A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition

Damien Imbs, Michel Raynal

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1923

A VERSATILE STM PROTOCOL WITH INVISIBLE READ OPERATIONS THAT SATISFIES THE VIRTUAL WORLD CONSISTENCY CONDITION

DAMIEN IMBS MICHEL RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES
Campus de Beaulieu – 35042 Rennes Cedex – France
Tél. : (33) 02 99 84 71 00 – Fax : (33) 02 99 84 71 71
http://www.irisa.fr

A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition

Damien Imbs* Michel Raynal**

Systèmes communicants
Projet ASAP

Publication interne n ˚ 1923 — Février 2009 — 18 pages

Abstract: The aim of a Software Transactional Memory (STM) is to discharge the programmers from the management of synchronization in multiprocess programs that access concurrent objects. To that end, a STM system provides the programmer with the concept of a transaction. The job of the programmer is to decompose each sequential process the application is made up of into transactions. A transaction is a piece of code that accesses concurrent objects, but contains no explicit synchronization statement. It is the job of the underlying STM system to provide the illusion that each transaction appears as being executed atomically. For efficiency, a STM system allows transactions to execute concurrently. Consequently, due to the underlying STM concurrency management, a transaction commits or aborts.

This paper first presents a new STM consistency condition, called *virtual world* consistency. This condition states that no transaction reads object values from an inconsistent global state. It is similar to opacity for the committed transactions but weaker for the aborted transactions. More precisely, it states that (1) the committed transactions can be totally ordered, and (2) the values read by each aborted transaction are consistent with respect to its causal past only. Hence, virtual world consistency is weaker than opacity while keeping its spirit. Then, assuming the objects shared by the processes are atomic read/write objects, the paper presents a STM protocol that ensures virtual world consistency (while guaranteeing the invisibility of the read operations). From an operational point of view, this protocol is based on a vector-clock mechanism. Finally, the paper considers the case where the shared objects are regular read/write objects. It also shows how the protocol can be weakened to satisfy the *causal consistency* condition (that is weaker than virtual world consistency).

Virtual world consistency does not require the aborted transactions to agree on what they have seen. This is captured by the local vector clocks associated with each process and the vector timestamps associated with each object. From a comprehensive point of view, the paper addresses how the interplay of these local control informations allows the execution of a set of transactions to be provided with a global meaning.

Key-words: Atomic object, Causal past, Commit/abort, Concurrency control, Consistency condition, Consistent global state, Lock, Read-from relation, Regular Read/write object, Serializability, Shared memory, Software transactional memory, Vector clock, Transaction.

(Résumé : tsvp)

* IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France damien.imbs@irisa.fr
** IRISA, Université de Rennes 1, Campus de Beaulieu, 35042 Rennes Cedex, France, raynal@irisa.fr

Centre National de la Recherche Scientifique
(UMR 6074) Université de Rennes 1 – Insa de Rennes

Institut National de Recherche en Informatique
et en Automatique – unité de recherche de Rennes

Un protocole pour les mémoires transactionnelles qui satisfait la cohérence des mondes virtuels

Résumé : Ce rapport présente une protocole pour les mémoires transactionnelles logicielles qui satisfait le critère de cohérence des mondes virtuels. Ce critère de cohérence est plus faible que le critère d'opacité mais en garde l'esprit, à savoir il porte à la fois sur les transactions validées et sur les transactions avortées. Il est plus adapté aux mémoires transactionnelles que la simple sérialisabilité qui ne porte que sur les transactions validées.

Mots clés : Atomicité, Contrôle de la concurrence, Etat global cohérent, Horloge vectorielle, Mémoire transactionnelle, Object partagé, Opacité, Transaction, Validation, Verrou.

1 Introduction

The challenging advent of multicore architectures The speed of light has a limit. When combined with other physical and architectural demands, this physical constraint places limits on processor clocks: their speed is no longer rising. Hence, software performance can no longer be obtained by increasing CPU clock frequencies. To face this new challenge, (since a few years ago) manufacturers have investigated and are producing what they call *multicore architectures*, i.e., architectures in which each chip is made up of several processors that share a common memory. This constitutes what is called "the multicore revolution" [12].

The main challenge associated with multicore architectures is "how to exploit their power?" Of course, the old (classical) "multi-process programming" (multi-threading) methods are an answer to this question. Basically, these methods provide the programmers with the concept of a *lock*. According to the abstraction level considered, this lock can be a semaphore object, a monitor object, or more generally the base synchronization object provided by the underlying programming language.

Unfortunately, traditional lock-based solutions have inherent drawbacks. On one side, if the set of data whose accesses are controlled by a single lock is too large (large grain), the parallelism can be drastically reduced. On another side, the solutions where a lock is associated with each datum (fine grain), are error-prone (possible presence of subtle deadlocks), difficult to design, master and prove correct. In other words, providing the application programmers with locks is far from being the panacea when one has to produce correct and efficient multi-process (multi-thread) programs. Interestingly enough, multicore architectures have (in some sense) rang the revival of concurrent programming.

The Software Transactional Memory approach The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory has first been proposed (fifteen years ago) by Herlihy and Moss to implement concurrent data structures [13]. It has then been implemented in software by Shavit and Touitou [23], and has recently gained a great momentum as a promising alternative to locks in concurrent programming, e.g., [9, 11].

Transactional memory abstracts the complexity associated with concurrent accesses to shared data by replacing locking with atomic execution units. In that way, the programmer has to focus where atomicity is required and not on the way it has to be realized. The aim of a STM system is consequently to discharge the programmer from the direct management of synchronization entailed by accesses to concurrent objects.

More generally, STM is a middleware approach that provides the programmers with the *transaction* concept (this concept is close but different from the notion of transactions encountered in databases [9]). More precisely, a process is designed as (or decomposed into) a sequence of transactions, each transaction being a piece of code that, while accessing any number of shared objects, always appears as being executed atomically. The job of the programmer is only to define the units of computation that are the transactions. He does not have to worry about the fact that the base objects can be concurrently accessed by transactions. Except when he defines the beginning and the end of a transaction, the programmer is not concerned by synchronization. It is the job of the STM system to ensure that transactions execute as if they were atomic.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is irrelevant from an efficiency point of view. So, a STM system has to do "its best" to execute as many transactions per time unit as possible. Similarly to a scheduler, a STM system is an on-line algorithm that does not know the future. If the STM is not trivial (i.e., it allows several transactions that access the same objects in a conflicting manner to run concurrently), this intrinsic limitation can direct it to abort some transactions in order to ensure both transaction atomicity and object consistency. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered as a new transaction).

Content of the paper and roadmap This paper is made up of 5 sections and has three contributions. Section 2 presents the computation model and the first contribution, namely, a new consistency condition, called *virtual world* consistency. Differently from serializability but similarly to opacity, this condition (1) takes into account both the committed transactions and the aborted transactions, but (2) is strictly weaker than opacity (and can consequently allow more transactions to commit). Intuitively, both opacity and virtual world consistency requires that every transaction (whatever its fate, commit or abort) reads object values from a consistent global state. They differ in what each considers as a *consistent* global state.

The second contribution, namely, a STM protocol that satisfies virtual world consistency, is presented in Section 3. Among its noteworthy features, this protocol allows invisible read operations (i.e., when a transaction reads an object, it is not required to write control information into the shared memory to inform the other transactions on possible read/write conflicts). From an operational point of view, the protocol does not use a global logical clock, but a distributed vector clock with one entry per object. So, the protocol is targeted for applications that manipulate few shared objects.

Then, Section 4 addresses the versatility of the proposed STM protocol (third contribution). It shows that the simple suppression of a consistency check provides a protocol that ensures the *causal consistency* condition. It also shows that the addition of a single consistency check allows to replace the atomic objects shared by the processes by regular objects. Finally, Section 5 concludes the paper.

2 A STM Computation model

2.1 Why a consistency condition has to take into account the aborted transactions

The classical consistency criterion for database transactions is serializability [20] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that commit. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting. In a STM system, the code encapsulated in a transaction can be any piece of code (involving shared data), it is not restricted to predefined patterns. Consequently a transaction always has to operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b-c$ is different from 0 in all the consistent states. If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b=c$ (and $b=c$ defines an inconsistent state). If this occurs, the transaction raises an exception that has to be handled by the process that invoked the corresponding transaction. (Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops.) Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction has to always see a consistent state of the data it accesses. The aborted transactions have to be harmless. This observation has first been stated in [8].

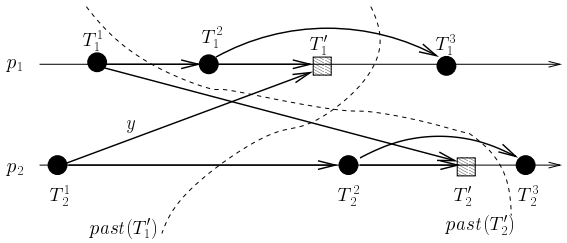
2.2 From opacity to virtual world consistency

Opacity Informally suggested in [8], and formally introduced and investigated in [10], the *opacity* consistency condition requires that no transaction reads values from an inconsistent global state where a *consistent global state* is defined as the state of the shared memory at some real time instant. Opacity is the same as strict serializability when we consider all the committed transactions, plus an appropriate read prefix for each aborted transaction.

More precisely, let us associate with each aborted transaction T the read prefix that contains all its read operations until T aborts (if the abort is entailed by a read, this read is not included in the prefix). An execution of a set of transactions satisfies the *opacity* condition if all the committed transactions plus the read prefix of each aborted transaction appear as if they have been executed one after the other (this is a "witness sequential execution"), this witness sequential execution being in agreement with the real time occurrence order of each transaction. (Examples of protocols implementing the opacity property -each with different additional features- can be found in [8, 15, 17, 22].)

Virtual world consistency This consistency condition is weaker than opacity while keeping its spirit. It states that (1) no transaction (committed or aborted) reads values from an inconsistent global state, (2) the consistent global states read by the committed transactions are mutually consistent (in the sense that they can be totally ordered) but (3) while the global state read by each aborted transaction is consistent from its individual point of view, the global states read by any two aborted transactions are not required to be mutually consistent. Said differently, virtual world consistency requires that (1) all the committed transactions be serializable [20] (so they all have the same "witness sequential execution") or linearizable [14] (if we want this witness execution to also respect real time) and (2) each aborted transaction (reduced to a read prefix as explained previously) reads values that are consistent with respect to its causal past only. As two aborted transactions can have different causal pasts, each can read from a global state that is consistent from its causal past point of view, but these two global states can be mutually inconsistent as aborted transactions have not necessarily the same causal past (hence the name *virtual world* consistency). This consistency condition can benefit lots of STM applications as, from its local point of view, a transaction cannot differentiate it from opacity.

A formal definition of virtual world consistency is presented in [18] and in the appendix of this paper. To make its intuition more precise, let us consider the transaction execution depicted on the right. There are two processes: p_i has sequentially issued T_1^1, T_1^2, T_1' and T_1^3, while p_2 has issued T_2^1, T_2^2, T_2' and T_2^3. The transactions associated with a black dot have committed, while the ones with a grey square have aborted. From a dependency point of view, each transaction issued by a process depends on its previous committed transactions, and on committed transactions issued by the other process as defined by the read-from relation due to the accesses to the shared objects, (e.g., the label y on the dependency edge from T_2^1 to T_1' means that T_1' has read a value that was written in y by T_2^1). Differently, since an aborted transaction does not write shared objects, there is no dependency edges originating from it. The causal past of the aborted transactions T_1' and T_2' are indicated on the figure. Virtual world consistency requires the following: (1) the committed transactions are serializable (or strict serializable if we want the witness sequence to respect the additional real time order constraint), and (2) each aborted transaction reads values from a state consistent with respect to its causal past (as an example, the values read by T_1' are consistent wrt the dependencies as indicated on the figure).



That consistency condition actually extends to STM systems the notions of *consistent cut*, *causal past*, and *consistent global state* encountered in asynchronous message-passing systems [5, 7, 24]. In these systems, two different processes can simultaneously compute two global states such that each global state is consistent with respect to the causal past of the invoking process, but these global states are mutually inconsistent from the point of view of an external omniscient observer (i.e., they cannot be serialized). The "read-from" relation linking transactions is the STM equivalent of the "message" relation that defines the flow of information exchange in message-passing systems.

In addition to the fact that it can allow more transactions to commit than opacity, one of the main interests of virtual world consistency lies in the fact that it prevents bad phenomena (as described in Section 2.1) from occurring without requiring all the transactions (committed or aborted) to agree on the same witness execution. Let us assume that, when executed alone and it reads a consistent state of the objects, each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.). As, due to the virtual world consistency condition, no transaction (committed or aborted) reads from an inconsistent state, it cannot behave incorrectly despite concurrency; it can only be aborted. This is a first class requirement for transactional memories.

2.3 The STM system interface

The STM system provides the transactions with four operations denoted $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$, where T is a transaction, and X a shared base object.

- $\mathsf{begin}_T()$ is invoked by T when it starts. It initializes local control variables.
- $X.\mathsf{read}_T()$ is invoked by the transaction T to read the base object X. That operation returns a value of X or the control value $abort$. If $abort$ is returned, the invoking transaction is aborted (in that case, the corresponding read does not belong to the read prefix associated with T).
- $X.\mathsf{write}_T(v)$ is invoked by the transaction T to update X to the new value v. That operation returns the control value ok or the control value $abort$. In the proposed protocol it always returns ok.
- If a transaction attains its last statement (as defined by the user, which means it has not been aborted before) it executes the operation $\mathsf{try\_to\_commit}_T()$. That operation decides the fate of T by returning $commit$ or $abort$. (Let us notice, a transaction T that invokes $\mathsf{try\_to\_commit}_T()$ has not been aborted during an invocation of $X.\mathsf{read}_T()$.)

2.4 The incremental read + deferred update model

In this transaction system model, each transaction T uses a local working space. When T invokes $X.\mathsf{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later $X.\mathsf{read}_T()$ invocations (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory can have changed in between.

When T invokes $X.\mathsf{write}_T(v)$, it writes v into its working space (and does not access the shared memory). Finally, if T is not aborted while it is executing $\mathsf{try\_to\_commit}_T()$, it copies the values written (if any) from its local working space to the shared memory. (A similar deferred update model is used in some database transaction systems.)

3 A STM protocol when the base objects are atomic

3.1 Processes and atomic base objects

The system is made up of an arbitrary number of processes and m base shared objects. The processes are denoted p_i, p_j, etc., while the objects are denoted $X, Y, \ldots$, where each id X is such that $X \in \{1, \cdots, m\}$. Each process is decomposed in a sequence of transactions (that are not known in advance).

Each of the m base objects is an atomic read/write object [19]. This means that the read and write operations issued on such an object X appear as if they have been executed sequentially, and this "witness sequence" is legal (a read returns the value written by the closest write that precedes it in this sequence)

and respects the real time occurrence order on the operations on X (if $op1(X)$ is terminated before $op2(X)$ starts, $op1$ appears before $op2$ in the witness sequence).

3.2 The STM algorithm: control variables

On a base object side Each base atomic object X is made up of two fields: $X.value$ which contains its value, and a vector $X.depend[1..m]$ that tracks value dependencies. More precisely, $X.depend[X]$ is the sequence number of the current value of X, while $X.depend[Y]$ ($Y \neq X$) is the sequence number of the value of Y on which the current value of X depends. (A sequence number can be seen as a logical date associated with an object.) Moreover a lock is associated with every base object.

On a process side A process issues transactions sequentially. So, when a process p_i issues a new transaction, that transaction has to work with object values that are not older than the ones used by the previous transactions issued by p_i. To that end, p_i manages a local vector $p\_depend_i[1..m]$ such that $p\_depend_i[X]$ contains the sequence number of the last value of X that (directly or indirectly) is known by p_i.

In addition to the previous array whose scope is the lifetime of the corresponding process, a process p_i manages local variables whose scope is the one of its current transaction T. Those are:
- An array $t\_depend_T[1..m]$ that is used instead of $p\_depend_i[1..m]$ during the execution of T. This is necessary because $p\_depend_i[1..m]$ must not be modified if T aborts,
- A set lrs_T (resp., lws_T) that is the read set (resp., write set) of the transaction T currently executed by p_i,
- Finally, for every object X accessed by T, p_i keeps a local copy that is denoted $lc(X)$.

3.3 The STM algorithm

The code of the STM system for a process p_i is described in Figure 1. It consists in the algorithms that implement the four operations of the STM interface (Section 2.3), namely, $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$, where T is a transaction issued by a process p_i and X a base object. When it is returned, the control value $abort$ is tagged 1 or 2 to indicate the cause of the abort to the corresponding transaction.

The operation $\mathsf{begin}_T()$ This operation is a simple initialization of the local control variables associated with the current transaction T. Let us notice that t_depend_T is initialized to p_depend_i to take into account the causal dependencies on the values previously accessed by p_i. This is due to the fact that a process p_i issues its transactions one after the other and the next one inherits the causal dependencies created by the previous ones.

The operation $X.\mathsf{read}_T()$ This operation returns a value of X or the control value $abort$ (in which case T is aborted). If (due to a previous read of X) there is a local copy, its value is returned (lines 01 and 07).

If $X.\mathsf{read}_T()$ is its first read of X, p_i first builds a copy $lc(X)$ from the shared memory (line 02), and updates accordingly its local control variables lrs_T and $t\_depend_T[X]$ (line 03).

As the reads are incremental (p_i does not read in one atomic action all the base objects it wants to read), p_i has to check that the value $lc(X).value$ it has just obtained from the shared memory does not make one of its previous reads inconsistent (in which case p_i has to abort T, line 04). Let Y be an object that has been previously read by T. Let us observe that the sequence number of the value of Y read by T is kept in $t\_depend_T[Y]$. If the value of X just read by T depends on a more recent value of Y, the values of X and Y are mutually inconsistent. This is exactly what is captured by the predicate $\exists Y \in lrs_T : t\_depend_T[Y] < lc(X).depend[Y])$ (line 04). If this predicate is true, p_i aborts T. Otherwise, p_i

```
operation begin_T(): lrs_T ← ∅; lws_T ← ∅; t_depend_T ← p_depend_i.
  ==========================================================================
operation X.read_T():
(01) if (there is no local copy of X) then
(02)    allocate local space -denoted lc(X)- for a local copy of X; lc(X) ← X;
(03)    lrs_T ← lrs_T ∪ {X}; t_depend_T[X] ← lc(X).depend[X];
(04)    if (∃Y ∈ lrs_T : t_depend_T[Y] < lc(X).depend[Y]) then return(abort, 1) end if;
(05)    for each Y ∉ lrs_T do t_depend_T[Y] ← max(t_depend_T[Y], lc(X).depend[Y]) end for
(06) end if;
(07) return (lc(X).value).
  ==========================================================================
operation X.write_T(v):
(08) if (there is no local copy of X) then allocate local space lc(X) to store v end if;
(09) lc(X).value ← v; lws_T ← lws_T ∪ {X};return (ok).
  ==========================================================================
operation try_to_commit_T():
(10) let ConsistencyCheck_T be the predicate (∀ Z ∈ lrs_T : t_depend_T[Z] = Z.depend[Z]);
(11) lock all the objects in lrs_T ∪ lws_T;
(12) if (lrs_T ≠ ∅) then if (¬ ConsistencyCheck_T) then release all the locks; return(abort, 2) end if end if;
(13) if (lws_T ≠ ∅) then for each X ∈ lws_T do t_depend_T[X] ← X.depend[X] + 1 end for;
(14)                    for each X ∈ lws_T do X ← (lc(X).value, t_depend_T) end for
(15) end if;
(16) release all the locks;
(17) p_depend_i ← t_depend_T;
(18) return(commit).
```

Figure 1: A STM algorithm that satisfies virtual world consistency

first updates $t\_depend_T[1..m]$ (line 05) to take into account the new dependencies (if any) created by this reading of X, and finally returns the value obtained from X (line 07).

A $X.\mathsf{read}_T()$ operation is *visible* if the issuing transaction T has to write the shared memory to inform the other transactions on its read of X. Otherwise it is *invisible*.

Property 1 *All the $X.\mathsf{read}_T()$ operations are invisible.*

Property 2 *If $(abort, 1)$ is returned to a transaction T, this is because T executes an operation $X.\mathsf{read}_T()$, and the abort is due to the fact that, while the values previously read by T define a consistent snapshot, the addition of the value of X obtained from the shared memory would make this snapshot inconsistent.*

In the case of Property 2, the read prefix associated with the aborted transaction T contains the values read before the operation $X.\mathsf{read}_T()$, and does not contain the value read from X.

The operation $X.\mathsf{write}_T(v)$ The algorithm implementing that operation is very simple. If there is no local copy for the object X, one is created (line 08). Then, the value v is written into that copy and the control variable lws_T is updated (line 09).

Property 3 *No $X.\mathsf{write}_T()$ operation can entail the abort of a transaction.*

The operation $\mathsf{try\_to\_commit}_T()$ The transaction T locks all the objects it has accessed (they are the objects in $lrs_T \cup lws_T$, line 11). The locking is done according to a canonical order to prevent deadlocks. If it is a read-only transaction (that has read more than one object), it can be committed if its incremental snapshot is still valid, i.e., the values it has read from the shared memory have not yet been overwritten. This is exactly

what is captured by the predicate $ConsistencyCheck_T$ (defined at line 10 and used at line 12). If this predicate is true, the transaction appears as if it was atomically executed just before the predicate evaluation. The transaction is then committed. If the predicate is false, there is no way to known if the transaction could be correctly serialized with respect to the committed transactions; it is consequently aborted (line 12).

If the transaction T is write-only (i.e., $lrs_T = \emptyset$, line 12), due to the locks on the objects of lws_T, the transaction T can atomically write their new values into the shared memory (line 14). Before these writes, T has to update the sequence number of each object X it writes so that the dependency vectors (vector timestamps) have correct values (line 13).

If the transaction T is neither read-only, nor write-only, it can be committed only if all its read and write operations could have been executed atomically. As just seen, the locks ensure that the writes appear as being executed atomically. For the read to appear as being executed atomically with the write of the new values in the shared memory, the predicate $ConsistencyCheck_T$ is evaluated once the locks on the objects in $lrs_T \cup lws_T$ have been acquired. If it is evaluated to true, the transaction appears as being executed atomically after the locks have been acquired and consequently the transaction T can be committed. Otherwise it is aborted (line 12).

Let us finally observe that, if a transaction is committed (line 18), the dependency vector of the process p_i has to be updated accordingly (line 17) to take into account the new dependencies created by the newly committed transaction T.

Property 4 *If $(abort, 2)$ is returned to a read-only transaction T, the values it has incrementally read define a consistent snapshot, but this snapshot cannot be serialized (with certainty) with respect to the committed transactions.*

Property 5 *If $(abort, 2)$ is returned to a read/write transaction T, the values it has incrementally read define a consistent snapshot, but this snapshot and the writes into the shared memory cannot appear as being executed atomically.*

In the case of the properties 4 and 5, all the read operations issued by the aborted transaction T belong to its read prefix, and this read prefix is consistent with respect to the causal past of T.

Property 6 *A write-only transaction cannot be aborted.*

Definition 1 *Two transactions $T1$ and $T2$ are independent if $(lrs_{T1} \cup lws_{T1}) \cap (lrs_{T2} \cup lws_{T2}) = \emptyset$.*

Property 7 *Independent transactions can commit concurrently.*

Remark A simple modification of the previous protocol provides us with the following additional property: a read-only transaction T that reads a single object X is never aborted. T is then only made up of $X.\mathsf{read}_T()$, and this operation is implemented as follows:

if (there is no local copy of X) **then**
allocate local space -denoted $lc(X)$- for a local copy of X; lock(X); $lc(X) \leftarrow X$; unlock(X) **end if**;
$\mathsf{return}(lc(X).value)$.

3.4 Properties of the protocol

Proof The previous section has stated a few properties whose aim is to give a better intuition of what the algorithms described in Figure 1 do and how they do it. The proof that they satisfy the virtual consistency condition requires a formal statement of that condition. This formal statement and the proof are presented in the appendix. The committed transactions can be linearized, and the appropriate read prefixes of each aborted transaction are consistent wrt their causal past.

Cost It is easy to see that the following values are upper bounds on the number of shared memory accesses issued by a transaction: $2|lrs_T|$ if T is read-only (lines 02 and 12), $2|lws_T|$ if T is write-only (lines 13 and 14), and $2|lrs_T| + 2|lwt_T|$ if T is a read/write transaction.

There is the additional cost due to locking/unlocking of base objects (lines 12 and 16). For the objects that are written this cost can be eliminated by placing the lock inside the object and (as in TL2 [8]) aborting a transaction when it accesses an object that is locked.

4 Versatility dimension of protocol

4.1 From virtual world consistency to causal consistency

Causally consistent transactions The concept of *causal consistency* for read/write objects has been introduced in [2] under the name *causal memory*. It has then been extended to transactions in [21] where only the committed transactions are considered. As for virtual world consistency, we extend here causal consistency to include the appropriate prefixes of the aborted transactions.

Intuitively, given an execution of a set of transactions issued by sequential processes, causal consistency allows each process to see its own "witness sequential execution" as long as these witness sequential executions respect the causal dependencies defined by the read-from relation.

More precisely, let $\mathcal{C}$ be the set of all the committed transactions that write base objects (whatever the issuing processes). For each process p_i, let $\mathcal{R}_i$ be the set of its committed read-only transactions plus its aborted transactions reduced to their read prefix (as defined previously in the paper). Causal consistency requires that, for each process p_i, there is a "witness sequential execution" involving only the transactions in $\mathcal{C} \cup \mathcal{R}_i$. Let us notice that all these witness sequential executions share the constraint imposed by the read-from relation as exhibited in $\mathcal{C}$.

Adapting the protocol The base protocol described in Figure 1 can be adapted very easily (weakened) to implement causal consistency: the single modification consists in inserting the following statement between lines 10 and 11:

$$\textbf{if } lws_T = \emptyset \textbf{ then } \mathsf{return}(commit) \textbf{ end if};$$

This modification does not alter the protocol for the aborted transactions whose abort is tagged 1 (line 04). As we have seen, the read prefix of such a transaction defines a consistent snapshot of the values previously read. It is now the same for a read-only transaction that does not abort at line 04. This is because the lines 11-16 are used to ensure that the consistent snapshot of the values read by the read-only transaction T belongs to the witness sequential execution including all the committed transactions. But, causal consistency does not impose this strong requirement: the values read by a read-only transaction have only to be mutually consistent (and consequently such a transaction can never return $(abort, 2)$ when one is interested in the weaker condition that is causal consistency).

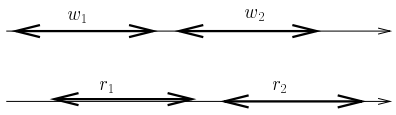
This shows that causal consistency weakens virtual world consistency by allowing a read-only transaction to commit as long as its snapshot of read values is consistent (as the prefix of an aborted transaction), without requiring that this snapshot be totally ordered with respect to all the committed transactions. The snapshot only has to be consistent with respect to the causal past of the read-only transaction.

4.2 From atomic objects to regular objects

Regular read/write object A *regular* read/write object [19] can have any number of writers and any number of readers. The writes appear as if they were executed sequentially, this sequence complying with

their real time order (i.e., if two writes w_1 and w_2 are concurrent they can appear in any order, but if w_1 terminates before w_2 starts, w_1 has to appear as being executed before w_2).

As far as a read operation is concerned we have the following. If no write operation is concurrent with a read operation, that read operation returns the current value kept in the object. Otherwise, the read operation returns any value written by a concurrent write operation or the last value of the object before these concurrent writes. A regular object can exhibit what is called a *new/old inversion*. The figure on the right depicts two write operations w_1 and w_2 and two read operations r_1 and r_2 that are concurrent (r_1 is concurrent with w_1 and w_2, while r_2 is concurrent with w_2 only). According to the definition of regularity, it is possible that r_1 returns the value written by w_2 while r_2 returns the value written by w_1.



An atomic read/write object is a regular read/write object without new/old inversion. This means that an atomic read/write object is such that all its read and write operations appear as if they have been executed sequentially, this total order respecting the real time order of the operations.

Adapting the protocol If the base objects are regular, we have to prevent new/old inversion so that they appear as if they were atomic. This can be obtained by adding a statement and modifying a predicate. More precisely the following modifications allow us to replace the base atomic read/write objects by weaker regular read/write objects.

- Line 03 is enriched by a test that prevents from reading an old value. That line becomes (the new statement is the **if** statement):
  $lrs_T \leftarrow lrs_T \cup \{X\}$;
  if $(t\_depend_T[X] > lc(X).depend_T[X])$ **then** return$(abort, 4)$ **end if**;
  $t\_depend_T[X] \leftarrow lc(X).depend[X]$.
- The predicate $ConsistencyCheck_T$ is now defined as $(\forall\ Z \in lrs_T : t\_depend_T[Z] \geq Z.depend[Z])$.

Property 8 *If the invocation of $X.\mathsf{read}_T()$ by T returns $(abort, 4)$, the abort is due to a new/old inversion.*

4.3 When the base objects are neither atomic nor regular

When the base objects are neither atomic nor regular, there is a very simple way to enrich the protocol of Figure 1 to make it work correctly. In order to make a base object X atomic, it is sufficient to use the lock associated with that object and replace the read of X from the shared memory at line 02 by "lock(X); $lc(X) \leftarrow X$; unlock(X)".

5 Conclusion

This paper has presented a new consistency condition called *virtual world* consistency [18], that is weaker than opacity while keeping its spirit. It has then presented a STM protocol with invisible read operations that implements this condition. This protocol, that is based on vector clocks that capture the causal dependencies among the values of the objects, presents an interesting versatility feature. The suppression of a consistency test provides a protocol satisfying the *causal consistency* condition (that is weaker than virtual world consistency), while the appropriate addition of a simple consistency test allows us to replace the base atomic objects by (weaker) regular objects.

The proposed STM protocol is targeted for applications where the processes share a "reasonable" number of base objects. This is in order to have small size vector clocks. When the application processes share

a large number of objects, it is possible to have small size vector clocks by requiring sets of objects to share the same entry of the vector clock as it is done in the "plausible vector clocks" [25]. In that case, no causal dependency is lost, but additional "false" dependencies can be witnessed by a vector clock. This is due to the fact that several objects share the same entry of the vector clock. The benefit of using such vector clocks the size k of which is bounded and much smaller than m (the number of shared objects) has a price: due to the false additional dependencies, more transactions can be aborted. (Let us remark that the objects that share the same vector clock entry also have to share the same lock.)

Finally, let us notice that both the *virtual world consistency* condition and the associated vector clock-based protocol offer an additional insight on STM systems, that participate in providing a better understanding of their underlying basic principles [3].

References

[1] Afek Y., Attiya H., Dolev D., Gafni E., Merritt M. and Shavit N., Atomic Snapshots of Shared Memory. *Journal of the ACM*, 40(4):873-890, 1993.

[2] Ahamad M., Neiger G., Burns J.E., Kohli P. and Hutto Ph.W., Causal Memory: Definitions, Implementation, and Programming. *Distributed Computing*, 9(1):37-49, 1995.

[3] Attiya H., Needed: Foundations for Transactional Memory. *ACM Sigact News, DC Column*, 39(1):59-61, 2008.

[4] Attiya H., Guerraoui R. and Ruppert E., Partial Snapshot Objects. *Proc. 20th ACM Symposium on Parallel Algorithms and Architectures (SPAA'08)*, ACP Press, ACM Press, pp. 336-343, 2008.

[5] Babaoğlu Ö. and Marzullo K., Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms. Chapter 4 in "Distributed Systems". ACM Press, Frontier Series, pp 55-93, 1993.

[6] Bernstein Ph.A., Shipman D.W. and Wong W.S., Formal Aspects of Serializability in Database Concurrency Control. *IEEE Transactions on Software Engineering*, SE-5(3):203-216, 1979.

[7] Chandy K.M. and Lamport L., Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Operating Systems*, 3(1):63-75, 1985.

[8] Dice D., Shalev O. and Shavit N., Transactional Locking II. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag, LNCS #4167, pp. 194-208, 2006.

[9] Felber P., Fetzer Ch., Guerraoui R. and Harris T., Transactions are coming Back, but Are They The Same? *ACM Sigact News, DC Column*, 39(1):48-58, 2008.

[10] Guerraoui R. and Kapałka M., On the Correctness of Transactional Memory. *Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, ACM Press, pp. 175-184, 2008.

[11] Harris T., Cristal A., Unsal O.S., Ayguade E., Gagliardi F., Smith B. and Valero M., Transactional Memory: an Overview. *IEEE Micro*, 27(3):8-29, 2007.

[12] Herlihy M.P. and Luchangco V., Distributed Computing and the Multicore Revolution. *ACM SIGACT News, DC Column*, 39(1): 62-72, 2008.

[13] Herlihy M.P. and Moss J.E.B., Transactional Memory: Architectural Support for Lock-free Data Structures. *Proc. 20th ACM Int'l Symp. on Comp. Arch. (ISCA'93)*, pp. 289-300, 1993.

[14] Herlihy M.P. and Wing J.M., Linearizability: a Correctness Condition for Concurrent Objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463-492, 1990.

[15] Imbs D. and Raynal M., A Lock-based STM Protocol that Satisfies Opacity and Progressiveness. *12th Int'l Conf. On Principles Of Distributed Systems (OPODIS'08)*, Springer-Verlag LNCS #5401, pp. 226-245, 2008.

[16] Imbs D. and Raynal M., Help When Needed, but No More: Efficient Read/Write Partial Snapshots. *Tech Report*, #1907, 26 pages, IRISA, Université de Rennes (France), 2008.

[17] Imbs D. and Raynal M., Provable STM Properties: Leveraging Clock and Locks to Favor Commit and Early Abort. *Proc. 10th Int'l Conference on Distributed Computing and Networking (ICDCN'09)*, Springer-Verlag, LNCS #5408, pp. 67-78, 2009.

[18] Imbs D. and Raynal M., On the Consistency Conditions of Transactional Memories. *Tech Report #1917*, 23 pages, IRISA, Université de Rennes (France), 2009.

[19] Lamport L., On interprocess communication. *Distributed Computing*, 1(2):77-101, 1986.

[20] Papadimitriou Ch.H., The Serializability of Concurrent Updates. *Journal of the ACM*, 26(4):631-653, 1979.

[21] Raynal M., Thia-kime G. and Ahamad M., From serializable to causal transactions. *BA. Proc. 20th ACM Symp. on Dist. Comp. (PODC'96)*, ACM Press, pp. 310, 1996. Full version: From serializable to causal transactions for collaborative applications. *Proc. 23th EUROMICRO Conference*, IEEE Computer Press, pp. 314-321, 1997.

[22] Riegel T., Fetzer C. and Felber P., Time-based Transactional Memory with Scalable Time Bases. *Proc. 19th annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'07)*, ACM Press, pp. 221-228, 2007

[23] Shavit N. and Touitou D., Software Transactional Memory. *Distributed Computing*, 10(2):99-116, 1997.

[24] Schwarz R. and Mattern F., Detecting Causal Relationship in Distributed Computations: in Search of the Holy Grail. *Distributed Computing*, 7:149-174, 1993.

[25] Torres-Rojas F. and Ahamad M., Plausible Clocks: Constant Size Logical Clocks for Distributed Systems. *Distributed Computing*, 12:179-195, 1999.

A Computation model and base definitions

This appendix is included in order to make this paper self-contained. This section is extracted from [18].

A.1 Processes and base objects

From an application point of view, a system is made up of a set of n processes $p_1, \ldots, p_n$, plus a set of base concurrent objects accessed by atomic read and write operations. There is no assumption on the respective speed of processes, except they are neither zero, nor infinite: the processes are *asynchronous*.

A.2 Transactions and base events

Transaction A transaction is a piece of code that is produced on-line by a sequential process (automaton), that is assumed to be executed atomically (commit) or not at all (abort). This means that (1) the transactions issued by a process are totally ordered, and (2) the designer of a transaction has not to worry about the management of the base objects accessed by the transaction. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or write any base object. Such a read or write access is atomic. The set of the objects read by a transaction defines its *read set*. Similarly the set of objects it writes defines its *write set*. A transaction that does not write base objects is a *read-only* transaction, otherwise it is an *update* transaction. A transaction that issues only write operations is a *write-only* transaction.

As in [6], we consider that the behavior of a transaction T can be decomposed in three sequential steps[1]: it first reads data objects, then does local computations and finally writes new values in some objects, which means that a transaction can be seen as a software read_modify_write() operation that is dynamically defined by a process[2]. The read set is defined incrementally, which means that a transaction reads the objects of its read set asynchronously one after the other (between two consecutive reads, the transaction can issue local computations that take arbitrary, but finite, durations). We say that the transaction T computes an *incremental snapshot*[3]. This snapshot has to be *consistent* which means that there is a time frame in which these values have co-existed (as we will see later, different consistency conditions consider different time frame notions). If it is about to read a new object whose current value would make inconsistent its current incremental snapshot, the transaction T is directed to abort. If it is not aborted during its read phase, T issues local computations. Finally, if T is an update transaction, and its write operations can be issued in such a way that T appears as being executed atomically, the objects of its write set are updated and T commits; otherwise, T is aborted. So, each aborted transaction is reduced to a read prefix. When, at the model level in the following, we speak about an aborted transaction, we implicitly refer to such a prefix. Independently of consistency reasons, a transaction T can also be aborted by the process that issued it. (From our point of view, namely the definition of *consistency conditions* for STM systems, we consider that such aborts include the case where transactions are aborted in order to improve the global efficiency[4].)

Events at the shared memory level Each transaction generates events defined as follows.

- Begin and end events. The event denoted B_T is associated with the beginning of the transaction T, while the event E_T is associated with its termination. E_T can be of two types, namely A_T and C_T, where A_T is the event "abort of T", while C_T is the event "commit of T".

- Read events. The event denoted $r_T(X)v$ is associated with the atomic read of X (from the shared memory) issued by the transaction T. The value v denotes the value returned by the read. If the value v, or T, is irrelevant $r_T(X)v$ is abbreviated $r_T(X)$, or $r(X)v$ or $r(X)$. The notation $r_T(X)v \in T$, or $r(X)v \in T$, or $r(X) \in T$, is used to express that $r_T(X)v$ is an event of T.

- Write events. The event denoted $w_T(X)v$ is associated with the atomic write of the value v in the shared object X (in the shared memory). If the value v is irrelevant $w_T(X)v$ is abbreviated $w_T(X)$. Without loss of generality we assume that no two writes on the same object X write the same value. We also assume that all the objects are initially written by a fictitious transaction. Similarly to the previous item, the notation $w_T(X)v \in T$, or $w(X)v \in T$, or $w(X) \in T$, is used to express that $w_T(X)v$ is an event of T.

At the shared memory level, only the events such as B_T, E_T, $r_T(X)v$ and $w_T(X)v$ are perceived. Let H be the set of all these events. Moreover, as $r_T(X)v$ and $w_T(X)v$ correspond to the execution of base atomic operations, the set of all the begin, end, read and write events can be totally ordered. This total order, denoted $\widehat{H} = (H, <_H)$, is called a *shared memory history*.

[1] This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model. It does not preclude an implementation where a transaction writes some objects before reading other objects. In that case, a transaction that aborts has to undo its previous writes.

[2] Different read_modify_write() operations are provided by some processors. Classical examples of such operations provided by hardware are the instructions test&set(), fetch&increment(), and compare&swap(). Their read set is equal to their write set, and contain a single atomic register. Moreover, their internal computation is defined once for all.

[3] The incremental approach to compute a snapshot reads asynchronously (separately) one object after the other. Differently, in [1, 4, 16], the whole set of the base objects to be atomically read is globally defined at the time of the snapshot invocation.

[4] This is the case for example in the system TL2 [8] where a transaction can be sacrificed (aborted) to increase the number of transactions that are committed per time unit. This occurs when a transaction tries to lock an object that is already locked.

A.3 Execution histories

Transaction history The execution of a set of transactions is represented by a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ that expresses a structural property of the execution of these transactions capturing the order of these transactions as issued by the processes and in agreement with the values they have read. More formally, we have:

- PO is the set of transactions, and
- $T1 \rightarrow_{PO} T2$ (we say "$T1$ precedes $T2$") if:
  1. (Process order.) Both $T1$ and $T2$ have been issued by the same process, and $T1$ is a committed transaction that has been issued before $T2$.
  2. (Read_from order.) $\exists\ w_{T1}(X)v\ \wedge\ \exists\ r_{T2}(X)v$. (There is an object X whose value written by $T1$ has been read by $T2$.)
  3. (Transitivity.) $\exists T : (T1 \rightarrow_{PO} T) \wedge\ (T \rightarrow_{PO} T2)$.

Remark When we look at the partial order $\widehat{PO}$, it is important to notice that, while all the committed transactions issued by a process are totally ordered, there is no precedence edge that originates from an aborted transaction. For the committed transactions issued by a process, this expresses the fact that those have been sequentially issued by that process and are possibly causally related. Roughly speaking, this total order defines what that process "really did". Differently, whatever the values read by an aborted transaction (a priori those can be mutually consistent or not), those values do not have to "causally" impact the future in a systematic way (except if a process voluntarily takes them into account in its next transaction).

As we can see, an important difference between classical (e.g., database) transactions and STM transactions lies in the fact that in a STM the transactions are issued by processes. (In a database, there is no notion of process that relates transactions.) Of course, in a STM system, it could be possible to ask a process to indicate which of its transactions are process-order related. This possibility would add flexibility (and could be relevant for some applications) but does not change fundamentally the process-based model previously introduced.

Independent transactions and sequential execution Given a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ that models a transaction execution, two transactions $T1$ and $T2$ are *independent* (or concurrent) if neither is ordered before the other: $\neg(T1 \rightarrow_{PO} T2)\ \wedge\ \neg(T2 \rightarrow_{PO} T1)$. An execution such that $\rightarrow_{PO}$ is a total order, is a *sequential* execution.

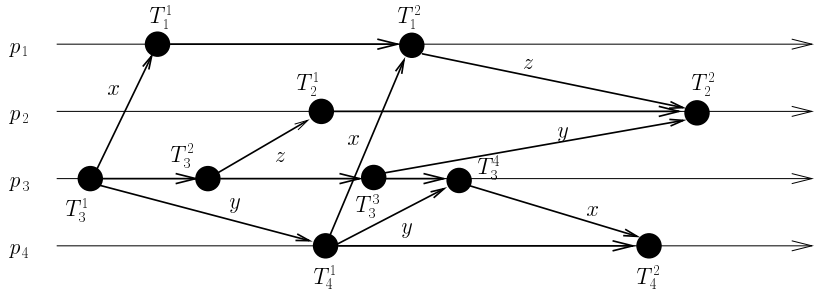


Figure 2: A partial order $\widehat{CH} = (CH, \rightarrow_{CH})$ (only committed transactions)

Committed transaction history A *committed transaction history* (in short c-history) is a partial order $\widehat{CH}$ as defined above where the set of transactions (denoted CH) is made up of all the committed transactions. Moreover, $\rightarrow_{PO}$ is then denoted $\rightarrow_{CH}$.

An example of such a partial order is described in Figure 2, where a committed transaction is depicted by a big black dot. The "time line" of each process is indicated with a slim long horizontal arrow. The precedence edges of the $\rightarrow_{PO}$ relation are indicated with black arrows. Assuming that the transactions access the base objects x, y and z, some read-from edges are indicated by labeled arrows where the label indicates the object written and read respectively by the endpoint transactions (the corresponding object values are not represented). Transitivity edges are not represented.

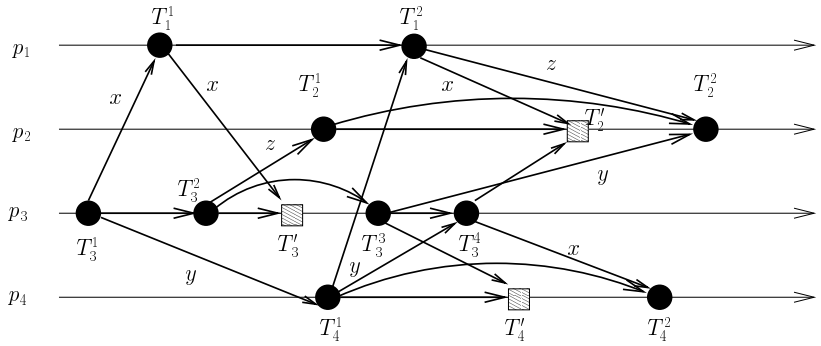


Figure 3: A partial order $\widehat{CAH} = (CAH, \rightarrow_{CAH})$ (committed and aborted transactions)

Complete transaction history A *complete transaction history* (in short ca-history) is a partial order $\widehat{CAH}$ as defined above where the set of transactions (denoted CAH) is made up of all the committed or aborted transactions. The order relation $\rightarrow_{PO}$ is denoted $\rightarrow_{CAH}$. Let us observe that $\rightarrow_{CH} \subseteq \rightarrow_{CAH}$.

Let T be an aborted transaction. If T reads, we have directed edges $T' \rightarrow_{CAH} T$ where T' is a committed transaction. Moreover, it follows from (1) the fact that an aborted transaction T does not write the shared memory, and (2) the definition of the process order relation, that there is no outgoing edge from an aborted transaction T.

Figure 3 describes a $\widehat{CAH}$ partial order in which the aborted transactions are depicted with squares (those are denoted T_2', T_3' and T_4'). When considering T_2', the figure shows that it reads two values one produced by T_1^2, the other by T_3^4. The arrow from T_2^1 to T_2' is a process order edge (and there is no process edge from T_2' to T_2^2).

A.4 Additional base definitions

Real time order Let $\rightarrow_{RT}$ be the *real time* relation defined as follows: $T1 \rightarrow_{RT} T2$ if E_{T1} occurs before B_{T2} ($E_{T1} <_H B_{T2}$). This relation (defined on the whole set of transactions, or only the committed transactions) is a partial order. In the particular case where it is a total order, we say that we have a real time-complying sequential execution.

Considering that the space/time diagrams depicted in the previous Figures 2 and 3 are real time diagrams, we see that $T_1^1 \rightarrow_{RT} T_3^4$, while the executions of T_2^1 and T_4^1 overlap in real time.

Linear extension A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ is a topological sort of this partial order, i.e., (1) $S = PO$ (same elements), (2) $\rightarrow_S$ is a total order, and (3) $(T1 \rightarrow_{PO} T2) \Rightarrow (T1 \rightarrow_S T2)$ (we say that $\rightarrow_S$ respects $\rightarrow_{PO}$).

As an example the sequence $T_3^1\ T_3^2\ T_2^1\ T_1^1\ T_4^1\ T_1^2\ T_3^3\ T_3^4\ T_2^2\ T_4^2$ is a linear extension of the partial order described in Figure 2. (Let us notice that this linear extension does not respect real time order.)

Legal transaction The notion of legality is crucial for defining a consistency condition. It expresses the fact that a transaction does not read an overwritten value. More formally, given a linear extension $\widehat{S}$, a transaction T is *legal* in $\widehat{S}$ if, for each $r_T(X)v \in T$, there is a committed transaction T' such that:

- • $T' \rightarrow_S T$ and $w_{T'}(X)v \in T'$, and
- • there is no transaction T'' s.t. $T' \rightarrow_S T'' \rightarrow_S T$ and $w_{T''}(X) \in T''$.

If all the transactions are legal, the linear extension $\widehat{S}$ is legal. In the following, a legal linear extension of a partial order, that models an execution of a set of transactions, is sometimes called a *sequential witness* (or witness) of that execution.

Causal past of a transaction Given a partial order $\widehat{PO}$ defined on a set of transactions, the *causal past* of a transaction T, denoted $past(T)$, is the set including T and all the transactions T' such that $T' \rightarrow_{PO} T$. Let us observe that, if T is an aborted transaction, it is the only aborted transaction contained in $past(T)$.

B Virtual world consistency

Real time or virtual time opacity requires that all the transactions (be them committed or aborted) see the same witness execution $\widehat{CAS}$ that complies with the (real or virtual) time notion considered. Weaker and meaningful consistency definitions that take into account aborted transactions are actually possible, and even desirable for STM systems. More precisely, we obtain the following family of consistency conditions.

- For the committed transactions: Either serializability or strict serializability can be considered.
- An aborted transaction T is *virtual world consistent* if there is a linear extension $\widehat{S_T}$ of the partial order $past(T)$ that is legal.

An execution of a set of transactions is *virtual world* (resp., *strong virtual world*) consistent if (1) all the committed transactions are serializable (resp., strict serializable), and (2) each aborted transaction is *virtual world* consistent.

Let us observe that the witness $\widehat{S_T}$ (from which T has been suppressed) is not required to be a prefix of the legal linear extension associated with the whole set of the committed transactions. It is easy to see that, while virtual world consistency is weaker than opacity, it remains a meaningful consistency condition as it requires that the object values read by each aborted transaction be mutually consistent.

The idea that underlies this family of consistency conditions is the following. It guarantees that, in addition to the committed transactions, every aborted transaction reads values from a consistent global state of the shared memory. This state is consistent in the sense that, for each aborted transaction T, it appears in some legal history that is a witness for T. This does not means that this state has really appeared in the shared memory; it only means that, from the point of view of the aborted transaction, the execution could have passed through this state. Hence, the name *virtual world* consistency. The important point is here that each of several aborted transactions $T1$ ($T2$, etc.), sees a consistent global state (from which it reads the values of the objects in its read set) as given by a linear extension $\widehat{S_{T1}}$ ($\widehat{S_{T2}}$, etc.): each witness linear extension represents a possible "virtual world" that can be different from the other witness linear extensions.

One of the main interests of virtual world consistency lies in the fact that it prevents bad phenomena from occurring without requiring all the transactions (committed or aborted) to agree on the same witness execution. Let us assume that, when executed alone and it reads a consistent state of the objects, each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.). As, due to the virtual world consistency condition, no transaction (committed or aborted) reads from an inconsistent state, it cannot behave incorrectly despite concurrency; it can only be aborted. This is a first class requirement for transactional memories.

C Proof of the protocol

C.1 Committed transactions are linearizable

In this section we prove that the committed transaction history $\widehat{CH} = (CH, \rightarrow_{CH})$ admits a legal linear extension. Let $\widehat{S} = (S, \rightarrow_S)$ be that extension, where $S = CH$ and $\rightarrow_S$ is a total order defined according to the linearization points of the transactions. The linearization point of a committed transaction T is placed just after it acquires all the locks on the objects it accesses (line 11).

In order to prove that $\widehat{S}$ is legal, we have to prove that

1. $\rightarrow_{CH} \subseteq \rightarrow_S$ (the total order $\rightarrow_S$ respects the partial order $\rightarrow_{CH}$),
2. $\forall T1, T2 \in S, \forall X : T1 \stackrel{X}{\rightarrow}_{rf} T2 \Rightarrow (\nexists T3 : T1 \rightarrow_S T3 \rightarrow_S T2 \wedge w(X) \in T3)$,
3. $\forall T1, T2 \in S, \forall X : T1 \stackrel{X}{\rightarrow}_{rf} T2 \Rightarrow T1 \rightarrow_S T2$, and
4. $\forall T1, T2 \in S : T1 \rightarrow_{RT} T2 \Rightarrow T1 \rightarrow_S T2$.

Let $AL_T(X)$ denote the event associated with the acquisition of the lock on the object X issued by the transaction T during an invocation of $\mathsf{try\_to\_commit}_T()$. Similarly, let $RL_T(X)$ denote the event associated with the release of the lock on the object X issued by the transaction T during an invocation of $\mathsf{try\_to\_commit}_T()$. Let us recall that, as $<_H$ (the shared memory history) is a total order, each event in H (including now $AL_T(X)$ and $RL_T(X)$) can be seen as a date of the time line. This "date" view of a sequential history on events will be used in the following proofs.

Lemma 1 $\rightarrow_{CH} \subseteq \rightarrow_S$.

Proof In order to prove that $\rightarrow_{CH} \subseteq \rightarrow_S$, we have to show that $\rightarrow_S$ respects the process order and the read-from relation. Transitivity is then obtained by the fact that $\rightarrow_S$ is a total order.

Process order The placement of the linearization points guarantees that process order is respected (they are placed during the lifetime of the transactions).

Read-from relation Consider two transactions $T1$ and $T2$ and an object X such that $T1 \stackrel{X}{\rightarrow}_{rf} T2$. We then have $w_{T1}(X) <_H r_{T2}(X)$. Because (1) the linearization point of $T1$ (line 11) is placed before it writes X (line 14), (2) $w_{T1}(X) <_H r_{T2}(X)$ and (3) the linearization point of $T2$ is placed after its read of X ($\mathsf{try\_to\_commit}_{T2}()$ is its last operation), the read-from relation is respected, which concludes the lemma. $\square_{Lemma\ 1}$

Lemma 2 $\forall T1, T2 \in S, \forall X : T1 \stackrel{X}{\rightarrow}_{rf} T2 \Rightarrow (\nexists T3 : T1 \rightarrow_S T3 \rightarrow_S T2 \wedge w(X) \in T3)$.

Proof This proof is by contradiction. Suppose such a $T3$ exists. We then have $w_{T1}(X) <_H w_{T3}(X)$ because of locking and of the placement of the linearization points. We also have $r_{T2}(X) <_H w_{T3}(X)$ because $T1 \xrightarrow{X}_{rf} T2$ (else $T3$ would read the value of X written by $T3$). Because $T3 \rightarrow_S T2$, we have $RL_{T3}(X) <_H AL_{T2}(X)$ which means that $T2$ should be aborted ($ConsistencyCheck$, line 14). Thus, such a $T3$ cannot exist, which concludes the lemma. $\square_{Lemma\ 2}$

Lemma 3 $\forall T1, T2 \in S, \forall X : T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \rightarrow_S T2$.

Proof We have $w(X)T1 <_H r(X)T2$ because $T1 \rightarrow_{rf} T2$. Because the commit of $T2$ can only be its last operation, we then have $w(X)T1 <_H r(X)T2 <_H AL_{T2}(X)$ and so $w(X)T1 <_H RL_{T1}(X) <_H AL_{T2}(X)$. From the definition of the linearization points we then have $T1 \rightarrow_S T2$ which concludes the proof of the lemma. $\square_{Lemma\ 3}$

Lemma 4 $\forall T1, T2 \in S : T1 \rightarrow_{RT} T2 \Rightarrow T1 \rightarrow_S T2$.

Proof The proof follows directly from the definition of the linearization points (they are placed during the lifetime of the transactions). $\square_{Lemma\ 4}$

C.2 Aborted transactions are virtual world consistent

In this section we prove that all aborted transactions are virtual world consistent, that is, they all read from consistent global states even though these global states do not have to be mutually consistent.

Definition 2 *Given a set S of transactions, we say that a subset S' of S is causally consistent if and only if* $\forall T \in S' : \{T' | T' \rightarrow_{PO} T\} \subseteq S'$.

Lemma 5 *If a set of transactions S admits a legal linear extension, then any causally consistent subset S' of S admits a legal linear extension.*

Proof Let $\widehat{S} = (S, \rightarrow_S)$ be the legal linear extension of S. Let $\rightarrow_{S'}$ be the relation $\rightarrow_S$ restricted to S'. In order to prove that $\widehat{S'} = (S', \rightarrow_{S'})$ is a legal linear extension of S', we have to prove that

1. $\forall T1, T2 \in S', \forall X : T1 \xrightarrow{X}_{rf} T2 \Rightarrow (\nexists T3 : T1 \rightarrow_{S'} T3 \rightarrow_{S'} T2 \wedge w(X) \in T3)$.
   The fact that such a $T3$ does not exist in S implies that it does not exist either in S'.
2. $\forall T1, T2 \in S' : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_{S'} T2$.
   From the facts that (1) $T1 \rightarrow_{rf} T2$, (2) $T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_S T2$ and (3) $\rightarrow_{S'}$ is derived from $\rightarrow_S$, we conclude that $T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_{S'} T2$, which concludes the proof of the lemma.

$\square_{Lemma\ 5}$

Lemma 6 *Given a transaction T, $past(T) \backslash \{T\}$ is a causally consistent subset of $\mathcal{C}$.*

Proof The proof follows directly from the definition of a causal consistent subset and from the construction of $past(T)$. $\square_{Lemma\ 6}$

For a committed transaction T and an object X, let $depend(X, T)$ be the value of $t\_depend_T[X]$ just before the release of the locks (line 18).

Lemma 7 $\forall T, T' \in \mathcal{C}, \forall X : T \rightarrow_{PO} T' \Rightarrow depend(X,T) \leq depend(X,T')$.

Proof The local variable t_depend_T is initialized in the $\mathsf{begin}_T()$ operation and can be modified in the $X.\mathsf{read}_T()$ and $\mathsf{try\_to\_commit}()$ operations.

$T \rightarrow_{PO} T'$ can be obtained in three ways:

- process order,
- read-from relation ($\rightarrow_{rf}$), and
- transitivity.

Process order Without loss of generality, we consider that T is the previous transaction committed by process i before the start of T'. $t\_depend_{T'}$ is initialized in the $\mathsf{begin}_{T'}()$ operation as p_depend_i. This implies that at the beginning of T', $\forall X, t\_depend_{T'}[X] = depend(X,T)$. Because $t\_depend_{T'}[X]$ can only grow during the transaction (line 03 if T' reads X's latest value, operation max line 05 if it doesn't and line 15 if it writes X), we obtain $\forall X : depend(X,T) \leq depend(X,T')$.

Read-from relation During a $Y.\mathsf{read}_{T'}()$ operation where Y's latest value has been written by T, T' updates each entry of $t\_depend_{T'}$. If $X = Y$, T has written X's latest value and so $t\_depend_{T'}[X]$ contains X's highest version number (line 03). If X has been read previously by T', if T's entry is higher than T''s, T' aborts (in order to avoid reading an inconsistent state, line 04). If X has not been read previously by T', T' updates $t\_depend_{T'}$ to T's entry only if it is higher than T''s previous value (line 05). Thus, we obtain $\forall X : depend(X,T) \leq depend(X,T')$.

Transitivity Let $T1 \rightarrow_i T2$ be the relation defined as: $T1$ and $T2$ have been issued by process i, and $T1$ precedes $T2$. We then have $\exists T'' : (T \rightarrow_i T'' \vee T \rightarrow_{rf} T'') \wedge T'' \rightarrow_{PO} T'$. From the previous reasonings, we have $\forall X : depend(X,T) \leq depend(X,T'')$. We then apply recursively the same inequality until $T'' \rightarrow_i T'$ or $T'' \rightarrow_{rf} T'$, which concludes the lemma. $\square_{Lemma\ 7}$

Lemma 8 *$\forall T \in \mathcal{A}, past(T)$ admits a legal linear extension.*

Proof Let $\widehat{T} = (past(T), \rightarrow_T)$ be that linear extension, where the total order $\rightarrow_T$ is defined as follows:

- $\forall T1, T2 \in past(T) \backslash \{T\} : T1 \rightarrow_S T2 \Rightarrow T1 \rightarrow_T T2$, and
- $\forall T' \in past(T) \backslash \{T\} : T' \rightarrow_T T$.

From Lemmas 5 and 6, $past(T) \backslash \{T\}$ admits a linear extension. Then, we only have to consider the cases involving T:

1. $\forall T1 \in past(T) \backslash \{T\}, \forall X : T1 \stackrel{X}{\rightarrow}_{rf} T \Rightarrow (\nexists T3 : T1 \rightarrow_T T3 \rightarrow_T T \wedge w(X) \in T3)$.

   This part of the proof is by contradiction. Suppose such a T3 exists. After the read of X by T, we have $t\_depend_T[X] = depend(X,T1)$ (line 03). Because $T1 \stackrel{X}{\rightarrow}_{rf} T$, we have $r(X)T <_H w(X)T3$ so T and $T3$ are concurrent. By the definition of $\rightarrow_T$, $T3$ commits after $T1$ and so, according to line 13, we have $depend(X,T1) < depend(X,T3)$. From Lemma 7 and line 04, any read of a value written by $T3$ or by a transaction $T4$ such that $T3 \rightarrow_{PO} T4$ would then be prohibited, which proves that such a $T3$ cannot exist.

2. $\forall T1 \in past(T) \backslash \{T\} : T1 \rightarrow_{rf} T \Rightarrow T1 \rightarrow_T T$.

   This follows directly from the definition of $\rightarrow_T$, and concludes the lemma.

$\square_{Lemma\ 8}$

Theorem 1 *The algorithm presented in Figure 1 satisfies strong virtual world consistency.*

Proof Lemmas 1, 2, 3 and 4 prove that the protocol satisfies linearizability for committed transactions. Lemma 8 proves that it satisfies virtual world consistency for aborted transactions. $\square_{Theorem\ 1}$