the configurations.

We first show (1), i.e. that *there is a feasible execution* α_0^i*, starting from* C^{i-1} *with the required properties.*

Notice that no transaction is live at C_0. This and the induction hypothesis imply that no transaction is live at C^{i-1}. So, if process p_0 starts executing solo from configuration C^{i-1}, it (re-)initiates transaction T_0 and invokes r_1 on x. Assume, by the way of contradiction, that either, repeatedly forever, r_1 terminates in a T_0.aborted event and p_0 re-initiates T_0 and re-invokes r_1, or that one of these invocations of r_1 never terminates. Let γ_0^i be the infinite solo execution by p_0, starting from C^{i-1}, in which this occurs. Consider the execution $\delta_0^i = \alpha^1 \dots \alpha^{i-1}\gamma_0^i$. Then $\langle \delta_0^i, \emptyset \rangle$ is fair. This is so because δ_0^i is infinite and the induction hypothesis implies that the following hold: (1) there are infinitely many configurations in δ_0^i (namely, all configurations in γ_0^i) in which p_1 does not have a live or aborted last transaction, and (2) p_0 takes an infinite number of steps in δ_0^i. However, $\langle H(\delta_0^i), \emptyset \rangle \notin \mathcal{LP}$. We remark that p_0 never commits the transaction it executes in δ_0^i. To prove that $\langle H(\delta_0^i), \emptyset \rangle \notin \mathcal{LP}$, we consider the following two cases.

1. $H(\delta_0^i)$ is finite. Notice that this holds only if one of the invocations of r_1 never terminates. Then p_0, which is non-faulty, has a live transaction at the end of $H(\delta_0^i)$.
2. $H(\delta_0^i)$ is infinite. Notice that this holds if r_1 repeatedly forever returns T_0.aborted. Then, for p_0, it neither contains infinitely many commit events, nor are there infinitely many prefixes of $H(\delta_0^i)$, in which the last transaction executed by p_0 in the prefix, commits.

We now use similar arguments to prove point (2) of Claim 1, i.e. that *for each* j*,* $1 \le j \le k$*, there exists a finite execution* α_j^i *starting from* C_{j-1}^i *such that* α_j^i *is a solo execution by* p_1 *resulting in configuration* C_j^i*, in which* T_j *eventually commits, given that* p_1 *re-executes* T_j *each time it aborts.*

Let f, $1 \le f \le k$, be the first index for which the claim does not hold. Let γ_f^i be the infinite solo execution by p_1, starting from C_{f-1}^i, in which either some t-operation invoked by T_f never terminates, or repeatedly forever, some t-operation executed by T_f aborts, and T_f is re-initialized. Consider the execution $\delta_f^i = \alpha^1 \dots \alpha^{i-1}\alpha_0^i \ \mathtt{stop}_{p_0} \ \alpha_1^i \dots \alpha_{f-1}^i \gamma_f^i$. Then, $\langle \delta_f^i, \{p_0\} \rangle$ is fair. This is because δ_f^i is infinite and the following holds: (1) there are infinitely many configurations in δ_f^i (namely, all configurations in γ_f^i) in which p_1 does not have a live or aborted last transaction, and (2) p_1 takes an infinite number of steps in δ_f^i. However, $\langle H(\delta_f^i), \{p_0\} \rangle \notin \mathcal{LP}$. To prove this, we consider the following two cases.

1. $H(\delta_f^i)$ is finite. Then p_1, which is non-faulty, has a live transaction at the end of $H(\delta_f^i)$.
2. $H(\delta_f^i)$ is infinite. Then, for p_1, it neither contains infinitely many commit events, nor are there infinitely many prefixes of $H(\delta_f^i)$, in which the last transaction executed by p_1 in the prefix, commits.

This contradicts the fact that I ensures local progress. Therefore, it holds that for each j, $1 \leq j \leq k$, there exists a finite execution α_j^i starting from C_{j-1}^i such that α_j^i is a solo execution by p_1 resulting in configuration C_j^i, in which T_j eventually commits, given that p_1 re-executes T_j each time it aborts. Moreover $\alpha_0^i \ldots \alpha_k^i$ is a feasible execution starting from C^{i-1}.

We finally show that point (3) of Claim 1 holds, i.e. that there is a feasible execution α_{k+1}^i by p_0 starting from configuration C_k^i such that T_0 is aborted in α_{k+1}^i. Starting from C_k, we let process p_0 execute solo to continue its execution with the invocation of r_2. Let α_{k+1}^i be the solo execution by p_0, starting from C_k^i, until r_2 completes; if this does not happen, let α_{k+1}^i be the infinite solo execution by p_0 starting from C_k^i. Let $\delta_{k+1}^i = \alpha^1 \ldots \alpha^{i-1}\alpha_0^i \ldots \alpha_k^i \alpha_{k+1}^i$.

We prove that if r_2 *returns in* δ_{k+1}^i*, then it returns* $T_0.\mathsf{aborted}$. Assume, by the way of contradiction, that r_2 returns a value (and not $T_0.\mathsf{aborted}$) in δ_{k+1}^i. By point (2) of Claim 1 (proved above), each of the transactions $T_1, \ldots, T_k$ executed in δ_{k+1}^i eventually commits. This in turn means that each T_j, $1 \leq j \leq k$, writes value v_j^i to both t-objects x and y. Since I is a k-version TM implementation, it follows that r_2 returns one of the last k written values for y, i.e. a value v_j^i, $j \in \{1, 2, \ldots, k\}$. However, neither of those values for t-object y is consistent with the value returned by r_1 in α_0^i which must be one of the k versions of x at configuration C^{i-1}. This contradicts the assumption that I satisfies snapshot isolation. Therefore, if r_2 returns, it returns $T_0.\mathsf{aborted}$.

We finally prove that α_{k+1}^i *is finite.* Assume, by way of contradiction, that α_{k+1}^i is infinite. Then, $\langle \delta_{k+1}^i, \emptyset \rangle$ is fair. This is so because δ_{k+1}^i is infinite and the following holds: (1) there are infinitely many configurations in δ_{k+1}^i (namely, all configurations in α_{k+1}^i) in which p_1 does not have a live or aborted last transaction, and (2) p_0 takes an infinite number of steps in δ_{k+1}^i. However, $\langle H(\delta^i), \emptyset \rangle \notin \mathcal{LP}$. This is so because $H(\delta_{k+1}^i)$ is finite, and p_0, which is non-faulty, has a live transaction at the end of $H(\delta_{k+1}^i)$. This contradicts the fact that I ensures local progress. Therefore, it holds that α_{k+1}^i is a finite execution by p_0 starting from configuration C_k^i in which T_0 is aborted. Denote by C^i the resulting configuration. Notice that execution $\alpha^i = \alpha_0^i \ldots \alpha_k^i \alpha_{k+1}^i$ is feasible from C^{i-1}. □

Notice that execution α_0^i corresponds to an execution of Phase 0. Since I satisfies snapshot isolation, the value returned by r_1 in α_0^i must be 0, i.e. the initial value for x. After Phase 0, the adversary moves to Phase 1. Notice that, for each j, $1 \leq j \leq k$, execution α_j^i corresponds to an execution of Phase j. In Phase j, T_j commits. After T_k commits, the adversary moves to Phase $k+1$. Execution α_{k+1}^i corresponds to an execution of Phase $k+1$. Claim 1 shows that each time the adversary executes phases $0, \ldots, k+1$, the resulting execution is finite.

The next claim shows that execution $\alpha = \alpha^0\alpha^1\alpha^2 \ldots$ is a feasible fair execution of I which violates local progress.

Claim 2. *Let* $\alpha = \alpha^0\alpha^1\alpha^2 \ldots$*. Then, the following hold:*

1. α *is a feasible infinite execution starting from* C^0*;*

2. the pair $\langle \alpha, \emptyset \rangle$ *is fair*
3. $\langle H(\alpha), \emptyset \rangle \notin \mathcal{LP}$*.*

Proof. Lemma 1 implies that, for each $i > 0$, α^i is a feasible execution starting from C^{i-1} in which T_0 is aborted. Therefore, α is a feasible infinite execution. Moreover, $\langle \alpha, \emptyset \rangle$ is fair. This is so because each process takes infinite steps in α. Since all t-operation invocations in α receive a response, $H(\alpha)$ is infinite as well. However, $\langle H(\alpha), \emptyset \rangle \notin \mathcal{LP}$. This is so since neither does $H(\alpha)$ contain infinite many commit responses for process p_0 (specifically, transaction T_0 that is repeatedly invoked by p_0 always completes by aborting in α), nor does $H(\alpha)$ contain infinitely many prefixes in which the last transaction executed by p_0 is committed. This contradicts the fact that I ensures local progress. □

Theorem 1 is an immediate consequence of Claims 1 and 2. □

4 Discussion

We have studied the progress that can be provided by a TM implementation that ensures k-staleness, a condition derivative of snapshot isolation, but where processes can crash, i.e., unexpectedly stop executing in between t-operations. Specifically, we have studied whether such a TM implementation can guarantee *local progress* for transactions. We provide a definition of local progress based on fair executions, which avoids the need to study other types of process malfunctions, such as the so-called *parasitic* processes. Parasitic processes have not suffered crash failures but still do not attempt to commit the transactions that they invoke, continuously invoking *Read* or $Write$ t-operations instead [5].

Our impossibility result could possibly be extended to other, even weaker consistency conditions, for example, *adaptive consistency* [3], because most consistency conditions require that each transaction obtains a consistent view of its read set. In this case, and assuming that a system is k-version, an adversary can always come up with a troublesome strategy that executes more than k update transactions between two reads of some read-only transaction.

It is interesting to explore the use of stronger primitives, such as m-assignment, an operation that atomically writes values to m different base objects, and other objects, such as snapshots, for implementing stronger consistency conditions, such as serializability, in conjunction with local progress. Alternatively, the impossibility might be sidestepped for weaker consistency conditions also, by using other assumptions and primitives, which might be less complex.

Acknowledgment. This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project. It has further been co-financed by the European Union and Greek national funds through the Operational Program Competitiveness, Entrepreneurship and Innovation, under the call RESEARCH – CREATE – INNOVATE (project code:T1EDK-02857)

References

1. Attiya, H., Hillel, E.: A single-version STM that is multi-versioned permissive. Theory Comput. Syst. **51**(4), 425–446 (2012)
2. Berenson, H., Bernstein, P., Gray, J., Melton, J., O'Neil, E., O'Neil, P.: A critique of ANSI SQL isolation levels. In: SIGMOD (1995)
3. Bushkov, V., Dziuma, D., Fatourou, P., Guerraoui, R.: The PCL theorem: transactions cannot be parallel, consistent and live. In: SPAA (2014)
4. Bushkov, V., Dziuma, D., Fatourou, P., Guerraoui, R.: The PCL theorem: transactions cannot be parallel, consistent, and live. J. ACM **66**(1), 2:1–2:66 (2019). https://doi.org/10.1145/3266141
5. Bushkov, V., Guerraoui, R., Kapalka, M.: On the liveness of transactional memory. In: PODC (2012)
6. Dalessandro, L., Spear, M.F., Scott, M.L.: Norec: streamlining STM by abolishing ownership records. In: PPoPP (2010)
7. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: DISC (2006)
8. Dziuma, D., Fatourou, P., Kanellou, E.: Consistency for transactional memory computing. In: Guerraoui, R., Romano, P. (eds.) Transactional Memory. Foundations, Algorithms, Tools, and Applications. LNCS, vol. 8913, pp. 3–31. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-14720-8_1
9. Fekete, A., Liarokapis, D., O'Neil, E., O'Neil, P., Shasha, D.: Making snapshot isolation serializable. ACM Trans. Database Syst. **30**(2), 492–528 (2005). https://doi.org/10.1145/1071610.1071615
10. Fernandes, S.M., Cachopo, J.a.: Lock-free and scalable multi-version software transactional memory. In: PPoPP (2011)
11. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: PPoPP (2008)
12. Herlihy, M., Luchangco, V., Moir, M., Scherer III, W.N.: Software transactional memory for dynamic-sized data structures. In: PODC (2003)
13. Herlihy, M., Moss, J.E.B.: Transactional memory: architectural support for lock-free data structures. SIGARCH Comput. Archit. News **21**(2), 289–300 (1993)
14. Kumar, P., Peri, S., Vidyasankar, K.: A timestamp based multi-version STM algorithm. In: Chatterjee, M., Cao, J., Kothapalli, K., Rajsbaum, S. (eds.) ICDCN 2014. LNCS, vol. 8314, pp. 212–226. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-642-45249-9_14
15. Litz, H., Cheriton, D.R., Firoozshahian, A., Azizi, O., Stevenson, J.P.: SI-TM: reducing transactional memory abort rates through snapshot isolation. In: ASPLOS (2014)
16. Lu, L., Scott, M.L.: Generic multiversion STM. In: Afek, Y. (ed.) DISC 2013. LNCS, vol. 8205, pp. 134–148. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-41527-2_10
17. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM **26**(4), 631–653 (1979)
18. Perelman, D., Byshevsky, A., Litmanovich, O., Keidar, I.: SMV: selective multiversioning STM. In: Peleg, D. (ed.) DISC 2011. LNCS, vol. 6950, pp. 125–140. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-24100-0_9

19. Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in STM. In: PODC (2010)
20. Riegel, T., Felber, P., Fetzer, C.: A lazy snapshot algorithm with eager validation. In: DISC (2006)
21. Riegel, T., Fetzer, C., Felber, P.: Snapshot isolation for software transactional memory. In: TRANSACT (2006)