# The Distributed Computing Column

BY

**Panagiota Fatourou**

Department of Computer Science, University of Crete
P.O. Box 2208 GR-714 09 Heraklion, Crete, Greece
and
Institute of Computer Science (ICS)
Foundation for Research and Technology (FORTH)
N. Plastira 100. Vassilika Vouton
GR-700 13 Heraklion, Crete, Greece
faturu@csd.uoc.gr

# Consistency for Transactional Memory Computing

Dmytro Dziuma
FORTH-ICS, Greece
dixond@acm.lviv.ua

Panagiota Fatourou*
FORTH-ICS & University of Crete, Greece
faturu@csd.uoc.gr

Eleni Kanellou
IRISA, Université de Rennes, France & FORTH-ICS, Greece
eleni.kanellou@irisa.fr

**Abstract**

This paper provides *formal definitions* for a comprehensive collection of consistency conditions for transactional memory (TM) computing. We express all conditions in a uniform way using a formal framework that we present.

For each of the conditions, we provide two versions: one that allows a transaction $T$ to read the value of a data item written by another transaction $T'$ that can be live and not yet commit-pending provided that $T'$ will eventually commit, and a version which allows transactions to read values written only by transactions that have either commited before $T$ starts or are commit-pending. Deriving the first versions was not an easy task but it has some benefits: (1) this version of each condition is weaker than the second one and so it results to a wider universe of algorithms which there is no reason to exclude from being considered correct, and (2) some definitions work, as is, for universal constructions contributing towards unifying the two models.

The formalism for the presented consistency conditions does not base on any unrealistic assumptions, such as that transactional operations are executed atomically or that write operations write distinct values for data items. Making such assumptions facilitates the task of formally expressing the consistency conditions significantly, but results to formal presentations of them that are unrealistic, i.e. that cannot be used to characterize the correctness of most of the executions produced by any reasonable TM algorithm.

*Currently with École Polytechnique Fédérale de Lausanne (EPFL), Switzerland, as an EcoCloud visiting professor.

# 1 Introduction

Transactional memory (TM) [19, 27] is a promising parallel programming paradigm that aims at simplifying parallel programming by using the notion of a transaction. A *transaction* is a piece of code containing accesses to pieces of data, known as *data items*, which are accessed simultaneously by several processes in a concurrent setting. A transaction may either *commit* and then its updates are effectuated or *abort* and then its updates are discarded. By using transactions, the naive programmer need only enhance its sequential code with invocations of special routines such as ReadDI and WriteDI (which we will call *transactional operations*) to indicate reads or writes for data items, respectively.

The TM algorithm provides a shared representation for each data item and implementations for ReadDI and WriteDI using the *base objects* supported by the system, so that all synchronization problems that may arise during the concurrent execution of transactional operations are addressed. When a transaction executes all its transactional operations it calls a routine called TryCommit in order to commit. TryCommit may return TRUE in which case the transaction commits or FALSE in which case the transaction aborts. We say that a transaction is *commit-pending* at some point in time if it has invoked TryCommit but it has not yet received a response. The implementation details of the TM algorithm are hidden by the naive programmer whose programming task is therefore highly simplified. TM has been given special attention in the last ten years with hundreds or even thousands of papers addressing different problems arising in TM computing (see e.g. [17, 16] for books addressing different aspects of TM computing).

One of the most fundamental problems of TM computing is *safety*. Most TM consistency conditions [3, 15, 16, 21, 12, 6, 7] originate from existing shared memory or database consistency models. However, in contrast to what happens in shared memory models where safety is defined in terms of read and write *operations* in memory, safety in TM computing is defined in terms of *transactions*, each of which may contain more than one read or write operations on data items. Comparing now to database transactions, the main difficulty when defining safety in TM computing is that transactional operations are executed by invoking ReadDI or WriteDI and therefore the execution of a transactional operation has duration and is usually overlapping with the execution of other transactional operations, whereas in database transactions read and write operations are considered to be atomic. For these reasons, existing safety definitions for these two settings (shared memory and database concurrent transactions) cannot be applied verbatim to TM algorithms. Formalizing safety definitions for TM computing requires more effort.

This article presents a comprehensive collection of consistency conditions for TM computing. All conditions are expressed in a uniform way using a formal framework that we present in Section 2. This article can therefore serve as a

survey of *consistency conditions* for TM computing. However, it aspires to be much more than this.

For all known TM consistency conditions we provide a new version, called *live*, in which a transaction $T$ is allowed to read the value of a data item written by another transaction $T'$ that can be live and not yet commit-pending provided that $T'$ will eventually commit (or that $T'$ will commit if $T$ commitss). All TM consistency conditions [3, 15, 16, 21, 6, 7] presented thus far (with TMS1 [12] being the only exception) allowed for transactions to read values that have been written by transactions that either have commited or are commit-pending when $T$ strarts its execution. The live version of a definition is weaker than the later thus resulting to a wider universe of algorithms which should not be excluded from being considered correct. For instance, consider an algorithm which produces executions in which a transaction $T$ is allowed to read a value for a data item $x$ writtten by some transaction $T'$ which has neither committed nor is commit-pending when $T$ starts its execution. However, suppose that the algorithm has been designed in such a way that when this occurs, the algorithm ensures that $T'$ will commit. Then, there is no reason for executions of the algorithm in which this behaviour is met not to be considered correct, i.e. such executions are correct. However, current consistency conditions, as they are formally expressed, exclude such executions from the set of executions they allow. The live version of a consistency condition we present here solves this problem.

A *universal construction* [18] is a mechanism for automatically executing pieces of sequential code in a concurrent environment. A universal construction supports a single operation PERFORM which takes as a parameter a pointer to a routine containing the piece of sequential code to execute concurrently and returns `TRUE` if this is done successfully. Similarly to TM, the sequential code must be enhanced so that accesses to data items are identified by calling routines READDI and WRITEDI. Aparently, universal constructions and TM algorithms are closely related since they both aim at simplifying parallel programming. There are however two basic differences between these two paradigms: (1) the application code must be programmed differently; specifically, in a universal construction, the piece of (the enhanced) sequential code must be included in a routine and a pointer to this routine must then be passed as a parameter to PERFORM, whereas in a TM setting, the code may contain direct invocations of READDI and WRITEDI, and (2) a TM algorithm allows the external environment to choose the action to be performed when a transaction aborts, whereas a call to PERFORM returns only when the simulated code has been successfully applied to the simulated state, i.e. after commit[1].

[1] We remark that the common behaviour for the external environment in a TM setting is to restart an aborted transaction until it eventually commits, so the difference is not essential.

A second benefit of the live versions of the consistency conditions presented here is that some of them work, as are, for universal constructions, by having a call to PERFORM to play the role of a transaction[2]. This contributes towards unifying the two models. It is remarkable that deriving the live version of consistency conditions was not an easy task so we consider their presentation as a significant contribution of this report.

For the derivation of the presented consistency conditions, we do not make any restrictive assumptions, such as that transactional operations are executed atomically or that write operations write distinct values for data items. Making such an assumption is unrealistically restrictive since all TM algorithms produce executions that do not satisfy these assumptions. Thus, a consistency condition that has been expressed making such an assumption cannot be used to characterize such executions, and thus fail to also characterize whether the TM algorithm itself satisfies the condition. We remark that making such assumptions significantly facilitates the task of formally expressing a consistency condition but the formal presentation that results is extremely restricitve since it cannot be used to characterize the correctness of most of the executions produced by any reasonable TM algorithm.

**Related Work.** Among the consistency conditions met in TM computing papers are the following: strict serializability [23], serializability [23], opacity [15, 16], virtual world consistency [21], TMS1 [12] (and TMS2 [12]), and snapshot isolation [2, 10, 25, 6, 7]. Weaker consistency conditions like processor consistency [7], causal serializability [6, 7] and weak consistency [7] have also been considered in the TM context when proving impossibility results.

Strict serializability, as well as serializability, are usually presented in an informal way in TM papers which cite the original paper [23] where these conditions have first appeared in the context of database research. Thus, the differences that exist between database and TM transactions have been neglected in TM research. We present formal definitions of these consistency conditions here. Additional consistency conditions originating from the database research are presented in [3]. To present their formalism, the authors of [3] make the restrictive assumption that transactional operations are atomic. The presentation of most of the other consistency conditions (e.g. opacity [15, 16], virtual world consistency [21], snapshot isolation [2, 10, 25, 6, 7] and weaker variants of them [6, 7]) is based on the assumption that a read for a data item by a transaction $T$ can read a value written by either transaction that has commited or is commit-pending when $T$ starts its execution. Finally, virtual world consistency [21] has been presented in a rather informal way and its definition is based on the assumption that each instance of

[2] This is not achieved by employing the second version of the definitions since the notion of pieces of code that are "commit-pending" is not defined for universal constructions.

WRITEDI writes a distinct value for the data item it accesses (or that the transactional operations are executed atomically).

# 2 TM Model

In this section, we describe a model for transactional memory (TM) computing.

## 2.1 Transactions and histories

Transactional memory (TM) is a parallel programing paradigm which employs transactions to synchronize the execution of threads. A *transaction* is a piece of code which accesses pieces of data, called *data items*. A data item may be accessed by several threads simultaneously in a concurrent environment. A TM algorithm uses base objects to store the state of each data item and ensures synchronization between threads accessing the same data items. A *base object* has a state and supports a set of operations, called *primitives*, to access or update its state. Base objects are usually simple objects that are provided by the hardware.

In order to *read* or *write* a data item, the transaction's code must call specific routines, called READDI and WRITEDI, respectively. The TM algorithm provides implementations for these routines from the base objects. A transaction may commit or abort. If it *commits*, all its updates to data items are realized, whereas if it *aborts*, all its updates are discarded. The TM algorithm provides implementations for two routines, called ABORT and TRYCOMMIT, which are called to try to commit or to abort a transaction, respectively. We refer to all these routines as *transactional operations*. Whenever it is clear from the context, we use the term operation to refer to a transactional operation.

A transactional operation starts its execution when the thread executing it issues an *invocation* for it; the operation completes its execution when the thread executing it returns a *response*. The response for an instance of TRYCOMMIT executed by some transaction $T$ can be either $C_T$ which identifies that $T$ has committed, or $A_T$ which identifies that $T$ has aborted. The response for an instance of ABORT executed by $T$ is always $A_T$. The response for READDI can be either a value or $A_T$; finally, the response for WRITEDI can be either an acknowledgment or $A_T$. We say that a response *res matches* an invocation *inv* in some history $H$, if they are both by the same thread $p$, *res* follows *inv* in $H$, and there is no other response by $p$ between *inv* and *res* in $H$. A transactional operation is *complete*, if there is a response for it; otherwise, the operation is *pending*.

An *event* is either an invocation or a response of a transactional operation. A *history* is a finite sequence of events. Thus, in a history $H$, there are two events for every completed operation $op$, an invocation $inv(op)$ and a matching response

$res(op)$. $H$ contains only the invocation of each pending operation in it. For each data item $x$, we denote by $H \mid x$ the subsequence of $H$ containing the invocations and responses of all transactional operations that access $x$. For each thread $p_i$, we denote by $H \mid p_i$ the subsequence of $H$ containing all invocations and responses of transactional operations executed by $p_i$. For each event $e$ in $H$, we denote by $H \uparrow e$ the longest prefix of $H$ that does not include $e$.

Consider any history $H$. We say that a transaction $T$ (executed by a thread $p_i$) *is in* $H$ or $H$ *contains* $T$, if there are events in $H$ issued by $p_i$ when executing $T$. The *transaction subhistory* of $H$ for $T$, denoted by $H \mid T$, is the subsequence of all events in $H$ issued by $p_i$ when executing $T$. Each transaction $T$ in $H$ for which $H \mid T$ contains at least one invocation of WriteDI is called an *update* transaction. A transaction in $H$ is called *read-only*, if it is not an update transaction.

A history $H$ is said to be *well-formed* if, for each transaction $T$ in $H$, $H \mid T$ is an alternating sequence of invocations and responses, starting with an invocation, such that:

- no events in $H \mid T$ follow $C_T$ or $A_T$;
- if $T'$ is any transaction in $H$ executed by the same thread that executes $T$, either the last event of $H \mid T$ precedes in $H$ the first event of $H \mid T'$ or the last event of $H \mid T'$ precedes in $H$ the first event of $H \mid T$.

From now on we focus on well-formed histories. Let $H$ be any such history. A transaction $T$ is *committed* in $H$, if $H \mid T$ includes $C_T$; a transaction $T$ is *aborted* in $H$, if $H \mid T$ includes $A_T$. A transaction is *completed* in $H$, if it is either committed or aborted, otherwise it is *live*. The *execution interval* of a completed transaction $T$ in $\alpha$ is the subsequence of consecutive steps of $\alpha$ starting with the first step executed by any of the operations invoked by $T$ and ending with the last such step. The *execution interval* of a transaction $T$ that does not complete in $\alpha$ is the suffix of $\alpha$ starting with the first step executed by any of the operations invoked by $T$.

A transaction is *commit-pending* in $H$ if it is live in $H$ and $H \mid T$ includes an invocation to TryCommit for $T$. We denote by $comm(H)$ the subsequence of all events in $H$ issued and received by committed transactions. Two histories $H$ and $H'$ are said to be *equivalent* if each thread $p$ executed the same transactions, in the same order, in $H$ and $H'$, and for every transaction $T$ in $H$, $H \mid T = H' \mid T$, i.e. for each transaction the same transactional operations are invoked and each of these operations has the same response in both histories.

Consider any history $H$. We denote by $Complete(H)$ a set of histories that extend $H$. Specifically, a history $H'$ is in $Complete(H)$ if and only if, all of the following hold:

1. $H'$ is well-formed, $H$ is a prefix of $H'$, and $H$ and $H'$ contain the same set of transactions;

2. for every live transaction[3] $T$ in $H$:

   (a) if $H \mid T$ ends with an invocation of TryCommit, $H'$ contains either $C_T$ or $A_T$;

   (b) if $H \mid T$ ends with an invocation other than TryCommit, $H'$ contains $A_T$;

   (c) if $H \mid T$ ends with a response, $H'$ contains $\text{Abort}_T$ and $A_T$.

Roughly speaking, each history in $Complete(H)$ is an extension of $H$ where some of the commit-pending transactions in $H$ appear as committed and all other live transactions appear as aborted.

A *configuration* is a vector consisting of the state of each thread and the state of each base object. In an *initial configuration*, threads and base objects are in initial states. A *step* of a thread consists of applying a single primitive on some base object, the response to that primitive, and zero or more local operations that are performed after the access and which may cause the internal state of the thread to change. As a step, we also consider the invocation of a transactional operation or the response to such an invocation; notice that a step of this kind does not change the state of any base object. Each step is executed atomically. An *execution* $\alpha$ is a sequence of steps. An execution is *legal* starting from a configuration $C$ if the sequence of steps performed by each thread follows the algorithm for that thread (starting from its state in $C$) and, for each base object, the responses to the operations performed on the object are in accordance with its specification (and the state of the object at configuration $C$). Given an execution $\alpha$, the history of $\alpha$, denoted by $H_\alpha$, is the subsequence of $\alpha$ consisting of just the invocations and the responses of transactional operations.

## 2.2 Relations and Partial Orders

Consider a well-formed history $H$. We define a partial order, called *real time order* and denoted $<_H$, on the set of *transactions* in $H$ as follows:

- for any two transactions $T_1$ and $T_2$ in $H$, if $T_1$ is completed in $H$ and the last event of $H \mid T_1$ precedes the first event of $H \mid T_2$ in $H$, then $T_1 <_H T_2$.

Transactions $T_1$ and $T_2$ are *concurrent* in $H$, if neither $T_1 <_H T_2$ nor $T_2 <_H T_1$. $H$ is *sequential* if no two transactions in $H$ are concurrent.

[3] We remark that the order in which the live transactions of $H$ are inspected to form $H'$ is immaterial, i.e. all histories that result from any possible such order are added in $Complete(H)$.

We also define a partial order, called *operational real-time* order and denoted by $<_H^{op}$, on the set of *transactional operations* in $H$ as follows:

- for any two transactional operations $op_1$ and $op_2$ in $H$, if $H$ contains a response for $op_1$ which precedes the invocation of $op_2$, then $op_1 <_H^{op} op_2$.

Operations $op_1$ and $op_2$ are *concurrent* in $H$, if neither $op_1 <_H^{op} op_2$ nor $op_2 <_H^{op} op_1$. $H$ is *operational-wise sequential* if no two operations in $H$ are concurrent.

Let $S_{op}$ be an operational-wise sequential history equivalent to $H$. We say that $S_{op}$ *respects* some relation $<$ on the set of *transactions* in $H$ if the following holds: for any two transactions $T_1$ and $T_2$ in $S$, if $T_1 < T_2$, then $T_1 <_S T_2$. We say that $S_{op}$ respects some relation $<^{op}$ on the set of *transactional operations* in $H$ if the following holds: for any two operations $op_1$ and $op_2$ in $S_{op}$, if $op_1 <^{op} op_2$, then $op_1 <_{S_{op}}^{op} op_2$. Notice that a partial order is a relation, so these definitions hold for partial orders as well.

Consider any operational-wise sequential history $S_{op}$ that is equivalent to $H$ and respects $<_H$. We define a binary relation (with respect to $S_{op}$), called *reads-from* and denoted by $<_H^r$, between *transactions* in $H$ such that, for any two transactions $T_1, T_2$ in $H$, $T_1 <_H^r T_2$ only if:

- $T_2$ executes a ReadDI operation $op$ that reads some data item $x$ and returns a value $v$ for it,
- $T_1$ is the transaction in $S_{op}$ which executes the last WriteDI operation that writes $v$ for $x$ and precedes $op$.

Notice that each operational-wise sequential history $S_{op}$ that is equivalent to $H$, induces a *reads-from* relation. We denote by $\mathcal{R}_{\mathcal{H}}$ the set of all reads-from relations that can be induced for $H$.

For each $<_H^r$ in $\mathcal{R}_{\mathcal{H}}$, we define the *causal* relation for $<_H^r$ on transactions in $H$ to be the transitive closure of $\bigcup_i \left(<_{H|p_i}\right) \cup <_H^r$. We define $C_{\mathcal{H}}$ to be the set of all causal relations in $H$.

## 2.3 Legality

A set $\mathcal{S}$ of sequences is prefix-closed if, whenever $H$ is in $\mathcal{S}$, every prefix of $H$ is also in $\mathcal{S}$. A history $H$ is a *single data-item* history for some data item $x$, if $H \mid x = H$. A *sequential specification* for a data item is a prefix-closed set of single data-item sequential histories for that data item. A sequential history $H$ is *legal* if, for each data item $x$, $H \mid x$ belongs to the sequential specification for $x$.

Consider a sequential history $S$ and a transaction $T$ in $S$. We say that $T$ is *legal* in $S$, if for every invocation *inv* of ReadDI on each data item $x$ that $T$ performs whose response is not $A_T$ the following hold:

1. if there is an invocation of WRITEDI for $x$ by $T$ that precedes $inv$ in $S$ then $v$ is the argument of the last such invocation,

2. otherwise, if there are no committed transactions preceding $T$ in $S$ which invoke WRITEDI for $x$, then $v$ is the initial value for $x$,

3. otherwise, $v$ is the argument of the last invocation of WRITEDI of any committed transaction that precedes $T$ in $S$.

A complete sequential history $S$ is legal if every transaction in $S$ is legal.

# 3 TM Consistency

## 3.1 Strict Serializability

Strict serializability was first introduced in [23] as a (strong) consistency condition for executions of concurrent transactions in database systems. In TM computing, it can be expressed in several different flavors, two of which are discussed below. We start with *live strict serializability* (or *$\ell$-strict serializability* for short).

**Definition 1** (Live Strict Serializability or L-Strict Serializability)**.** *We say that an execution $\alpha$ is* $\ell$-strictly serializable *if it is possible to do all of the following:*

- *If $A$ is the set of all complete transactions in $\alpha$ that are not aborted, for each transaction $T \in A$, to insert a serialization point $*_T$ somewhere between $T$'s first invocation of a transactional operation and $T$'s last response for a transactional operation in $\alpha$.*

- *To choose a subset $B$ of the live transactions in $\alpha$ and, for each transaction $T \in B$, insert a serialization point $*_T$ somewhere after $T$'s first invocation of a transactional operation in $\alpha$.*

*These serialization points should be inserted, so that, in the sequential execution $\sigma$ that we get by serially executing each transaction $T \in A \cup B$ at the point that its serialization point has been inserted, the following hold:*

- *for each transaction $T \in A$, the same transactional operations, as in $\alpha$, are invoked by $T$ in $\sigma$ and the response of each such operation in $\sigma$ is the same as that in $\alpha$, and*

- *for each transaction $T \in B$, a prefix of the operations[4] invoked by $T$ in $\sigma$ are the same as the sequence of operations invoked by $T$ in $\alpha$ and the response of each such operation in $\sigma$ is the same as that in $\alpha$.*

[4]Notice that since $\sigma$ is a sequential execution, each transaction $T \in B$ commits in $\sigma$.

We continue to provide a stronger version of $\ell$-strict serializability in Definition 2 called *commit-oriented strict serializability* (or *c-strict serializability* for short) which is based on the definition of *Complete*.

**Definition 2** (C-Strict Serializability)**.** *A history $H$ is* c-strictly serializable*, if there exist a history $H' \in Complete(H)$ and a history $S$ equivalent to $comm(H')$ such that:*

- *$S$ is a legal sequential history, and*
- *$S$ respects $<_{comm(H')}$.*

We remark that Definition 1 provides a weaker version of strict serializability than Definition 2, since it allows a transaction to read a value for a data item written by another transaction that is not committed or commit-pending in $H$. This is allowed only if eventually, all complete transactions that are not aborted, and some of those that are still live can be "serialized" within their execution intervals. For instance, let's consider the history $H$ and its prefix $H_1$ both shown on Figure 1. $H$ is both $\ell$-strictly serializable and c-strictly serializable, whereas $H_1$ is just $\ell$-strictly serializable. Notice that since $\ell$-strict serializability is weaker than c-strict serializability, the universe of algorithms that are $\ell$-strictly serializable is larger than that of the algorithms that are c-strictly serializable.

We remark that c-strict serializability is not a prefix-closed property. On the contrary, $\ell$-strict serializability is a prefix-closed property. We remark that prefix-closure can be imposed to c-strict serializability in an explicit way, i.e. by directly stating in Definition 2 that each prefix $H_p$ of $H$ must also satisfy the conditions imposed by the definition (as it is done in Definition 5 in Section 3.3). However, this would make Definition 2 even stronger, and therefore the resulted universe of c-strictly serializable TM algorithms even smaller.

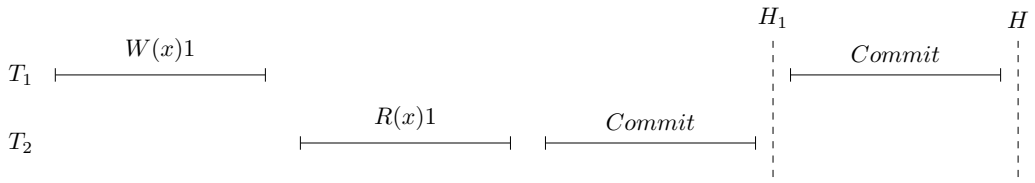


Figure 1: Example showing that strict serializability is not a prefix-closed property.

## 3.2 Serializability

As with strict serializability, serializability was first introduced in [23] as a consistency condition for executions of concurrent transactions in database systems.

Below we discuss two different flavors of serializability in a way similar to that for strict serializability.

**Definition 3** (L-Serializability)**.** *We say that an execution $\alpha$ is* $\ell$-serializable *if it is possible to do all of the following:*

- *If $A$ is the set of all complete transactions in $\alpha$ that are not aborted, for each transaction $T \in A$, to insert a serialization point $*_T$ in $\alpha$.*
- *To choose a subset $B$ of the live transactions in $\alpha$ and, for each transaction $T \in B$, insert a serialization point $*_T$ in $\alpha$.*

*These serialization points should be inserted, so that, in the sequential execution $\sigma$ that we get by serially executing each transaction $T \in A \cup B$ at the point that its serialization point has been inserted, the following hold:*

- *for each transaction $T \in A$, the same transactional operations, as in $\alpha$, are invoked by $T$ in $\sigma$ and the response of each such operation in $\sigma$ is the same as that in $\alpha$, and*
- *for each transaction $T \in B$, a prefix of the operations invoked by $T$ in $\sigma$ are the same as the sequence of operations invoked by $T$ in $\alpha$ and the response of each such operation in $\sigma$ is the same as that in $\alpha$.*

We continue to provide a stronger version of serializability in Definition 4, called *commit-oriented serializability* (or *c-serializability* for short), which is based on the definition of $Complete$.

**Definition 4** (C-Serializability)**.** *A history $H$ is* c-serializable*, if there exist a history $H' \in Complete(H)$ and a history $S$ equivalent to $comm(H')$ such that:*

- *$S$ is a legal sequential history.*

Notice that $S$ in Definition 4 respects the program order of transactional operations executed by the same process in $H$. This is implied by the definition of equivalent histories.

We remark that, similarly to the corresponding definitions of strict serializability, Definition 3 provides a weaker version of serializability than Definition 4.

The difference between serializability and strict serializability is that strict serializability additionally ensures that the real-time order of transactions is respected by the sequential history defined by the serialization points. Thus, every history/execution that is strict serializable is also serializable but not vice versa.

It is worth-pointing out that $\ell$-serializability and c-serializability are not prefix-closed properties. This is so, since it is easy to design a history $H$ which is $\ell$-serializable (as well as c-serializable) in which a committed transaction $T$ (executed by some process $p$) reads for some data item $x$ a value $v$ written by some

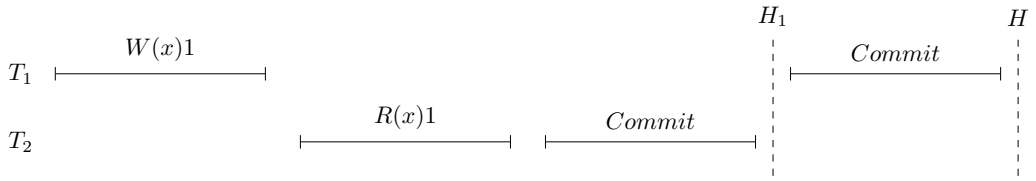


Figure 2: Example showing that serializability is not a prefix-closed property.

other committed (or commit-pending) transaction $T'$ such that $T'$ is executed by some process $p' \neq p$ in $H$ and $T'$'s execution has started after $T$ has been completed. Aparently, the prefix of $H$ up until $C_T$ is neither $\ell$-serializable, nor c-serializable.

We remark that prefix-closure can be imposed to $\ell$-serializability (as well as to c-serializability) in an explicit way, as discussed for c-strict serializability above. It is not clear if the versions that would then result will be weaker than the corresponding versions of strict serializability. Imposing prefix closure to the consistency conditions presented in Sections 3.4.1-3.5 may be too restrictive as well. Thus, we present the non-prefix-closed versions of them given that it is straightforward to derive their prefix-closed versions, in an explicit way.

Several impossibility results [4, 8, 13] and lower bounds [4] in TM computing have been proved for strict serializability or serializability. Most TM algorithms in the literature (see e.g. [9, 28, 11, 26] for some examples) satisfy some form of serializability.

## 3.3 Opacity

Opacity was first introduced in [15]. In [16], a prefix-closed version of it was formally stated. Here, we will present the later version which we will call c-opacity (to be coherent with definitions in previous sections).

**Definition 5** (C-Opacity [16])**.** *A history $H$ is c-opaque if, for each prefix $H_p$ of $H$, there exists a sequential history $S_p$ equivalent to some history $H'_p \in Complete(H_p)$ such that:*

- *$S_p$ respects $<_{H'_p}$, and*
- *every transaction $T_i$ in $S_p$ is legal in $S_p$.*

C-opacity is stronger than c-strict serializability. Figure 3 shows an example of a history that is not c-opaque but is c-strictly serializable. This history is not c-opaque because it violates the last condition of Definition 5; specifically, transaction $T_2$ cannot be legal.


Figure 3: A strict serializable history which is not opaque.

Strict serializability (independently of the variant we consider) doesn't impose any restrictions on non-committed (or not commit-pending) transactions, whereas c-opacity requires that all reads of each transaction $T$ (independently of whether the transaction is committed, aborted or live in the considered history) read values written by previously commited transactions (or by $T$ itself). This additional property is required in order to avoid undesired situations where a transaction may cause an exception or enter into an infinite loop after reading a value for a data item written by a live transaction that may eventually abort.

It is remarkable that the first of these undesired situations (i.e. the production of an exception or an error code) can be avoided even by TM system that ensure only strict serializability if we make the following simple assumptions in our model. An exception (or an error code) that has been resulted by the execution of a transactional operation $op$ is considered as a response for $op$. A transaction that has experienced an exception or has received an error code as a response, to one of its operations, is considered to be completed (but not aborted). Then, a ($\ell$- or c-) strictly serializable TM implementation will never produce such exceptions (or error codes). Notice that the second undesirable situation, namely having some transaction enter an infinite loop, will not appear in TM systems that ensure standard progress properties, like lock-freedom, starvation-freedom, etc.

We continue to present live opacity ($\ell$-opacity). Consider any history $H$ and a transaction $T$ in $H$. An instance $op$ of ReadDI for some data item $x$ executed by $T$ is *global* if $T$ has not invoked WriteDI on $x$ in $H$ before invoking $op$. Let $T|read$ be the longest subsequence of $H|T$ consisting only of the invocations of ReadDI (and their responses if they exist), excluding the pair of the last such invocation and its response if the response is $A_T$. Let $T|read_g$ be the subsequence of $T|read$ consisting only of the invocations of the *global* instances of ReadDI (and their responses if they exist). Let $\lambda$ be the empty sequence. We denote by $T_r$ a transaction that invokes the same transactional operations as those invoked in $T|read \cdot \text{TryCommit}_{T_r}$ if $T|read \neq \lambda$, or $T_r = \lambda$ otherwise. Similarly, denote by $T_{gr}$ a transaction that invokes the same transactional operations as those invoked $T|read_g \cdot commit_{T_{gr}}$ if $T|read_g \neq \lambda$, or $T_{gr} = \lambda$ otherwise. For each ReadDI operation $op$ on any data item $x$ that is in $T_r$ but not in $T_{gr}$, we say that the response for $op$ (if it exists) is *legal*, if it is the value written by the last WriteDI for $x$

performed by $T$ before the invocation of $op$.

**Definition 6** (L-Opacity)**.** *We say that an execution $\alpha$ is $\ell$-*opaque *if there exists some sequential execution $\sigma$ which justifies that $\alpha$ is strictly serializable, and all of the following hold:*

1. *We can extend the history $H_\sigma$ of $\sigma$ to get a* sequential *history $H'_\sigma$ such that:*
   - *for each transaction $T$ in $\alpha$ that is not in $\sigma$, $H'_\sigma$ contains $T_{gr}$*
   - *if $<$ is the partial order which is induced by the real time order $<_{H_\alpha}$ in such a way that for each transaction $T$ in $\alpha$ that is not in $\sigma$, we replace each instance of $T$ in the set of pairs of $<_{H_\alpha}$ with transaction $T_{gr}$, then $H'_\sigma$ respects $<$*
   - *$H'_\sigma$ is legal*
2. *for each transaction $T$ in $\alpha$ that is not in $\sigma$, and for each transactional operation $op$ in $T_r$ that is not in $T_{gr}$, the response for $op$ is legal.*

We remark that most TM algorithms presented in the literature are opaque.

## 3.4 Causality-Related Consistency Conditions

### 3.4.1 Causal Consistency

Causal consistency was informally introduced as a shared memory consistency condition in [20], and it was formally defined in [1]. As in the previous sections, we provide two formal definitions of causal consistency for TM computing using the framework of Section 2.

**Definition 7** (L-Causal Consistency)**.** *Consider an execution $\alpha$ and let $A$ be the set of all complete transactions in $\alpha$ that are not aborted. We say that $\alpha$ is $\ell$-*causally-consistent *if there exists a subset $B$ of live transactions in $\alpha$ and a causal relation $<^c$ in $C_{H'_\alpha}$ where $H'_\alpha$ is the subsequence of $H_\alpha$ containing just the events of transactions in $A \cup B$, such that,* for each process $p_i$*, it is possible to do the following:*
*For each transaction $T \in A \cup B$, to insert a serialization point $*_T$ in $\alpha$ so that, if $\sigma_i$ is the sequential execution that we get by serially executing each transaction $T \in A \cup B$ at the point that its serialization point has been inserted, then the following hold:*

- *$H_{\sigma_i}$ respects $<^c$,*

- *for each transaction $T \in A$, the same transactional operations, as in $\alpha$, are invoked by $T$ in $\sigma_i$ and the response of each such operation in $\sigma_i$ is the same as that in $\alpha$, and*
- *for each transaction $T \in B$, a prefix of the operations invoked by $T$ in $\sigma_i$ are the same as the sequence of operations invoked by $T$ in $\alpha$, the response of each such operation in $\sigma_i$ is the same as that in $\alpha$.*

**Definition 8** (C-Causal Consistency)**.** *A history $H$ is* c-causally consistent *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $C_{comm(H')}$ such that, for each process $p_i$, there exist a sequential history $S_i$ such that:*

- *$S_i$ is equivalent to $comm(H')$,*
- *$S_i$ respects the causality order $<^c$, and*
- *every transaction executed by $p_i$ in $S_i$ is legal.*

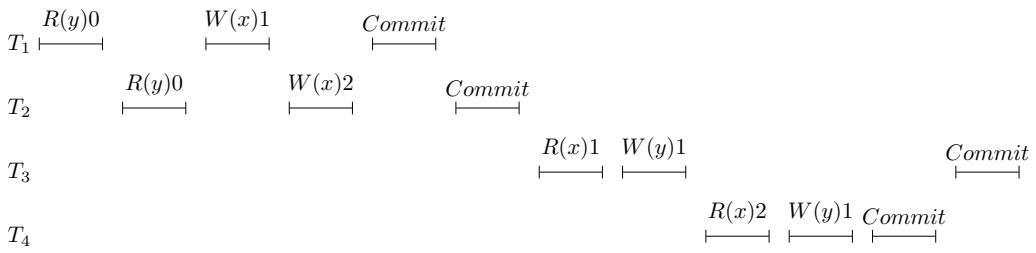


Figure 4: A causally consistent history which is not serializable.

L-causal consistency and c-causal consistency are weaker properties than $\ell$-serializability and c-serializability. For instance, Figure 4 shows an example of a history which is ($\ell$- and c-) causally consistent but not ($\ell$- or c-) serializable. In this history both transactions $T_1$ and $T_2$ should be serialized before transactions $T_3$ and $T_4$, because both $T_1$ and $T_2$ read 0 from data item $y$ which is written by $T_3$ and $T_4$. Regardless of how the serialization points for $T_1$ and $T_2$ are ordered, both $T_3$ and $T_4$ should read the same value for data item $x$. Thus, this history is not serializable. However, it is causally consistent because processes running $T_3$ and $T_4$ may see writes executed by processes running $T_1$ and $T_2$ in a different order.

### 3.4.2 Causal Serializability

Causal serializability was introduced in [24] as a consistency condition which is stronger than causal consistency but weaker than serializability. Informally, in

addition to the constraints imposed by causal consistency, the following constraint must also be satisfied: all transactions that update the same data item must be perceived in the same order by all processes.

**Definition 9** (L-Causal Serializability)**.** *Consider an execution $\alpha$ and let $A$ be the set of all complete transactions in $\alpha$ that are not aborted. We say that $\alpha$ is* $\ell$-causally serializable *if there exists a subset $B$ of live transactions in $\alpha$ and a causal relation $<^c$ in $C_{H'_\alpha}$ where $H'_\alpha$ is the subsequence of $H_\alpha$ containing just the events of transactions in $A \cup B$, such that,* for each process *$p_i$, it is possible to do the following:*
*For each transaction $T \in A \cup B$, to insert a serialization point $*_T$ in $\alpha$ so that, if $\sigma_i$ is the sequential execution that we get by serially executing each transaction $T \in A \cup B$ at the point that its serialization point has been inserted, then the following hold:*

- *$H_{\sigma_i}$ respects $<^c$,*
- *for each transaction $T \in A$, the same transactional operations, as in $\alpha$, are invoked by $T$ in $\sigma_i$ and the response of each such operation in $\sigma_i$ is the same as that in $\alpha$,*
- *for each transaction $T \in B$, a prefix of the operations invoked by $T$ in $\sigma_i$ are the same as the sequence of operations invoked by $T$ in $\alpha$, the response of each such operation in $\sigma_i$ is the same as that in $\alpha$.*
- *for each pair of transactions $T_1, T_2 \in A \cup B$ that write to the same data item, if $T_1 <_{H_{\sigma_i}} T_2$, then for each $j \in \{1, \ldots, n\}$, it holds that $T_1 <_{H_{\sigma_j}} T_2$.*

**Definition 10** (C-Causal Serializability)**.** *A history $H$ is* c-causally serializable *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $C_{comm(H')}$ such that, for each process $p_i$, there exist a sequential history $S_i$ for which the following hold:*

- *$S_i$ is equivalent to $comm(H')$,*
- *$S_i$ respects the causality order $<^c$,*
- *every transaction executed by $p_i$ in $S_i$ is legal, and*
- *for each pair of transactions $T_1$ and $T_2$ in $comm(H')$ that write to the same data item, if $T_1 <_{S_i} T_2$, then for each $j \in \{1, \ldots, n\}$, it holds that $T_1 <_{S_j} T_2$.*

Obviously, every causally serializable history satisfies the properties of causal consistency, but the oposite is not true. For instance, the history shown in Figure 4

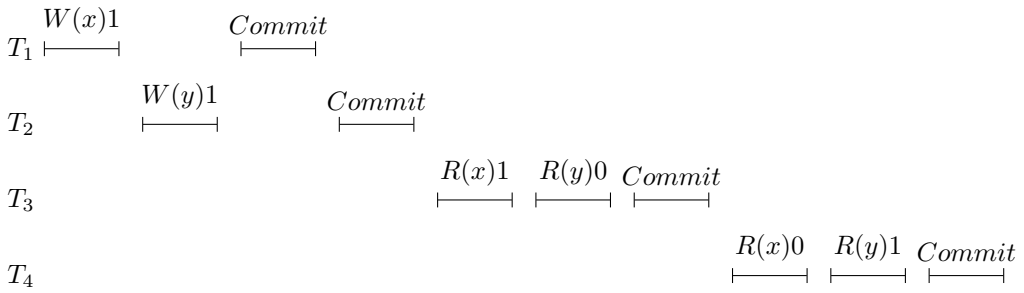


Figure 5: A causally serializable history which is not serializable.

is not causally serializable, since processes executing transactions $T_3$ and $T_4$ do not see writes from $T_1$ and $T_2$ to data item $x$ in the same order.

Figure 5 shows an example of a history which is causally serializable but not serializable. Here, if transaction $T_1$ is serialized before $T_2$ (the opposite case is symmetrical), then it is not possible to serialize transaction $T_4$. However, by definition of causal serializability, sequential histories constructed for processes $p_3$ and $p_4$ may include transactions $T_1$ and $T_2$ in different orders.

In the context of TM research, causal consistency, as well as causal serializability, are interesting in the context of proving impossibility results [6, 7] and lower bounds. We remark that when proving such results, considering a weak consistency condition makes the result stronger. It is therefore an interesting open problem to see whether some of the TM impossibility results (e.g. [4, 8, 13]) that have been proved assuming some strong consistency condition, like opacity, strict serializability or serializability, can be extended to hold for weaker consistency conditions like those formulated in this or later sections. For instance in this avenue, the impossibility result proved in [14] assuming serializability is extended in [6, 7] to hold for a much weaker consistency condition.

### 3.4.3 Virtual World Consistency

Virtual World Consistency (VWC) was defined in [21] as a family of consistency conditions. Informally, VWC ensures serializability or strict serializability for the committed (and some of the commit-pending) transactions but a weaker condition than that imposed by opacity for the rest of the transactions.

For each transaction $T$ in history $H$ and each causal relation $<^c_H$ in $C_H$, we define the *causal past* of $T$ denoted by $past_T(H, <^c_H)$ as the subsequence of all events produced either by transaction $T$ in $H$ itself or by any transaction $T_i$ in $H$ such that $T_i <^c_H T$.

**Definition 11** (C-Virtual World Consistency)**.** *A history $H$ is* c-virtual world consistent *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $C_{H'}$*

*such that:*

- *there exists a legal sequential history $S$ which is equivalent to comm($H'$), and*
- *for each transaction $T_i$ in $H'$ that is not in $S$, there exists a legal sequential history $S_i$ which is equivalent to $past_{T_i}(H', <^c)$ and respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H', <^c)$.*

**Definition 12** (C-Strong Virtual World Consistency)**.** *A history $H$ is* c-strong virtual world consistent *if there exists a history $H' \in Complete(H)$ and a causal relation $<^c$ in $C_{H'}$ such that:*

- *there exists a legal sequential history $S$ which is equivalent to comm($H'$) and respects the real-time order of $H'$, and*
- *for each non-committed transaction $T_i$ in $H'$, there exists a legal sequential history $S_i$ which is equivalent to $past_{T_i}(H', <^c_{H'})$ and respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H', <^c)$.*

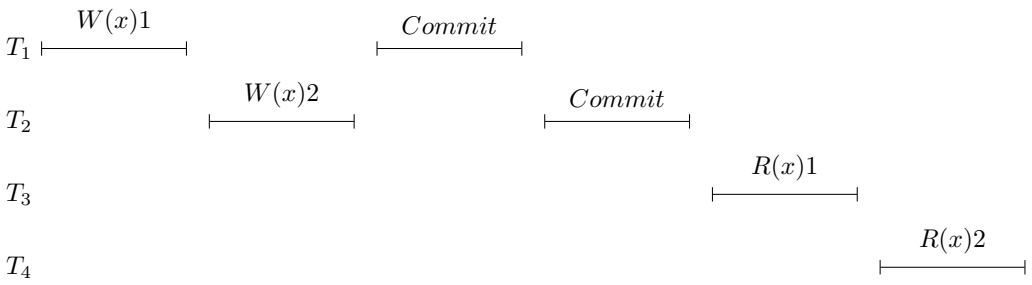


Figure 6: A virtual world consistent history which is not opaque.

Clearly, virtual world consistency is a stronger consistency condition than serializability. Similarly, strong virtual world consistency is stronger than strict serializability. Still, strong virtual world consistency (and therefore also virtual world consistency) is weaker than opacity. The history shown in Figure 6 is strong virtual world consistent but not opaque: regardless of the order of the serialization points of transactions $T_1$ and $T_2$, it is not possible to derive a sequential history where both transaction $T_3$ and $T_4$ are legal.

The history shown in Figure 7 is a slightly modified version of the history shown in Figure 6. This history is virtual world consistent but not strong virtual world consistent. In this history, transactions $T_1$ and $T_2$ are not concurrent, and

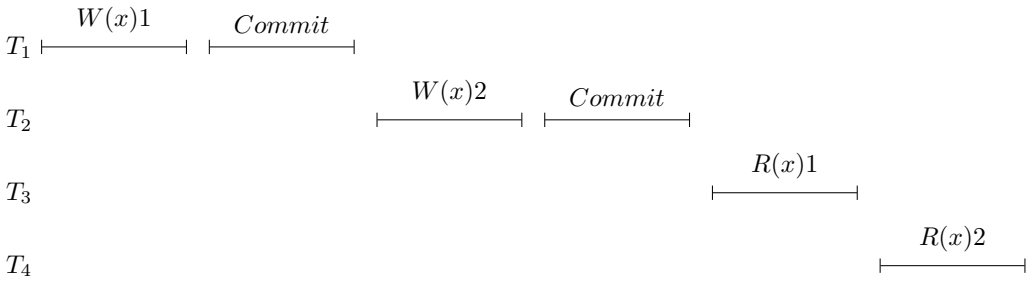


Figure 7: A virtual world consistent history which is not strong virtual world consistent.

since strong virtual world consistency respects the real-time order of transactions, there is only one way that the serialization points of these two transactions can be ordered in any equivalent sequential history.

We continue to present the live versions of virtual world consistency and strong virtual world consistency.

**Definition 13** (L-Virtual World Consistency and L-Strong Virtual World Consistency)**.** *We say that an execution $\alpha$ is* $\ell$-virtual world consistent *($\ell$-*strong virtual world consistent*) if there exists some sequential execution $\sigma$ which justifies that $\alpha$ is serializable (strictly serializable, respectively), and the following holds:*

1. *for each transaction $T_i$ in $\alpha$ that is not in $\sigma$ there exists a legal sequential history $S_i$ which is equivalent to $past_{T_i}(H', <^c)$ and respects the restriction of $<^c$ to those pairs whose components are transactions in $past_{T_i}(H', <^c)$.*

Strict consistency conditions such as opacity ensure the safe execution of non-commited transactions by imposing on them the same safety demands as those that committed transactions are required to obey. This has been criticized in [21] to result in TM algorithms that produce histories in which live transactions are forced to abort in order to preserve the safety of other transactions that are deemed to also abort. Virtual world consistency relaxes the correctness criteria used for non-committed transactions in order to avoid such scenarios when possible, and by consequence, allow for more live transactions to commit, than a TM algorithm that implements a stricter criterion would.

## 3.5 Snapshot Isolation

Snapshot isolation was originally introduced as a safety property in the database world [5, 22]. Snapshot isolation is an appealing property for TM computing [2, 10, 25] since it provides the potential to increase throughput for workloads with

long transactions [25]. The first formal definitions for TM snapshot isolation was given in [6, 7].

Consider a history $H$ and let $T$ be a transaction that either commits or is commit-pending in $H$. Recall that we have already defined the sequences $T|read$, $T|read_g$, as well as transactions $T_r$ and $T_{gr}$ in Section 3.3. Let $T|other$ be the subsequence $H|T - T|read_g$, i.e. $T|other$ consists of all invocations performed by $T$ (and any matching responses) other than those comprising $T|read_g$. Let $T_o$ be a transaction that invokes the same transactional operations (and in the same order) as those invoked in $T|other \cdot commit_{T_o}$ if $T|other \neq \lambda$, and $T_o = \lambda$ otherwise.

**Definition 14** (C-Snapshot isolation [7])**.** *An execution $\alpha$ satisfies* c-snapshot isolation*, if there exists a set $D$ consisting of all committed and some of the commit-pending transactions in $\alpha$ for which the following holds: for each transaction $T \in D$, it is possible to insert (in $\alpha$) a global read point $*_{T,gr}$ and a write point $*_{T,w}$, so that if $\delta_\alpha$ is the sequence defined by these serialization points, the following hold:*

1. *$*_{T,gr}$ precedes $*_{T,w}$ in $\delta_\alpha$,*
2. *both $*_{T,gr}$ and $*_{T,w}$ are inserted within the execution interval of $T$,*
3. *if $H_{\delta_\alpha}$ is the history we get by replacing each $*_{T,gr}$ with $T_{gr}$ and each $*_{T,w}$ with $T_o$ in $\delta_\alpha$, then $H_{\delta_\alpha}$ is legal.*

We finally present live snapshot isolation. Consider a legal execution $\alpha$ and let $C(\alpha)$ be the set of all legal executions such that each execution $\alpha' \in C(\alpha)$ is an extension of $\alpha$ such that the same transactions are executed in $\alpha$ and $\alpha'$ and no transaction is live in $\alpha'$.

**Definition 15** (L-Snapshot Isolation)**.** *Consider an execution $\alpha$. We say that $\alpha$ satisfies $\ell$-*snapshot isolation*, if there exists an execution $\alpha' \in C(\alpha)$ for which the following holds: if $A$ is the set of transactions that commit in $\alpha'$ then for each transaction $T \in A$, it is possible to insert a global read point $*_{T,gr}$ and a write point $*_{T,w}$, so that:*

1. *both $*_{T,gr}$ and $*_{T,w}$ are inserted within the execution interval of $T$ in $\alpha$*
2. *$*_{T,gr}$ precedes $*_{T,w}$, and*
3. *if $\sigma$ is the sequential execution that we get when for each transaction $T \in A$, we serially execute transactions $T_{gr}$ and $T_o$ at the points that $*_{T,gr}$ and $*_{T,w}$ have been inserted, respectively, then for each transaction $T \in A$, the response of each transactional operation invoked by $T_{gr}$ and $T_o$ in $\sigma$ is the same as that of the corresponding transactional operation in $T|read_g$ and $T|other$ (as defined based on $\alpha'$), respectively.*

# 4 Acknowledgements

This work has been supported by the European Commission under the 7th Framework Program through the TransForm (FP7-MC-ITN-238639) project and by the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project.

We would like to thank Victor Bushkov for his valuable comments in a preliminary version of this paper and Eleftherios Kosmas for several useful discussions that motivated this work. Many thanks also to Hagit Attiya for her comments on a previous version of this article.

# References


[1] M. Ahamad, G. Neiger, J. E. Burns, P. Kohli, and P. W. Hutto. Causal memory: definitions, implementation, and programming. *Distributed Computing*, 9(1):37–49, 1995.

[2] M. S. Ardekani, P. Sutra, and M. Shapiro. The impossibility of ensuring snapshot isolation in genuine replicated stms. In *The 3rd edition of the Workshop on the Theory of Transactional Memory*, WTTM2011, 2011.

[3] H. Attiya and S. Hans. Transactions are Back-but How Different They Are? In *TRANSACT*, feb 2012.

[4] H. Attiya, E. Hillel, and A. Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, SPAA '09, pages 69–78, New York, NY, USA, 2009. ACM.

[5] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. *SIGMOD Rec.*, 24(2):1–10, may 1995.

[6] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. Snapshot isolation does not scale either. Technical Report TR-437, Foundation of Research and Technology – Hellas (FORTH), 2013.

[7] V. Bushkov, D. Dziuma, P. Fatourou, and R. Guerraoui. The pcl theorem - transactions cannot be parallel, consistent and live. In *Proceedings of the 4th Annual ACM symposium on Parallelism in Algorithms and Architectures (SPAA 14)*. ACM Press, jul 2014.

[8] V. Bushkov, R. Guerraoui, and M. Kapałka. On the liveness of transactional memory. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 9–18, New York, NY, USA, 2012. ACM.

[9] L. Dalessandro, M. F. Spear, and M. L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[10] R. J. Dias, J. Seco, and J. M. Lourenço. Snapshot isolation anomalies detection in software transactional memory. In *Proceedings of InForum 2010*, 2010.

[11] D. Dice and N. Shavit. What really makes transactions faster? In *Proc. of the 1st TRANSACT 2006 workshop*, 2006. Electronic, no. page numbers.

[12] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Towards formally specifying and verifying transactional memory. *Formal Aspects of Computing*, pages 1–31, mar 2012.

[13] F. Ellen, P. Fatourou, E. Kosmas, A. Milani, and C. Travers. Universal constructions that ensure disjoint-access parallelism and wait-freedom. In *Proceedings of the 2012 ACM symposium on Principles of distributed computing*, PODC '12, pages 115–124, New York, NY, USA, 2012. ACM.

[14] R. Guerraoui and M. Kapalka. On obstruction-free transactions. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '08, pages 304–313, New York, NY, USA, 2008. ACM.

[15] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[16] R. Guerraoui and M. Kapalka. *Principles of Transactional Memory (Synthesis Lectures on Distributed Computing Theory)*. Morgan and Claypool Publishers, 2010.

[17] T. Harris, J. Larus, and R. Rajwar. *Transactional Memory, 2Nd Edition*. Morgan and Claypool Publishers, 2nd edition, 2010.

[18] M. Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, Jan. 1991.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

[20] P. Hutto and M. Ahamad. Slow memory: weakening consistency to enhance concurrency in distributed shared memories. In *Distributed Computing Systems, 1990. Proceedings., 10th International Conference on*, pages 302–309, 1990.

[21] D. Imbs and M. Raynal. Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations). *Theoretical Computer Science*, 444(0):113 – 127, 2012. Structural Information and Communication Complexity ï£¡ SIROCCO 2009.

[22] R. Normann and L. T. Østby. A theoretical study of 'snapshot isolation'. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 44–49, New York, NY, USA, 2010. ACM.

[23] C. H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, oct 1979.

[24] M. Raynal, G. Thia-Kime, and M. Ahamad. From serializable to causal transactions for collaborative applications. In *EUROMICRO 97. New Frontiers of Information Technology., Proceedings of the 23rd EUROMICRO Conference*, pages 314–321, 1997.

[25] T. Riegel, C. Fetzer, and P. Felber. Snapshot isolation for software transactional memory. In *In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing*, TRANSACT'06, 2006.

[26] T. Riegel, C. Fetzer, and P. Felber. Time-based transactional memory with scalable time bases. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 221–228, New York, NY, USA, 2007. ACM.

[27] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[28] M. F. Spear, M. M. Michael, and C. von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.