Survey on consistency conditions

Dmytro Dziuma
FORTH-ICS
ddziuma@ics.forth.gr

Panagiota Fatourou
FORTH-ICS a& Univ. of Crete
faturu@ics.forth.gr

Eleni Kanellou
FORTH-ICS & Univ. of Rennes 1
kanellou@ics.forth.gr

Amirtahmasebi Kasra
FORTH-ICS
akasra@ics.forth.gr

FORTH-ICS TR 439, DECEMBER 2013

Abstract

This survey provides formal definitions of consistency conditions for Transactional Memory. For reasons of completeness, a big collection of consistency conditions from other relevant fields, such as shared-memory consistency and consistency for database systems, are also studied.

The report attempts to bridge the gaps that exist between different formulations of consistency conditions. To do this, it establishes a common framework for expressing consistency and presents a comprehensive collection of consistency conditions through it.

Contents

1 Introduction **4**

2 Shared Memory Systems **4**
2.1 Useful Definitions 4
2.2 Consistency Conditions for Shared Memory 5

3 Model of Transactional Memory **9**
3.1 Transactions and histories 9
3.2 Relations and Partial Orders 12
3.3 Legality 13

4 Correctness Conditions for Transactional Memory **13**
4.1 Strict Serializability 13
4.2 Serializability 14
4.3 Opacity 16
4.4 Causal Consistency 17
4.5 Causal Serializability 18
4.6 Virtual World Consistency 18
4.7 Snapshot Isolation 20

5 Additional Consistency Conditions for Database Systems **21**
5.1 Useful Assumptions 21
5.2 View Serializability 21
5.3 Conflict Serializability 22
5.4 Recoverability 23
5.5 Avoiding Cascading Aborts 23
5.6 Strictness 23
5.7 Rigorousness 23

List of Figures

1 Example of a sequentially consistent history. . . . 5
2 Example of a linearizable history. . . . 6
3 Example of a casually consistent history. . . . 7
4 Sequential history for process p_3. . . . 7
5 Example of a PRAM consistent history. . . . 8
6 Example of a cache consistent history which is not PRAM consistent. . . . 8
7 Example of a PRAM consistent history which is not cache consistent. . . . 8
8 History that is PRAM and cache consistent but not processor consistent. . . . 9
9 A slowly consistent history, which is not PRAM nor cache consistent. . . . 9
10 Example showing that s-strict serializability is not a prefix-closed property. . . . 14
11 A history that is serializable but not strictly serializable. . . . 15
12 A strict serializable history which is not opaque. . . . 16
13 A causally consistent history which is not serializable. . . . 17
14 A causally serializable history which is not serializable. . . . 18
15 A virtual world consistent history which is not opaque. . . . 19
16 A virtual world consistent history which is not strong virtual world consistent. . 19
17 Example of a history which is not recoverable. . . . 23

1 Introduction

This survey presents a comprehensive collection of consistency conditions, starting with memory-consistency, then presenting consistency for transactional memory (TM) systems, and finally discussing database systems consistency. Most TM consistency conditions originate from either shared-memory consistency or database consistency, so including formal definitions of most consistency conditions from these two areas in this survey provide a good insight in the origins of TM consistency research.

The survey starts by presenting in Section 2 useful definitions and a comprehensive collection of memory-consistency conditions, all expressed in a formal way by emplying the formal framework of Section 2.1. Then, in Section 3, we extend this model to make it appropriate for TM computing and we extensively discuss in Section 4 well-known consistency conditions for TM, as well as several variants of them, and their properties. Section 5 discusses additional consistency conditions for database systems.

2 Shared Memory Systems

2.1 Useful Definitions

In this section, we describe the standard shared memory model, based on which the TM model described later, is derived. Most of the definitions in this section resemble those in [16].

We consider an asynchronous system of a collection of processes that are executed concurrently. Each process is a software entity that is being executed sequentially. The processes communicate by accessing shared objects, called *base objects*. A base object provides a set of *primitive operations* or *primitives* for manipulating (reading or writing) its state. We say that an operation is *non-trivial* if it may update the state of the object; otherwise, the operation is *trivial*. An *implementation* of a shared object from base objects uses the base objects to store the state of the implemented object and provides algorithms, one for each process, for each operation that it supports.

An *event* is either an invocation *inv* of an operation *op* or a response *res* to some operation *op*. Each event contains the object identifier on which the operation is invoked, the identifier of the process that executes the event, and the arguments of the operation (if the event is an invocation of an operation) or its returned values (if the event is a response to an operation). A *history* is a (finite or infinite) sequence of events. A response *res matches* an invocation *inv* in some history H, if they are both by the same process p on the same object, *res* follows *inv* in H, and there is no other response by p between *inv* and *res* in H. An invocation is *pending* in a history if no matching response follows the invocation. For a history H, $complete(H)$ is the largest subsequence of H containing (1) those invocations in H for which there are matching responses, and (2) their matching responses. An operation *op* in a history is a pair consisting of an invocation $inv(op)$ and the next matching response $res(op)$ (if it exists).

Fix any history H. We define the partial *real-time order* $<_H$ on operations in H as follows:

- for each pair of operations op_1 and op_2 in H, $op_1 <_H op_2$ if and only if $res(op_1)$ precedes $inv(op_2)$ in H.

Two operations op_1 and op_2 in H are said to be *concurrent* if neither $op_1 <_H op_2$ nor $op_2 <_H op_1$. H is *sequential* if there are no concurrent operations in H. We remark that if H is sequential, then $<_H$ is a total order.

A *process subhistory* $H \mid p$ of H is the subsequence of all events in H executed by process p. For each object o, an *object subhistory* $H \mid o$ is defined similarly. H is *well-formed* if each process

subhistory $H \mid p$ of H is sequential. For the rest of this section, all histories are considered to be well-formed if not specified differently. A history H' is *equivalent* to H if, for every process p, $H \mid p = H' \mid p$.

We define a partial order, called *program order* and denoted $<_H^{pr}$, on operations in H, as follows:

- for each pair of operations op_1 and op_2 in H, $op_1 <_H^{pr} op_2$ if and only if op_1 and op_2 are executed by the same process and $res(op_1)$ precedes $inv(op_2)$ in H.

We remark that the following holds: $<_H^{pr} = \cup_i (<_{H|p_i})$.

Consider any sequential history S. For each process p_i, we define $S \mid^w p_i$ to be the subsequence of S consisting of all events in $S \mid p_i$, as well as invocations and responses of non-trivial (e.g. write) operations in S.

A set $\mathcal{H}$ of histories is *prefix-closed* if, for each history H in $\mathcal{H}$, every prefix of H is also in $\mathcal{H}$. A history H is a *single-object history* for some object o, if $H \mid o = H$. A *sequential specification* for an object o is a prefix-closed set of single-object sequential histories for o. A sequential history H is *legal*, if for each object o, $H \mid o$ belongs to the sequential specification for o.

2.2 Consistency Conditions for Shared Memory

In this section, we provide formal definitions for most well-known memory consistency conditions. We express all these conditions using the framework provided in Section 2.1. We start with *sequential consistency* which was introduced by Lamport in [19].

Definition 1. *A history H is* sequentially consistent *if it can be extended (by appending zero or more response events) to some history H' such that:*

- *$complete(H')$ is equivalent to some legal sequential history S.*

Figure 1 shows an example of a history. Here, the time axis is progressing from left to right, and each operation is illustrated by an interval. Overlapping intervals indicate concurrent operations. Each operation is executed by the process noted on the left (p_1, p_2 and so on). In Figure 1, we use $W(x)a$ to denote a write operation to the shared variable x with the value a. A similar notation is used for the read operations. We assume that the initial value of each data item is 0. Notice that these conventions are followed to all subsequent examples of this section unless stated otherwise.

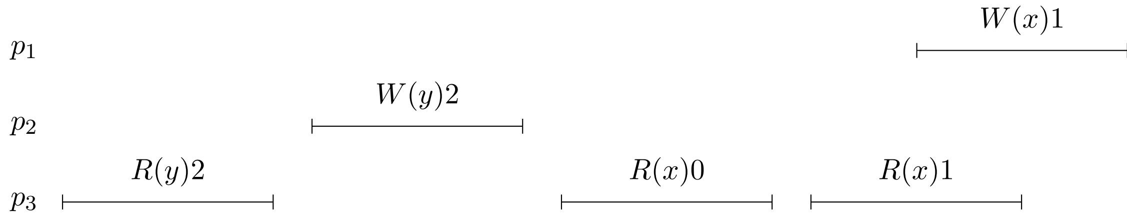

Figure 1: Example of a sequentially consistent history.

The history of Figure 1 is sequentially consistent since we can serialize the events as follows:

$$\langle inv(W(y)2), res(W(y)2)\rangle;\ \langle inv(R(y)2), res(R(y)2)\rangle;\ \langle inv(R(x)0), res(R(x)0)\rangle;$$
$$\langle inv(W(x)1), res(W(x)1)\rangle;\ \langle inv(R(x)1), res(R(x)1)\rangle.$$

Linearizability was introduced by Herlihy and Wing [16].

Definition 2. *A history H is* linearizable *if it can be extended (by appending zero or more response events) to some history H' such that:*

- *$complete(H')$ is equivalent to some legal sequential history S, and*
- $<_H \subseteq <_S$.

It follows from the definitions of sequential consistency and linearizability that if some history satisfies linearizability, it also satisfies sequential consistency. Thus every linearizable history is also sequentially consistent, but the opposite is not necessarily true. For instance, the history shown in Figure 1 is not linearizable, because linearizability respects the real-time order. So, we cannot place the serialization point of operation $W(y)2$ before that of $R(y)2$.

An example of a linearizable history is shown in Figure 2. Note, that since operations $R(x)1$ and $W(x)1$ are concurrent, they can be serialized in any order.

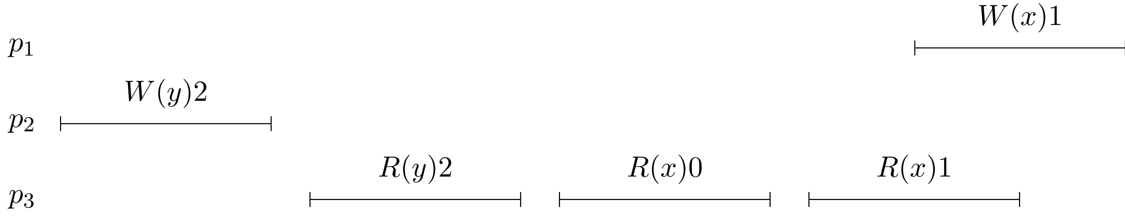

Figure 2: Example of a linearizable history.

Causal consistency was introduced by Hutto and Ahamad [17]. More formally, it was defined in [2]. A *reads-from* relation, denoted by $<^r_H$, on operations of a history H is defined as follows:

- for each operation op in H that returns a value for a shared object o (i.e. op reads o), we add, if possible, in $<^r_H$ a single pair (op', op) such that (a) op' is an operation of H that performs a non-trivial operation storing into o the same value returned by op, (b) $op \not<_H op'$, and (c) there is no other pair $\langle op', op \rangle$ in $<^r_H$.

Notice that for each trivial operation op that reads a value x from o, there may be multiple non-trivial operations in H that stores x into o. Still, $<^r_H$ contains only a single pair (op', op) where op' can be any of the non-trivial operations that store x into o and satisfy the properties above. We define $\mathcal{R}_\mathcal{H}$ to be the set of **all** possible reads-from relations on H.

For each $<^r_H$ in $\mathcal{R}_\mathcal{H}$, we define the *causal* relation for $<^r_H$ on operations in H to be the transitive closure of $<^r_H \cup <^{pr}_H$. We define $\mathcal{C}_\mathcal{H}$ to be the set of all causal relations in H. Let $<^r_H \in \mathcal{R}_\mathcal{H}$ be any reads-from relation and let $<^c_H \in \mathcal{C}_\mathcal{H}$ be the causal relation for $<^r_H$. We remark that, by the way $<^r_H$ is defined, $<^c_H$ *does not contain any cycles*, i.e. there is no sequence of k operations $op_1, \ldots, op_k$ in H such that $op_1 <^c_H op_2 <^c_H \ldots <^c_H op_k < op_1$, where $k > 1$ is any integer.

Definition 3. *A history H is* causally consistent *if it can be extended (by appending zero or more response events) to some history H' such that there exists a causal relation $<^c_{H'}$ in $\mathcal{C}_{\mathcal{H}'}$ for which the following holds: for each process p_i, there exists a sequential history S_i equivalent to $complete(H')$ such that*

- *$S_i \mid^w p_i$ is legal, and*
- *for each pair of operations op_1 and op_2 from S_i, if $op_1 <^c_{H'} op_2$, then $op_1 <_{S_i} op_2$.*

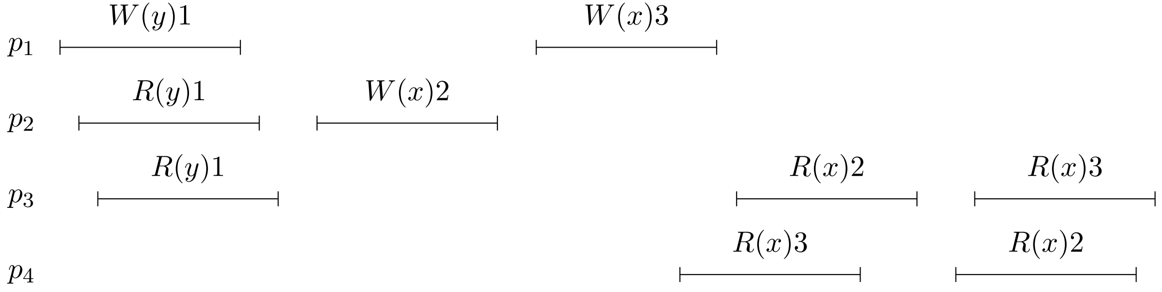

Figure 3: Example of a casually consistent history.

Let's discuss the history shown in Figure 3. Notice that the write operations $W(y)1$ and $W(x)2$ are related by the causality order by transitivity: the operations $W(y)1$ and $R(y)1$ are related by any reads-from relation (defined on the operations of this history) and the operations $R(y)1$ and $W(x)2$ are performed by the same process. This history is not sequentially consistent because processes p_3 and p_4 observe the values written by $W(x)2$ and $W(x)3$ in different order. However this history is casually consistent since there exists a sequential history for each process that ensures the properties of Definition 3. For example, the sequential history for process p_3 is shown in Figure 4.

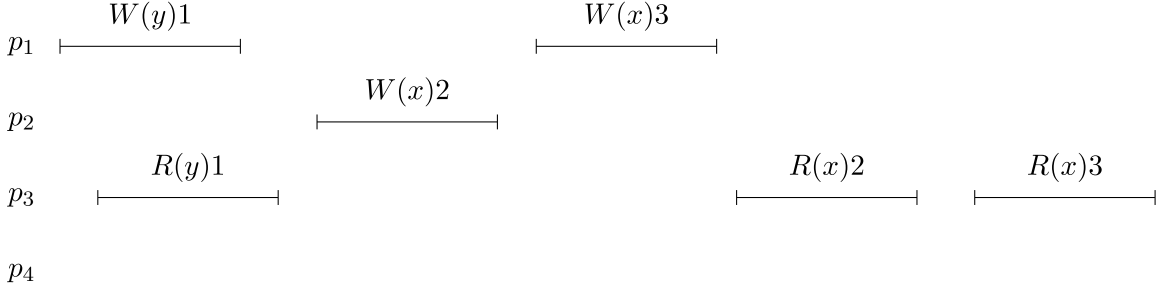

Figure 4: Sequential history for process p_3.

Pipelined RAM was introduced by Lipton and Sandberg [20]. Informally, the definition of PRAM consistency is as follows [25]: the memory model is PRAM consistent if for each process p, non-trivial operations performed by p are seen in program order by all processes. However, a process q may see non-trivial operations performed by different processes in different order than other processes.

The next formal definition follows Ahamad et al [2].

Definition 4. *A history H is PRAM consistent if it can be extended (by appending zero or more response events) to some history H' such that, for each process p_i, there exists a sequential history S_i equivalent to $complete(H')$ for which it holds that*

- *$S_i \mid^w p_i$ is legal.*

The history shown in Figure 5 is PRAM consistent but not casually consistent. All operations in Figure 5 are related by the causality order: $W(x)0 <^c_H W(x)1 <^c_H R(x)1 <^c_H W(y)2 <^c_H R(y)2 <^c_H R(x)0$. For this reason there is only one way to construct a sequential history equivalent to *complete*(H') (notice that here $H' = H$ since all operations are completed in H) that respects the causality order. But this sequential history is not legal because $R(x)0$ does not return the last value written to shared object x.

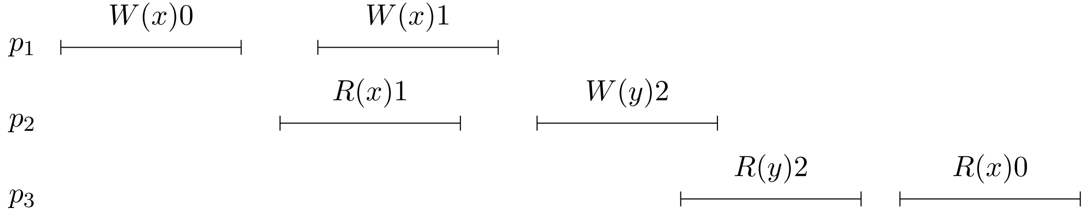

Figure 5: Example of a PRAM consistent history.

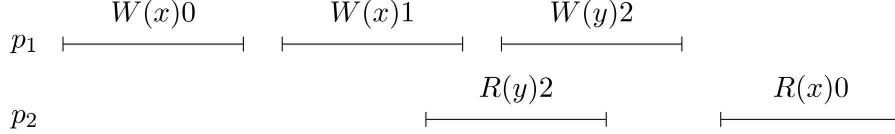

Figure 6: Example of a cache consistent history which is not PRAM consistent.

Goodman defined *cache consistency* (or *coherence*) as follows [12]: each read of the same memory location is guaranteed to see the "most recently written" value there. In other words, all writes to the same memory location are performed in some sequential order [25].

Using the formalism of Section 2.1 and the definition given in [1], we can define cache consistency as follows:

Definition 5. *A history H is* cache consistent *or* coherent *if it can be extended (by appending zero or more response events) to some history H' such that, for each object o, there exists a sequential history S_o equivalent to complete$(H') \mid o$ for which the following hold:*

- *S_o is legal, and*
- *for any pair of operations op_1 and op_2 in S_o, if $op_1 <_H^{pr} op_2$, then $op_1 <_{S_o} op_2$.*

Figure 6 shows a history that is cache consistent but not PRAM consistent. An example of a history that is PRAM consistent but not cache consistent appears in Figure 7.

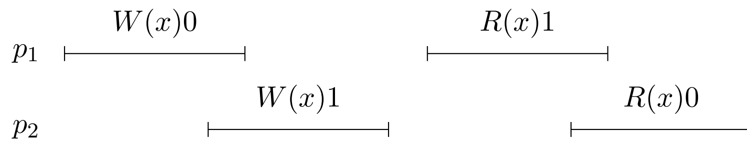

Figure 7: Example of a PRAM consistent history which is not cache consistent.

The original definition of *processor consistency* was given by Goodman [12]. We remark that there is some ambiguity on whether that definition of processor consistency meant to be stronger than cache consistency. Based on arguments from [1], we present processor consistency as a stronger property than cache consistency. Informally, processor consistency is a combination of PRAM consistency and cache consistency. To define it in a formal way, we will express the definition provided in [1], using the formalism discussed in Section 2.1, as follows:

Definition 6. *A history H is* processor consistent *if it can be extended (by appending zero or more response events) to some history H' such that, for each process p_i, there exists a sequential history S_i equivalent to complete(H') for which the following hold:*

- *$S_i \mid^w p_i$ is legal, and*

- *for each pair of operations op_1 and op_2 that write to the same shared object o, either all total orders $<_{S_i}$ contain a pair (op_1, op_2) or all $<_{S_i}$ contain a pair (op_2, op_1).*

We remark that an alternative definition of processor consistency is provided by Gharachorloo *et al.* [11] which is omitted here. A discussion about these definitions can be found in a paper by Ahamad *et al.* [1].

Processor consistency is stronger than PRAM consistency and cache consistency. The converse is not true. An example of a history that is PRAM consistent and cache consistent but not processor consistent is shown in Figure 8.

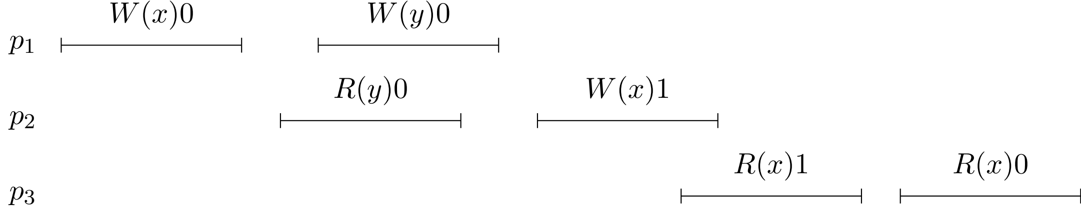

Figure 8: History that is PRAM and cache consistent but not processor consistent.

Slow consistency is a weakening of PRAM consistency [17] regarding accesses to shared memory objects. Informally, it requires that all processes agree on the order of the write operations that are performed to each object by a single processor. In other words, a history is slowly consistent if all of its object subhistories are PRAM consistent.

Definition 7. *A history H is* slowly consistent *if it can be extended (by appending zero or more response events) to some history H' such that, for each process p_i and for each object o, there exists a sequential history $S_{i,o}$ equivalent to complete$(H') \mid o$ for which the following hold:*

- *$S_{i,o} \mid^w p_i$ is legal.*

Slow consistency is weaker than PRAM consistency and cache consistency. Figure 9 gives an example of such a history.

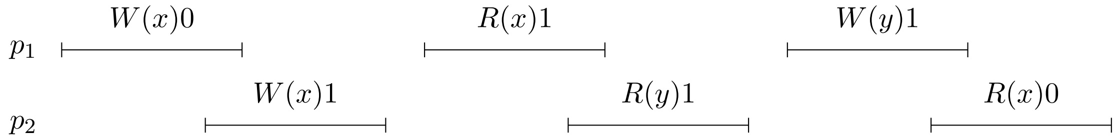

Figure 9: A slowly consistent history, which is not PRAM nor cache consistent.

3 Model of Transactional Memory

In this section, we describe a model for transactional memory (TM) computing following [10] and [13].

3.1 Transactions and histories

Transactional memory (TM) is a parallel programing paradigm which employs transactions to synchronize the execution of processes. A *transaction* is a piece of code which may read or write one or more pieces of data, called *data items*. Roughly speaking, a data item is an abstract object which provides convenient encapsulation of data. A data item may be accessed by several

processes simultaneously in a concurrent environment. A TM algorithm uses base objects to store the state of each data item and ensures synchronization between processes reading or writing data items. A transaction may commit or abort. If it *commits*, all of its updates to data items are realized, whereas if it *aborts*, all of its updates are discarded.

In order to *read* and *write* a data item, the TM algorithm provides implementations of routines ReadDI and WriteDI, respectively. Additionally, the TM algorithm provides implementations of routines Abort and Commit, which are called when a transaction tries to abort or commit, respectively. We refer to all these routines as *transactional operations*[1]. A transactional operation starts its execution when the process executing it issues an *invocation* for it; the operation completes its execution when the process executing it returns a *response*. The response for an instance of Commit 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 process 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. As in Section 2.1, a *history* is a finite sequence of events. Thus, in a history H, every completed operation *op* is represented as a pair of 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 all invocations and responses in H of transactional operations executed on x. For each process p_i, we denote by $H \mid p_i$ the subsequence of all invocations and responses in H of transactional operations executed by process 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 transaction T *is in* H or H *contains* T, if there are events in H issued or produced for T. Let T be a transaction in H. The *transaction subhistory* of H for T, denoted by $H \mid T$, is the subsequence of all events in H issued or produced for T. An *update* transaction in H is a transaction for which at least one invocation of WriteDI is issued. No invocation of WriteDI is issued for a *read-only* transaction.

A history H is said to be *well-formed* if, for each transaction T in H, all of the following hold:

- the first event in $H \mid T$ is an invocation;
- any invocation in $H \mid T$ that is not the last event of $H \mid T$ is followed by a matching response;
- any response in $H \mid T$ that is not the last event of $H \mid T$ is followed by an invocation;
- no events in $H \mid T$ follow C_T or A_T;
- if T' is any transaction in H executed by the same process 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*.

[1] Whenever it is clear from the context, we use the term operation to refer to a transactional operation.

A transaction is *commit-pending* in H if it is live in H and $H \mid T$ includes an invocation to COMMIT 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 process 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 $Complete_1(H)$ the set of histories defined as follows. A history H' is in $Complete_1(H)$ if and only if:

1. H' is well-formed;
2. every transaction T which is live and contains a pending operation in H is completed in H' as follows:
   (a) if T is commit-pending then T is completed in H' by adding either C_T or A_T;
   (b) if T is not commit-pending then T is aborted in H' by adding A_T;
3. every transaction T which is live and doesn't contain a pending operation in H is aborted in H' by adding ABORT_T and A_T.

Roughly speaking, each history in $complete_1(H)$ 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* consists of the state of each process and the state of each base object. In an *initial configuration*, processes and base objects are in initial states. A *step* of a process consists of 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 process to change; as a step, we also consider the invocation of a transactional operation or the response to such an invocation (we remark that a step of this kind does not change the state of any base object). Each step is executed atomically. An *execution* α is a sequence of steps. An execution is *legal* 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 operations performed on the object are in accordance with its specification (and the state of the object at configuration C). Given an execution α, the history of α, denoted by H_α, is the subsequence of α containing only the invocations and responses of transactional operations. Given a set of executions $\mathcal{A}$, let $\mathcal{H}(\mathcal{A}) = \{H_\alpha \text{ such that } \alpha \in \mathcal{A}\}$, i.e. $\mathcal{H}(\mathcal{A})$ contains all histories of executions in $\mathcal{A}$.

Let L be any subset of the live transactions in H. Denote by H_L an extension of H constructed as follows:

- for each transaction $T \in L$,
  (a) if T contains a pending transactional operation in H, then A_T is added in H_L;
  (b) if T does not contain any pending transactional operation in H, then the sequence $Abort_T, A_T$ is added in H_L.

Notice that in either case, T is aborted in H_L. We remark that the order in which transactions in L are inspected (and thus the order in which the extensions are added in H_L for each of them) is not essential. Let $\mathcal{LH}_H = \{H_L \mid \text{ for each subset } L \text{ of live transactions in } H\}$.

Fix any history $H_L \in \mathcal{LH}_H$ and let L be the subset of live transactions of H based on which H_L was constructed. Let $\mathcal{A}_{H_L}$ be a set containing each legal execution α of the transactions in H_L other than those in L such that each transaction is executed in α by the same process as in

H_L until it completes, and each process executes in α its transactions in the same order as in H_L. Let $\mathcal{A}_H = \bigcup_{H_L \in \mathcal{LH}_H} \mathcal{A}_{H_L}$ and let $\mathcal{HA}_H = \{H_\alpha |$ for each $\alpha \in \mathcal{A}_H\}$. Denote by $Complete_2(H)$ the maximal subset of $\mathcal{HA}_H$ such that all of the following conditions hold:

- for each history $H' \in Complete_2(H)$ and for each transaction T in H':
  (a) there exists an event e such that if $H_T = (H' \mid T) \uparrow e$, then $H_T = H \mid T$, and
  (b) if T is a committed transaction in H', then for each WRITEDI issued by T in H', if *res* is the response to this WRITEDI, then $res \in H_T$.

Roughly speaking, each history in $Complete_2(H)$ is an extension of a subsequence of H. This subsequence contains all completed transactions in H and some live transactions from those that, in H, have issued all their calls to WRITEDI and have received the response for them.

3.2 Relations and Partial Orders

Fix any well-formed history H. Now, the real time order $<_H$ is a partial order defined *over 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.

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

- for any two transactional operations op_1 and op_2 executed by transactions in H, if H contains a response for op_1 and this response 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 $op_2 <_H^{op} op_1$. A history is *operational-wise sequential* if no two operations in it are concurrent.

We define a *reads-from relation* on transactions in H, denoted by $<_H^r$, as follows:

- for each transaction T that contains an instance *op* of READDI which reads from a data item x, we add, if possible, a single pair $\langle T', T\rangle$ in $<_H^r$ such that:
  - T' contains an instance op' of WRITEDI which writes into x the same value read by *op*, and $op \not<_H^{op} op'$,

  (b) $\langle T', T\rangle$ does not create a cycle in $<_H^r$.

Notice that for each transaction that contains an instance of READDI which reads a value v for a data item x, there may be multiple transactions that contain an instance of WRITEDI called for x with value v. Still, each $<_H^r$ contains only a single pair (T', T) where T' can be any of these transactions containing such an instance of WRITEDI. We define $\mathcal{R}_\mathcal{H}$ to be the set of **all** possible reads-from relations on 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 remark that the causal relation for $<_H^r$ is not necessarily a partial order since it may contain cycles. We define $\mathcal{C}_\mathcal{H}$ to be the set of **all** causal relations in H.

Let S be a sequential history. We say that S *respects* some relation $<$ *over 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 $T_1 <_S T_2$. Notice that a partial order is also a relation, so this definition holds for partial orders as well.

Similarly, let S_{op} be an operational-wise sequential history. We say that S_{op} respects some partial order $<^{op}$ *over the set of transactional operations* in H if the following holds: for any operations op_1 and op_2 in S_{op}, if $op_1 <^{op} op_2$, then $op_1 <^{op}_{S_{op}} op_2$.

3.3 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. Let S' be the subsequence of all events issued and received either by transaction T in S itself or by any committed transaction T_i in S such that $T_i <_S T$. Then T is *legal* in S, if S' is a legal sequential history; we remark that if T aborts, then we exclude from S' the last invocation and response for T.

4 Correctness Conditions for Transactional Memory

4.1 Strict Serializability

Strict serializability was first introduced in [22] as a (strong) consistency condition for database systems. In TM computing, it could be expressed (by generalizing its original definition) in several different ways, as discussed below.

Definition 8 (Definition 1 of Strict Serializability). *We say that an execution α is* w-strictly serializable *if it is possible to do all of the following:*

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

These serialization points should be inserted, so that, in the sequential execution σ constructed by serially executing each transaction $T \in A \cup B$ at the point that its serialization point has been inserted:

- *if $T \in A$, the same transactional operations, as in α, are invoked by T in σ and the response of each such operation in σ is the same as that in α, and*
- *if $T \in B$, a prefix of the operations invoked by T in σ are the same as all operations invoked by T in α and the response of each such operation in σ is the same as that in α.*

In Definition 9, we give a stronger version of strict serializability.

Definition 9 (Definition 2 of Strict Serializability). *A history H is* s-strictly serializable, *if there exists a history $H' \in Complete_1(H)$ 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 8 provides a weaker version of strict serializability than Definition 9, since it allows a transaction to read a value for a data item written by some other transaction that is not committed or commit-pending in H. However, 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 10. H is w-strictly serializable and s-strictly serializable but H_1 is only w-strictly serializable. Thus, s-strict serializability is not a prefix-closed property. On the contrary, w-strict serializability is a prefix-closed property.

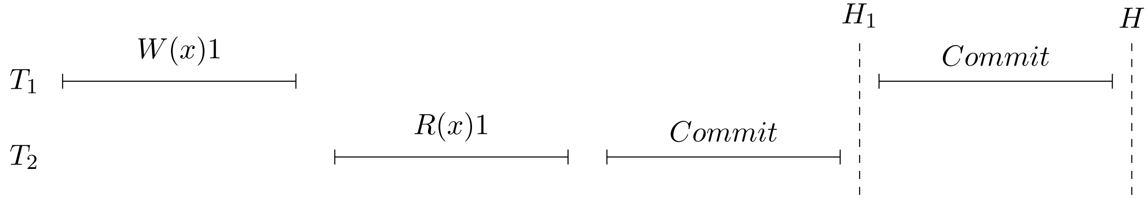

Figure 10: Example showing that s-strict serializability is not a prefix-closed property.

We finally introduce one more version of strict serializability. It is expressed in such a way that it is explicitly prefix-closed.

Definition 10 (Definition 3 of Strict Serializability). *A history H is l-strict serializable if, for each prefix H_p of H, there exists a history H'_p from $Complete_2(H_p)$ and a history S equivalent to $comm(H'_p)$ such that:*

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

We remark that prefix-closure can be imposed to w-strict serializability in an explicit way, as it is done in Definition 10 for l-strict serializability, i.e. by directly stating in Definition 9 that all prefixes of the considered history must also satisfy the properties of the definition. We remark that then l-strict serializability would be weaker than the prefix-closed version of w-strict serializability. Similarly, a non-prefix-closed version of l-strict serializability (which can be derived in a straightforward way by Definition 10 by removing the requirement that the properties of the definition hold for all prefixes of H and instead insist that they hold only for H) is weaker than s-strict serializability.

4.2 Serializability

As with strict serializability, serializability was first introduced in [22] as a consistency condition for databases. Below we discuss several flavors of serializability in a way similar to that for strict serializability.

Definition 11 (Definition 1 of Serializability). *We say that an execution α is* w-serializable *if it is possible to do all of the following:*

- *If A is the set of all complete transactions in α that are not aborted, to insert, for each transaction $T \in A$, a serialization point $*_T$ in α.*

- *To choose a subset B of the live transactions in α and insert, for each transaction $T \in B$, a serialization point $*_T$ in α.*

These serialization points should be inserted, so that, in the sequential execution σ constructed by serially executing each transaction $T \in A \cup B$ at the point that its serialization point has been inserted:

- *if $T \in A$, the same transactional operations, as in α, are invoked by T in σ and the response of each such operation in σ is the same as that in α, and*
- *if $T \in B$, a prefix of the operations invoked by T in σ are the same as all operations invoked by T in α and the response of each such operation in σ is the same as that in α.*
- *for each process p, the transactions by p appear in $H_\sigma \mid p$ in the same order as in $H_\alpha \mid p$.*

In Definition 12, we give a stronger version of serializability.

Definition 12 (Definition 2 of Serializability)**.** *A history H is* s-serializable*, if there exists a history $H' \in Complete_1(H)$ and a history S equivalent to $comm(H')$ such that:*

- *S is a legal sequential history.*

Notice that S in Definition 12 respects the program order of transactional operations executed by the same process in H. This is implied because of the definition of equivalent histories.

We remark that, similarly to the corresponding definitions of strict serializability, Definition 11 provides a weaker version of serializability than Definition 12.

We finally present a third definition of serializability, weaker than Definition 12 (but stronger than Definition 11).

Definition 13 (Definition 3 of Serializability)**.** *A history H is* l-serializable *if there exists a history H' from $Complete_2(H)$ and a history S equivalent to $comm(H')$ such that:*

- *S is a legal sequential history.*

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. For instance, the history shown in Figure 11 is serializable but not strictly serializable.

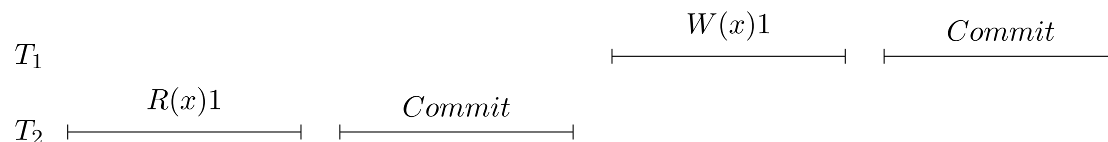

Figure 11: A history that is serializable but not strictly serializable.

It is worth-pointing out that w-serializability, s-serializability, and l-serializability (as well as sequential consistency) are not prefix-closed properties. This is so, since it is easy to design a serializable history H in which a committed transaction T (executed by some process p) reads for some data item x a value v written by some 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. Such a history is shown in Figure 11. Apparently, the prefix of H up until C_T is neither w-serializable, nor s-serializable, nor l-serializable.

We remark that prefix-closure can be imposed to w-serializability and l-serializability in an explicit way, as done in Definition 10 for l-strict serializability, i.e. by directly stating in each of the Definitions 12 and 13 that all prefixes of the considered history must also satisfy the properties of the definition. However, the versions that would then result may be too strong to be of much interest.

Imposing prefix closure to the consistency conditions presented in Sections 4.4-4.5 may be too restrictive as well. Thus, we present the non-prefix-closed version of them given that it is straightforward to derive their prefix-closed version, in an explicit way, as described above.

4.3 Opacity

The opacity correctness condition was first introduced in [13]. In [14], a prefix-closed version of it was formally stated. Here, we will present the later version.

Definition 14 (Opacity [14])**.** *A history H is opaque if, for each prefix H_p of H, there exists a sequential history S_p equivalent to some history $H'_p \in Complete_1(H_p)$ such that:*

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

Figure 12: A strict serializable history which is not opaque.

Opacity is stronger than s-strict serializability. Figure 12 shows an example of a history that is not opaque but is s-strictly serializable. This history is not opaque because it violates the last condition of Definition 14; specifically, transaction T_2 cannot be legal.

S-strict serializability doesn't impose any restrictions concerning non-committed transactions, whereas 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 committed 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 w-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 w-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 next provide a weaker form of opacity based on the definition of $Complete_2$:

Definition 15 (L-Opacity)**.** *A history H is* l-opaque *if, for each prefix H_p of H, there exists a sequential history S_p equivalent to some history $H'_p \in Complete_2(H_p)$, such that:*

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

We remark that non-prefix-closed versions of opacity and l-opacity can be derived in a straightforward way by Definitions 14 and 15, respectively, by removing the requirement that the properties of the definition hold for all prefixes of H and instead insist that they hold only for H. Apparently, these versions would be weaker than those presented above.

A weaker version of opacity, called TMS1, appears in [10]. That paper also presents a stronger version of opacity, called TMS2.

4.4 Causal Consistency

Causal consistency was first informally introduced for the shared memory model in [17] and then formally defined in [2]. In [23], causal consistency was stated in terms of a transactional memory model. Here we provide a formal defintion for it using the framework of Section 3.

Definition 16. *A history H is* causally consistent *(*l-causally consistent*) if there exists a causal relation $<^c_H$ in $\mathcal{C}_\mathcal{H}$ such that, for each process p_i, there exists a history $H'_i \in Complete_1(H_p)$ ($H'_i \in Complete_2(H_p)$, respectively) and a sequential history S_i such that:*

- *S_i is equivalent to H'_i,*
- *S_i respects the causality order $<^c_H$, and*
- *every committed transaction executed by p_i is legal in S_i.*

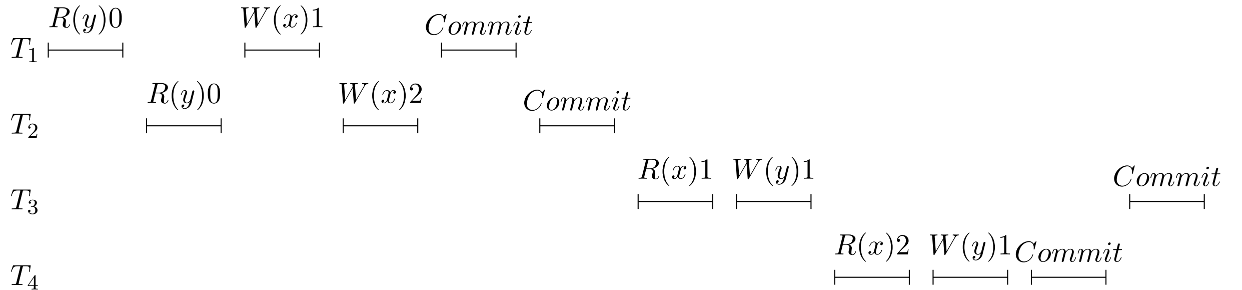

Figure 13: A causally consistent history which is not serializable.

Causal consistency is a weaker property than serializability. For instance, Figure 13 shows an example of a history which is causally consistent but not 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.

4.5 Causal Serializability

Causal serializability was introduced in [23] 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 17. *A history H is causally serializable if there exists a causal relation $<^c_H$ in $\mathcal{C}_\mathcal{H}$ such that, for each process p_i, there exists a history $H'_i \in Complete_1(H_p)$ ($H'_i \in Complete_2(H_p)$, respectively) and a sequential history S_i such that:*

- *S_i is equivalent to H'_i,*
- *S_i respects the causality order $<^c_H$,*
- *every committed transaction executed by p_i is legal in S_i, and*
- *for each data item x and for each pair of transactions T_j and T_k which contain* WriteDI *instances writing to x, either all total orders $<_{S_i}$ contain a pair (T_j, T_k) or all $<_{S_i}$ contain a pair (T_k, T_j).*

Obviously, every causally serializable history satisfies the properties of causal consistency, but the opposite is not true. For instance, the history shown in Figure 13 is not causally serializable, because processes executing transactions T_3 and T_4 should see writes from T_1 and T_2 to data item x in the same order.

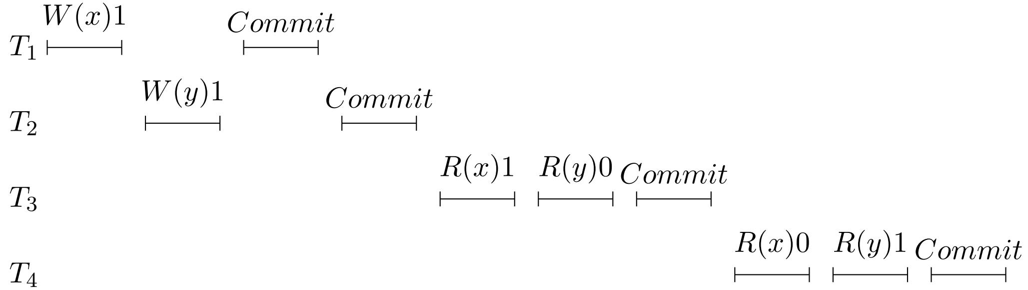

Figure 14: A causally serializable history which is not serializable.

Figure 14 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.

4.6 Virtual World Consistency

Virtual World Consistency (VWC) was defined in [18] as a family of consistency conditions. Informally, VWC ensures serializability or strict serializability for committed transactions but a weaker condition for non-committed transactions than that of opacity.

For each transaction T in history H and each causal relation $<^c_H$ in $\mathcal{C}_\mathcal{H}$, we define the *causal past* of T denoted by $past_T(H, <^c_H)$ as the subsequence of all events issued and received either by transaction T in H itself or by any transaction T_i in H such that $T_i <^c_H T$.

Definition 18. *A history H is* virtual world consistent (l-virtual world consistent) *if there exists a causal relation $<^{c}_{H}$ in $\mathcal{C}_{\mathcal{H}}$ and a history $H' \in Complete_1(H)$ ($H' \in Complete_2(H)$, respectively) such that:*

- *there exists a legal sequential history S which is equivalent to $comm(H')$, 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 $<^{c}_{H}$.*

Definition 19. *A history H is* strong virtual world consistent (l-strong virtual world consistent) *if there exists a causal relation $<^{c}_{H}$ in $\mathcal{C}_{\mathcal{H}}$ and a history $H' \in Complete_1(H)$ ($H' \in Complete_2(H)$, respectively) 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 $<^{c}_{H}$.*

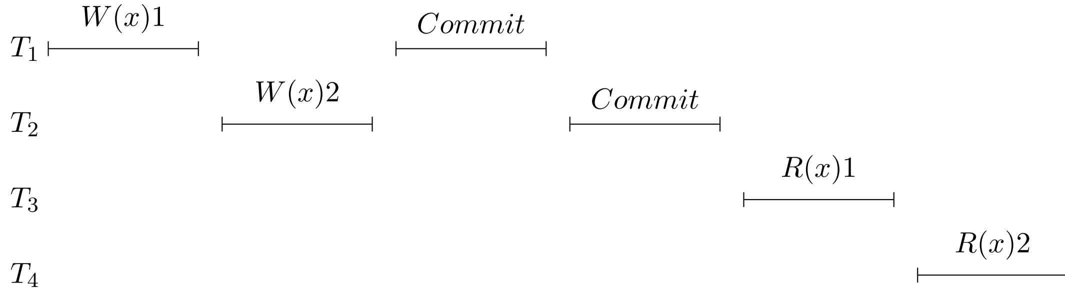

Figure 15: A virtual world consistent history which is not opaque.

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 15 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.

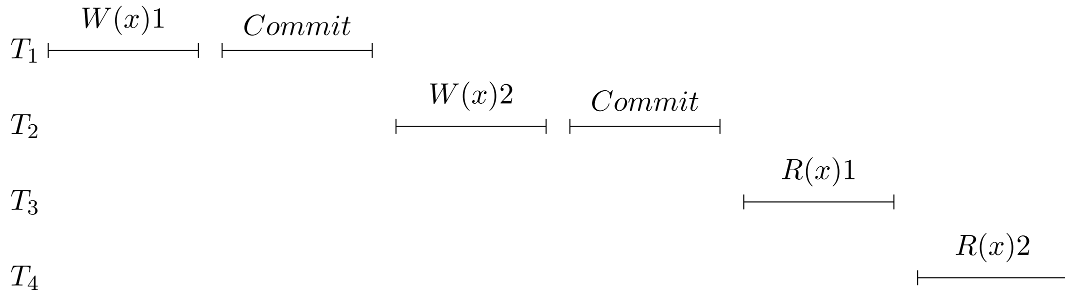

Figure 16: A virtual world consistent history which is not strong virtual world consistent.

The history shown in Figure 16 is a slightly modified version of the history shown in Figure 15. 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 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.

4.7 Snapshot Isolation

Snapshot isolation was originally introduced as a safety property in the database world [5, 21] to increase throughput for long read-only transactions. In the concept of TM, snapshot isolation has been studied in [3, 9, 24]. One of first formal definitions of snapshot isolation for the TM model was given in [8] a variation of which we present below.

Let T be a committed or commit-pending transaction in a history H. An instance *op* of ReadDI for some data item x executed by T is *global* if T has not invoked WriteDI on x before invoking *op*. Let $T \mid read_g$ be the longest subsequence of $H \mid T$ consisting only of the invocations of the global instances of ReadDI (and their responses if they exist) and $T \mid other$ be the subsequence $H \mid T - T \mid read_g$, i.e. $T \mid other$ consists of all invocations performed by T (and any matching responses) other than those comprising $T \mid read_g$. Let λ be the empty execution. Then we define T_g and T_o in the following way:

- $T_g = T \mid read_g \cdot commit_{T_g} \cdot C_{T_g}$ if $T \mid read_g \neq \lambda$, and $T_g = \lambda$ otherwise, and
- $T_o = T \mid other \cdot commit_{T_o} \cdot C_{T_o}$ if $T \mid other \neq \lambda$, and $T_o = \lambda$ otherwise.

Definition 20. *A history H satisfies* snapshot isolation (l-snapshot isolation)*, if there exists a history $H' \in Complete_1(H)$ ($H' \in Complete_2(H)$, respectively) such that:*

1. *no two committed transactions in H' are concurrent and invoke* WriteDI *on the same data item, and*
2. *for every committed transaction T (and for some of the commit-pending transactions) in H' it is possible to insert a read serialization point $*_{T,g}$ and a write serialization point $*_{T,o}$ such that: (i) $*_{T,g}$ precedes $*_{T,o}$, (ii) both $*_{T,g}$ and $*_{T,o}$ are inserted within the execution interval of T, and (iii) if $\sigma_{H'}$ is the sequence defined by these serialization points, in order, and $H_{\sigma_{H'}}$ is the history we get by replacing each $*_{T,g}$ with T_g and each $*_{T,o}$ with T_o in $\sigma_{H'}$, then $H_{\sigma_{H'}}$ is legal.*

Now, let $T \mid read$ be the longest subsequence of $H \mid T$ consisting only of ReadDI invocations and their corresponding responses and $T \mid write$ be the longest subsequence of $H \mid T$ consisting only of WriteDI invocations and their corresponding responses. Then we define T_r and T_w in the following way:

- $T_r = T \mid read \cdot commit_{T_r} \cdot C_{T_r}$ if $T \mid read \neq \lambda$, and $T_r = \lambda$ otherwise, and
- $T_w = T \mid write \cdot commit_{T_w} \cdot C_{T_w}$ if $T \mid write \neq \lambda$, and $T_w = \lambda$ otherwise.

Definition 21. *A history H satisfies* R/W-independent snapshot isolation (R/W-independent l-snapshot isolation) *if there exists a history $H' \in Complete_1(H)$ ($H' \in Complete_2(H)$, respectively) such that:*

1. *no two committed transactions in H' are concurrent and invoke* WriteDI *on the same data item, and*
2. *for every committed transaction T (and for some of the commit-pending transactions) in H' it is possible to insert a read serialization point $*_{T,r}$ and a write serialization point $*_{T,w}$ such that: (i) $*_{T,r}$ precedes $*_{T,w}$, (ii) both $*_{T,r}$ and $*_{T,w}$ are inserted within the execution interval of T, and (iii) if $\sigma_{H'}$ is the sequence defined by these serialization points, in order, and $H_{\sigma_{H'}}$ is the history we get by replacing each $*_{T,r}$ with T_r and each $*_{T,w}$ with T_w in $\sigma_{H'}$, then $H_{\sigma_{H'}}$ is legal.*

5 Additional Consistency Conditions for Database Systems

Some of the conditions presented in Section 4, such as strict serializability, serializability, causal consistency, and causal serializability have first appeared as consistency conditions for database systems (DBS). In this section, we discuss additional consistency conditions for DBS. These conditions have previously been discussed in [26]. From the conditions presented in [26], we have chosen those that seem to be the most meaningful to be studied for TM computing.

5.1 Useful Assumptions

The model for database systems differs from that considered in Section 3 in the following:

- DB transactions call READ and WRITE operations (as software transactions call READDI and WRITEDI, respectively);
- in each legal history G, every pair of conflicting operations are ordered , i.e. every pair of operations op_1, op_2, executed by different transactions, that access the same data item and one of them is a WRITE, is ordered so that either $op_1 <_G^{op} op_2$ or $op_2 <_G^{op} op_1$;
- In each legal history, all transactions are complete (aborted or committed).

Under these assumptions, some of the definitions presented in Section 4 can be significantly simplified. For instance, under this setting, the *reads-from relation* on the transactions of a history G (which, for clarity, we will call *DB-reads-from*) is defined as follows. A READ operation op executed by a transaction T on a data item x *reads-from* a write operation op' executed by transaction T', denoted $T' <_G^r T$, iff op' is *the last* WRITE operation on x such that $op' <_G^{op} op$. The *DB-reads-from relation* thus contain all pairs of transactions that satisfy the above properties. (For simplicity, we sometimes assume that there are two fictitious transactions in each history, an initial transaction that writes in all data items their initial values, and a final transaction which reads all data items.) For clarity, we use the letter G to denote DB histories and the letter H to denote TM histories. We use the same letter S for DB or TM sequential histories because in all our references to sequential histories below it is clear from the context whether the sequential history is a DB history or a TM history.

In each of Sections 5.2 and 5.3, we present two definitions. The first is usually simpler and corresponds to the DB version of the corresponding consistency condition. The second is what we believe is the meaningful version of it for TM computing.

5.2 View Serializability

Definition 22 (DB view serializability)**.** *A history G is* DB view serializable *if there exists a sequential history S such that:*

- *S is equivalent to G,*
- *S respects $<_G^r$,*
- *each committed transaction in S is legal.*

We now turn our attention on how we can define view serializability in a TM context. Let H_{op} be any operational-wise sequential history. We remark that for such histories, $<_{H_{op}}^r$, the reads-from relationship on transactions in H_{op} can be defined in a way similar to that in DB histories. Specifically, then $<_{H_{op}}^r$ is defined over the set of transactions in H_{op}, as follows:

- for any two transactions T_1 and T_2 in H_{op}, $T_1 <^r_{H_{op}} T_2$ if and only if all of the following conditions hold:
  - T_1 contains an instance op_1 of WriteDI which writes a value v into a data item x and T_2 contains an instance op_2 of ReadDI which reads the value v from x;
  - $op_1 <^{op}_{H_{op}} op_2$;
  - there is no instance op_3 of WriteDI in H_{op} that writes v into x and $op_1 <^{op}_{H_{op}} op_3 <^{op}_{H_{op}} op_2$.

We remark that Definition 23 employs this definition of the reads-from relation.

Definition 23 (TM view serializability)**.** *A history H is* TM view serializable *if there exists a history $H' \in Complete_1(H)$, an operational-wise sequential history H_{op} and a sequential history S such that:*

- *S and H_{op} are equivalent to H',*
- *H_{op} respects $<^{op}_{H'}$,*
- *S respects $<^r_{H_{op}}$,*
- *each committed transaction in S is legal.*

5.3 Conflict Serializability

Consider any *operationally-wise sequential* history H_{op}. We say that two operations op_1 and op_2, executed by distinct transactions, in H_{op}, *conflict*, if they access the same data item x and at least one of them is a write operation . We define a partial order, called *conflict order* and denoted by $<^f_{H_{op}}$, *over the set of transactional operations* in H_{op} as follows:

- for any two transactional operations op_1 and op_2 executed by transactions in H_{op}, $op_1 <^f_{H_{op}} op_2$ iff op_1 and op_2 conflict in H_{op} and $op_1 <^{op}_{H_{op}} op_2$.

Notice that, because of the assumptions discussed in Section 5.1, the above definitions can be also applied to a DB history G, as well.

Definition 24 (DB Conflict Serializability)**.** *A history G is* DB conflict serializable *if there exists a sequential history S such that:*

- *S is equivalent to G,*
- $<^f_G = <^f_S$,
- *each committed transaction in S is legal.*

Definition 25 (TM Conflict Serializability)**.** *A history H is* TM conflict serializable *if there exists a history $H' \in Complete_1(H)$, an operational-wise sequential history H_{op} and a sequential history S such that:*

- *H_{op} and S are equivalent to H,*
- $<^f_{H_{op}} = <^f_S$,
- *each committed transaction in S is legal.*

5.4 Recoverability

Recoverability (RC) was introduced for database systems [15]. It states that if a transaction T_1 reads some value written by another transaction T_2, then T_1 cannot commit before T_2 commits. Recoverability has been studied in [4] for a TM setting. Definition 26 follows that in [26].

Definition 26. *A history G is* recoverable *if for any two distinct transactions T_1 and T_2 in G such that $T_1 <^r_G T_2$ and T_2 is a committed transaction in G, the following holds:*

- *T_1 is also a committed transaction and it commits before T_2 commits in G.*

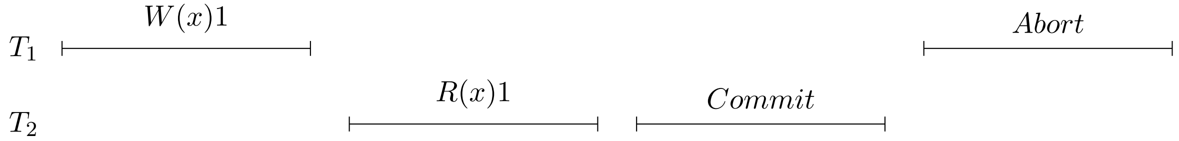

Figure 17: Example of a history which is not recoverable.

Figure 17 shows a history which includes events executed by two transactions T_1 and T_2. T_1 writes a value 1 to a data item x. Then T_2 reads this value and commits, so T_1 is not allowed to abort after that. Thus, the history shown on Figure 17 is not recoverable.

5.5 Avoiding Cascading Aborts

If some history satisfies the property of avoiding cascading aborts [6], then all values for data items ready by any transaction has been written by committed transactions only. Avoiding cascading aborts has been studied in [4] for TM computing. Definition 27 follows that in [26].

Definition 27. *A history G* avoids cascading aborts *if for any two distinct transactions T_1 and T_2 from G such that $T_1 <^r_G T_2$ the following holds:*

- *if op is the first read operation from T_2 which reads some value written by T_1, then C_{T_1} precedes the invocation of op in G.*

It is known [26, Theorem 11.2] that $ACA \subset RC$.

5.6 Strictness

Strictness (ST) for DB systems is presented in [6, 26]. For TM computing, it has been studied in [4]. Definition 28 follows that in [26].

Definition 28. *A history G is* strict *if for each transaction T_1 in G and for each read or write operation op on some data item x performed by T_1, if $T_2 \neq T_1$ is a transaction in G that performs a write operation op' on data item x in G such that $op' <^{op}_G op$ then T_2 completes (i.e. commits or aborts) in G before the invocation of op.*

It is known [26, Theorem 11.2] that $ST \subset ACA$.

5.7 Rigorousness

Rigorousness (RG) was introduced in [7]. It is studied in [4] for a TM setting. Our definition below follows that in [26].

Definition 29. *A history G is* rigorous *if G is strict and additionally satisfies the following condition:*

- *for any two transactions T_1 and T_2 in G such that T_2 contains a read operation op' which reads from a data item x and T_1 contains a write operation op which writes into x and $op' <_G^{op} op$, it holds that T_2 completes (i.e. commits or aborts) before the invocation of op in G.*

It is known [26, Theorem 11.2] that $RG \subset ST$.

References

[1] M. Ahamad, R. A. Bazzi, R. John, P. Kohli, and G. Neiger. The power of processor consistency. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures - SPAA '93*, pages 251–260, New York, New York, USA, aug 1993. ACM Press.

[2] 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.

[3] 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.

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

[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] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, jan 1987.

[7] Y. Breitbart, D. Georgakopoulos, M. Rusinkiewicz, and A. Silberschatz. On rigorous transaction scheduling. *IEEE Trans. Softw. Eng.*, 17(9):954–960, sep 1991.

[8] 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.

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

[10] 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.

[11] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proceedings of the 17th annual international symposium on Computer Architecture - ISCA '90*, volume 18, pages 15–26, New York, New York, USA, may 1990. ACM Press.

[12] J. R. Goodman. Cache consistency and sequential consistency. Technical report, University of Wisconsin - Madison, feb 1991.

[13] 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.

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

[15] V. Hadzilacos. A theory of reliability in database systems. *Journal of the ACM*, 35(1):121–145, jan 1988.

[16] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, jul 1990.

[17] 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.

[18] 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 SIROCCO 2009.

[19] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, C-28(9):690–691, sep 1979.

[20] R. J. Lipton and J. S. Sandberg. Pram: A scalable shared memory. Technical Report CS-TR-180-88, Princeton University, September 1988.

[21] 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.

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

[23] 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.

[24] 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.

[25] R. C. Steinke and G. J. Nutt. A unified theory of shared memory consistency. *Journal of the ACM*, 51(5):800–849, sep 2004.

[26] G. Weikum and G. Vossen. *Transactional Information Systems: Theory, Algorithms, and the Practice of Concurrency Control and Recovery.* Morgan Kaufmann Publishers, 2002.