Theoretical Computer Science 444 (2012) 113–127

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Virtual world consistency: A condition for STM systems (with a versatile protocol with invisible read operations)

Damien Imbs *, Michel Raynal

IRISA, Université de Rennes 1, 35042 Rennes, France

A R T I C L E   I N F O

Keywords:
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

A B S T R A C T

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, an STM system provides the programmer with the concept of a transaction. The job of the programmer is to design each process the application is made up of as a sequence of 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. Of course, for efficiency, an STM system has to allow 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. 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 an 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 easily be weakened while still providing an STM system that satisfies *causal consistency*, a condition strictly weaker than virtual world consistency.

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 cannot be further incremented. 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 carried out investigation 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" [13].

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

* Corresponding author.
E-mail addresses: damien.imbs@irisa.fr (D. Imbs), raynal@irisa.fr (M. Raynal).

0304-3975/$ – see front matter © 2012 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2012.04.037

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 (coarse grained), the parallelism can be drastically reduced. On another side, the solutions where a lock is associated with each datum (fine grained) 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 [14]. It has then been implemented in software by Shavit and Touitou [26], and has recently gained a great momentum as a promising alternative to locks in concurrent programming, e.g., [10,12,20].

Transactional memory abstracts the complexity associated with concurrent accesses to shared data by replacing locking with atomic execution units (called transactions). 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 an STM system is consequently to discharge the programmer from the direct management of synchronization entailed by concurrent accesses to shared 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 [10]). 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 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, an STM system has to do "its best" to execute as many transactions per time unit as possible. Similarly to a scheduler, an 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 a new transaction).

Content of the paper and roadmap. This paper is made up of 7 sections and has three contributions. Section 2 presents the computation model and the first contribution, namely, a new consistency condition, called *virtual world* consistency. In contrast to serializability [23] but similarly to opacity [11], 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 require 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. Virtual world consistency is formally defined in Section 3.

The second contribution, namely, an STM protocol that satisfies virtual world consistency, is presented in Section 4. 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. The proof of the protocol is presented in Section 5.

Then, Section 6 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 [1]. It also shows that, with a very simple modification, the protocol may ensure virtual world consistency under the use of *regular* instead of the stronger *atomic* shared objects. Finally, Section 7 concludes the paper.

2. Computation model and definitions

2.1. 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 $X \in \{1, \ldots, m\}$. Each process consists of a sequence of transactions (that are not known in advance).

Each of the m base objects is an atomic read/write object [22]. 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 associated with X).

2.2. 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 does not have to worry about the management of the base objects accessed by the transaction. In contrast from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or write any base object.

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.

Object operations. We denote operations on shared objects in the following way. A read operation by transaction T on object X is denoted $X.\mathsf{read}_T()$. Similarly, a write operation by transaction T on object X is denoted $X.\mathsf{write}_T()$.

Incremental snapshot. As in [6], we assume that the behavior of a transaction T can be decomposed in three sequential steps.[1] A transaction 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 reads a new object whose current value makes its incremental snapshot inconsistent, 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.

An aborted transaction is reduced to a read prefix. In the following, when 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]

2.3. 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 may have changed in between. As already said, one usually says that the transaction T computes an *incremental snapshot*.

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, 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.)

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

The classical consistency criterion for database transactions is serializability [23] (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 contrast to database transactions that are usually produced by SQL queries, in an STM system the code encapsulated in a transaction is not restricted to particular patterns. Consequently a transaction always has to operate on a consistent state.

---

1 This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model.

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. In contrast, in [2,4,16], 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 [9] 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.

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 consistent states (intuitively, a consistent state is a global state that, considering only the committed transactions, could have existed at some real time instant). 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 throws 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 see always a consistent state of the data it accesses. The aborted transactions have to be harmless. This observation has first been stated in [9].

2.5. From opacity to virtual world consistency

Opacity. Informally suggested in [9], and formally introduced and investigated in [11], the *opacity* consistency condition requires that no transaction reads values from an inconsistent global state where, considering only the committed transactions, 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, as defined below.

Let us associate with each aborted transaction T its execution prefix (called *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 (i) all committed transactions plus each aborted transaction reduced to a read prefix appear as if they have been executed sequentially and (ii) this sequence respects the transaction real-time occurrence order. Such a sequence id called *witness sequential execution*.

Examples of protocols implementing the opacity property, each with different additional features, can be found in [9,16,17,25].

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 [23] (so they all have the same "witness sequential execution") or linearizable [15] (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.[5] 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 may be mutually inconsistent as aborted transactions have not necessarily the same causal past (hence the name *virtual world* consistency). This consistency condition can benefit many STM applications as, from its local point of view, a transaction cannot differentiate it from opacity.

The formal definition of virtual world consistency is based on a total order on the committed transactions and a partial order on the whole set of transactions (where each aborted transactions is reduced to a read prefix). This definition is presented in the next section. To give the intuition of the condition, it is explained here informally with a simple example. Let us consider the transaction execution depicted in Fig. 1. There are two processes: p_1 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 gray square have aborted. From a dependency point of view, each transaction issued by a process depends on its previous committed transactions (process order relation[6]), 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 from y a value written by T_2^1). In contrast, 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 (left of the corresponding dotted lines). The values read by T_1' (resp., T_2') are consistent with respect to its causal past dependencies.

This consistency condition actually extends to STM systems the notions of *consistent cut*, *causal past*, and *consistent global state* encountered in asynchronous message-passing systems [5,7,8,28]. 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 sequential observer (i.e., they cannot be serialized). The "read-from" relation linking transactions is the STM equivalent

[5] The notion of *causal past* of a transaction is analogous to the notion of causal past encountered in message-passing [5,28]. See [18] for a formal definition and a parallel between transaction systems and message-passing systems.

[6] A process issues a new transaction only when its previous transaction has completed (by committing or aborting). This defines the *process order* relation [18].

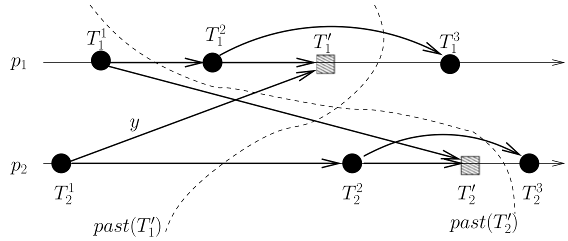

Fig. 1. Examples of causal pasts.

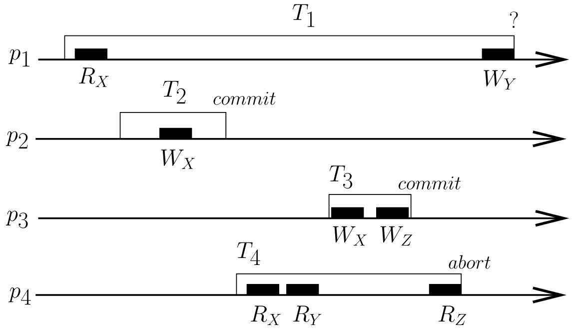

Fig. 2. A virtual world consistent history that is not opaque if T_1 is committed.

of the "message" relation that defines the flow of information exchange in message-passing systems. The "process order relation" is the same as 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 2.4) from occurring without requiring all the transactions (committed or aborted) to agree on the very same witness execution. Let us assume that, when executed alone reading 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.

2.6. Virtual world consistency vs. opacity

We will show in Section 3.3 that all opaque histories also satisfy virtual world consistency. We show here that a history can be virtual world consistent without being opaque. Informally, a transaction history is a partial order on a set of committed transactions and read prefixes of aborted transactions (see Section 3.1).

Let us consider Fig. 2. It presents an example of a transactional history that always satisfies virtual world consistency, but does not satisfy opacity if transaction T_1 is committed. In this figure, a read operation on object X is denoted R_X. Similarly, a write operation on object Y is denoted W_Y. Each process p_i executes a single transaction denoted $T_i \cdot T_2$ and T_3 are committed, T_4 is aborted, while the fate of T_1 is not decided yet.

Let us first consider the transactional history where T_4 does not exist and T_1 commits. This history is opaque: it (has only committed transactions and) accepts the linearization T_1 T_2 T_3.

Let us now consider the transactional history with T_2, T_3 and T_4, and where T_1 commits. Because T_1 commits, despite the fact that T_4 aborts, the presence of T_4 makes this history not opaque. Let us first notice that T_4 has to be aborted because its read of Z would cause it to observe an inconsistent state of the shared memory (it would read the value of Z written by T_3 but the value of X written by T_2, making T_3 no longer atomic). The history is not opaque because T_4 observes T_2 before T_1 (it reads T_2's value of X but not T_1's value of Y). However, before aborting, the read prefix of T_4 (made up of its reads R_X followed by R_Y) reads a consistent state of the shared memory. Hence, although the history is not opaque, it is virtual world consistent. In contrast, if T_1 was aborted (so it does not execute W_Y, and we have then two committed transactions and two aborted transactions), the history including these four transactions would be opaque.

So, when considering opacity, an aborted transaction (here T_4) can be the only cause entailing the abort of another transaction (here T_1). In that sense, opacity is a consistency condition more conservative than necessary.

3. Virtual world consistency

This section presents the formal definition of virtual world consistency. First, we define some properties of STM executions. Then, based on these definitions, we define virtual world consistency.

3.1. Base definitions

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 emphasize 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 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 emphasize 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*.

Transaction history. The execution of a set of transactions is represented by a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$, called *transaction history*, that states 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 at least one of the following is satisfied:
  1. (Process order.) $T1$ and $T2$ have been issued by the same process, $T1$ is a committed transaction and $E_{T1} <_H B_{T2}$.
  2. (Read_from order.) $\exists\ w_{T1}(X)v \ \wedge\ \exists\ r_{T2}(X)v$. This is denoted $T1 \xrightarrow{X}_{rf} T2$. (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". In contrast, whatever the values read by an aborted transaction (a priori those can be mutually consistent or not), those values do not "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 an STM the transactions are issued by processes. (In a database, there is no notion of process that relates transactions.) Of course, in an 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.

Committed transaction history. A *committed transaction history* (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 Fig. 3, 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.

Complete transaction history. A *complete transaction history* (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 (i) the fact that an aborted transaction T does not write into the shared memory, and (ii) the definition of the process order relation, that there is no outgoing edge from an aborted transaction T.

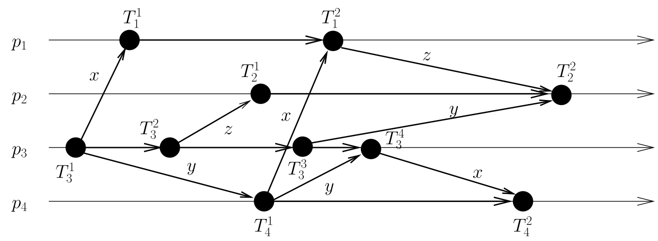

Fig. 3. A partial order $\widehat{CH} = (CH, \rightarrow_{CH})$ (only committed transactions).

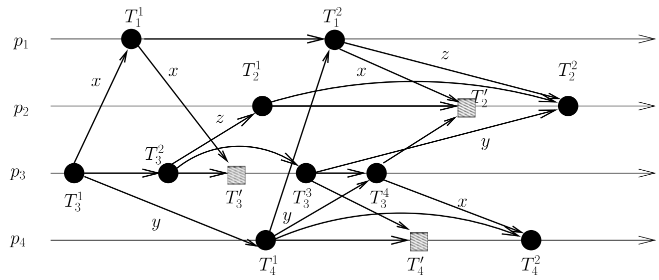

Fig. 4. A partial order $\widehat{CAH} = (CAH, \rightarrow_{CAH})$ (committed and aborted transactions).

Fig. 4 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).

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. Let us notice that this partial order relation $\rightarrow_{RT}$ on transactions is defined from the total order relation on events denoted $<_H$. Hence it differs from Lamport's "happened before" relation [21] that defines a partial order on all events.

Considering that the space/time diagrams depicted in the previous Figs. 3 and 4 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.,

- $S = PO$ (same elements),
- $\rightarrow_S$ is a total order, and
- $(T1 \rightarrow_{PO} T2) \Rightarrow (T1 \rightarrow_S T2)$ (we say "$\rightarrow_S$ 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 Fig. 3. (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'' such that $T' \rightarrow_S T'' \rightarrow_S T$ 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 $past(T)$.

3.2. Formal definition of virtual world consistency

Definition. Opacity requires that all transactions (be them committed or aborted) see the very same witness execution. 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 legal linear extension $\widehat{S_T}$ of the partial order $past(T)$.

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.

Rationale. The idea that underlies this consistency condition 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 mean 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 here is 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.

3.3. Virtual world consistency is strictly weaker than opacity

In Section 2.6 we have presented a transactional history that is virtual world consistent but not opaque. In this section, we show that any opaque history is also virtual world consistent, thus showing that virtual world consistency is strictly weaker than opacity.

Theorem 1. *Let us consider virtual world consistency where strict serializability is considered for the committed transactions. Any transactional history that is opaque is also virtual world consistent.*

Proof In an opaque history, an aborted transaction is reduced to a maximal prefix that contains only read operations that obtain mutually consistent values. In such a history, the committed transactions and the maximal read prefixes of the aborted transactions are linearizable. Thus, as the aborted transactions do not modify the objects, the subset of all the committed transactions is also linearizable.

Let us now consider the case of aborted transactions. In order to prove that an aborted transaction is virtual world consistent, we have to prove that its causal past is linearizable. Let us consider an aborted transaction T in an opaque history. If T's causal past was not linearizable, the set of all transactions would not be linearizable either. This is because, due to its very definition, the causal past of T includes all transactions that are causally related to T. So, any transactional history that is opaque is also virtual world consistent. $\Box_{Theorem\ 1}$

Theorem 1, associated with the example in Fig. 2, shows that virtual world consistency is strictly weaker than opacity while keeping its noteworthy feature (no transaction reads from an inconsistent state).

4. An STM protocol when the base objects are atomic

This section presents a protocol, called VWC-Prot, that implements the virtual world consistency condition. Among several noteworthy properties, this protocol, based on vector clocks, ensures the invisibility of the read operations issued by the transactions and presents noteworthy versatility features.

4.1. VWC-Prot: interface

VWC-Prot 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. This 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. This 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 $\mathsf{try\_to\_commit}_T()$. This 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 $X.\mathsf{read}_T()$.)

4.2. VWC-Prot: 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 and the update of both an object X and its vector $X.depend$ is done atomically.

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 $lc(X)$.

4.3. VWC-Prot: the algorithm and its properties

This section presents the VWC-Prot protocol and some of its properties. The code of VWC-Prot for a process p_i is described in Fig. 5. It consists of the algorithms that implement the four operations of the STM interface (Section 4.1), namely, $\text{begin}_T()$, $X.\text{read}_T()$, $X.\text{write}_T()$, and $\text{try\_to\_commit}_T()$, where T is a transaction issued by a process p_i and X is a base object. When the control value *abort* is returned, it carries a tag (1 or 2) that indicates the cause of the abort to the corresponding transaction.

The operation $\text{begin}_T()$. 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.

operation $\text{begin}_T()$: $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$; $t\_depend_T \leftarrow p\_depend_i$.
==
operation $X.\text{read}_T()$:
(01) **if** (there is no local copy of X) **then**
(02) allocate local space – denoted $lc_i(X)$ – for a local copy of X; $lc_i(X) \leftarrow X$;
(03) $lrs_T \leftarrow lrs_T \cup \{X\}$; $t\_depend_T[X] \leftarrow lc_i(X).depend[X]$;
(04) **if** $(\exists Y \in lrs_T : t\_depend_T[Y] < lc_i(X).depend[Y])$ **then** return(*abort*, 1) **end if**;
(05) **for each** $Y \notin lrs_T$ **do** $t\_depend_T[Y] \leftarrow \max(t\_depend_T[Y], lc_i(X).depend[Y])$ **end for**
(06) **end if**;
(07) return ($lc_i(X).value$).
==
operation $X.\text{write}_T(v)$:
(08) **if** (there is no local copy of X) **then** allocate local space $lc_i(X)$ to store v **end if**;
(09) $lc_i(X).value \leftarrow v$; $lws_T \leftarrow lws_T \cup \{X\}$; return (*ok*).
==
operation $\text{try\_to\_commit}_T()$:
(10) **let** $ConsistencyCheck_T$ **be the predicate** $(\forall Z \in lrs_T : t\_depend_T[Z] = Z.depend[Z])$;
(11) lock all the objects in $lrs_T \cup lws_T$;
(12) **if** $(lrs_T \neq \emptyset)$ **then**
if $(\neg ConsistencyCheck_T)$ **then** release all the locks; return(*abort*, 2) **end if end if**;
(13) **if** $(lws_T \neq \emptyset)$ **then for each** $X \in lws_T$ **do** $t\_depend_T[X] \leftarrow X.depend[X] + 1$ **end for**;
(14) **for each** $X \in lws_T$ **do** $X \leftarrow (lc_i(X).value, t\_depend_T)$ **end for**
(15) **end if**;
(16) release all the locks;
(17) $p\_depend_i \leftarrow t\_depend_T$;
(18) return(*commit*).

Fig. 5. VWC-Prot (code for p_i).

The operation $X.\text{read}_T()$. 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 $X.\text{read}_T()$ is its first read of X, p_i first builds a copy $lc_i(X)$ 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 $lc_i(X).value$ it has just obtained from the shared memory and the values it has previously read can belong to a consistent global state. If it is not the 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 $\exists Y \in lrs_T : t\_depend_T[Y] < lc_i(X).depend[Y]$) (line 04). If this predicate is true, p_i aborts T. Otherwise, p_i 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 on 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 belong to a consistent global state (also called "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 deadlock and starvation.[7] 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 know 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 and their dependencies 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. To check if both reads and the writes of T can appear as being executed atomically, the predicate $ConsistencyCheck_T$ is evaluated, and this evaluation is done after 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. $T1$ and $T2$ are independent if $(lrs_{T1} \cup lws_{T1}) \cap (lrs_{T2} \cup lws_{T2}) = \emptyset$.

Property 7. *Concurrent transactions that are independent can commit independently.*

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 $lc_i(X)$ – for a local copy of X; $lc_i(X) \leftarrow X$
> **end if**;
> $\mathsf{return}(lc_i(X).value)$.

[7] This can be easily obtained by defining a total order on the objects shared by the transactions.

Proof. Properties have been stated. Their aim is to give a better intuition of what the algorithms described in Fig. 5 do and how they do it. The proof that they satisfy the virtual consistency condition is presented in Section 5, namely, all committed transactions can be linearized, and the appropriate read prefixes of each aborted transaction are consistent with respect to their causal past. □

4.4. *Cost of the protocol*

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 [9]) aborting a transaction when it accesses an object that is locked.

5. Correctness proof of VWC-Prot

The proof is structured in three parts. Section 5.1 shows that the committed transaction history $\widehat{CH} = (CH, \rightarrow_{CH})$ admits a legal linear extension. Then, Section 5.2 shows that the appropriate prefixes of every aborted transaction reads form a global state that is consistent with respect to its causal past. Finally, Section 5.3 pieces together the previous proofs to show that VWC-Prot ensures the virtual world consistency condition.

Notation. "$w(X) \in T$" is used as a shortcut for "T is a committed transaction and it has issued the operation $X.\mathsf{write}_T()$".

5.1. *Committed transactions are linearizable*

To show that the committed transaction history $\widehat{CH} = (CH, \rightarrow_{CH})$ admits a legal linear extension, let us consider an extension $\widehat{S} = (S, \rightarrow_S)$ 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. $\forall T1, T2 \in S, \forall X : T1 \xrightarrow{X}_{rf} T2 \Rightarrow (\nexists T3 : T1 \rightarrow_S T3 \rightarrow_S T2 \wedge w(X) \in T3)$ (no transaction reads an overwritten value),
3. $\forall T1, T2 \in S, \forall X : T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \rightarrow_S T2$ (no transaction reads from the future), and
4. $\forall T1, T2 \in S : T1 \rightarrow_{RT} T2 \Rightarrow T1 \rightarrow_S T2$ (real-time order is respected).

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

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 $T2$ 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 If $T1 \xrightarrow{X}_{rf} T2$ there is an event $w_{T1}(X)$ and an event $r_{T2}(X)$ such that $w(X)T1 <_H r(X)T2$. Because the commit of $T2$ can only be its last operation, we then have $w_{T1}(X) <_H r_{T2}(X) <_H AL_{T2}(X)$ and so $w_{T1}(X) <_H RL_{T1}(X) <_H AL_{T2}(X)$. 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). $\square_{Lemma\ 4}$

5.2. 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. $\square_{Lemma\ 5}$

Definition 3. Let $\mathcal{C}$ denote the set of committed transactions.

Lemma 6. *Given a transaction T, past(T)\{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 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 at line 05 if it does not, and line 13 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). Otherwise ($X \neq Y$), 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. $\square_{Lemma\ 7}$

Definition 4. Let $\mathcal{A}$ denote the set of aborted transactions.

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. $\forall T1 \in past(T)\backslash\{T\}, \forall X : T1 \xrightarrow{X}_{rf} T \Rightarrow (\nexists T3 : T1 \rightarrow_T T3 \rightarrow_T T \wedge w(X) \in T3)$.
   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 $T1 \xrightarrow{X}_{rf} T$, we have $r_T(X) <_H w_{T3}(X)$ 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. $\Box_{Lemma\ 8}$

5.3. VWC-Prot is correct

Theorem 2. *VWC-Prot satisfies strong virtual world consistency.*

Proof The follows from Lemmas 1–4 that prove that the protocol satisfies linearizability for committed transactions and from Lemma 8 that proves that it satisfies virtual world consistency for aborted transactions. $\Box_{Theorem\ 2}$

6. Versatility dimension of VWC-Prot

This section shows that VWC-Prot is particularly versatile. With very simple modifications, we obtain a protocol that satisfies causal consistency [1,24] (Section 6.1), a protocol that works with regular objects [22] instead of atomic objects (Section 6.2), and a protocol that works when objects are neither atomic nor regular (Section 6.3).

6.1. From virtual world consistency to causal consistency

Causally consistent transactions. The concept of *causal consistency* for read/write objects has been introduced in [1] under the name *causal memory*. It has then been extended to transactions in [24] 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" and "process order" relations.

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" and "process order" relations as exhibited in $\mathcal{C}$.

Adapting the protocol. The base protocol described in Fig. 5 can be adapted very easily (weakened) to implement causal consistency. The single modification consists in adding the statement "**if** $lws_T = \emptyset$ **then** return($commit$) **end if**" just before line 11. This means that, when $lws_T \neq \emptyset$, the cost is the same as for virtual world consistency.

This modification does not alter VWC-Prot 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 causal consistency condition).

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 has only to be consistent with respect to the causal past of the read-only transaction.

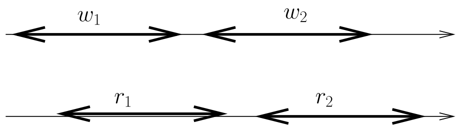

Fig. 6. Operations on a shared object.

6.2. From atomic objects to regular objects

Regular read/write object. A single writer *regular* read/write object [22] has one writer and any number of readers. Regular objects with multiple writers and multiple readers have been investigated in [27] where three different regularity definitions are presented. Here we consider that the writes appear as being 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*. Fig. 6 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 (i) its read and write operations appear as if they have been executed sequentially, and (ii) this total order respects 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. This line becomes (the new statement is the **if** statement):
  $lrs_T \leftarrow lrs_T \cup \{X\}$;
  if $(t\_depend_T[X] > lc_i(X).depend_T[X])$ **then** $\mathsf{return}(abort, 3)$ **end if**;
  $t\_depend_T[X] \leftarrow lc_i(X).depend[X]$.
- $ConsistencyCheck_T$ becomes $(\forall\, Z \in lrs_T : t\_depend_T[Z] \geq Z.depend[Z])$.

The meaning of the result $(abort, 3)$ returned in the **if … end if** statement is the following. First, the transaction T has previously read an object (say Y) the value of which depends on the value of X whose sequence number is $sn = t\_depend_T[X]$. The sequence number sn' of X just read by T ($sn' = lc_i(X).depend_T[X]$)) is such that $sn' < sn$. This witnesses a new/old inversion involving the "early" read of X – issued by = some T' – that obtained the new value of X to produce the value of Y, and the "late" read of X by T that obtained a previous value of X. While this behavior is impossible when the base objects are atomic, it can happen in concurrency patterns when the base objects $X, Y, \ldots$ are only regular.

Property 8. *If the invocation of* $X.\mathsf{read}_T()$ *by* T *returns* $(abort, 3)$*, the abort is due to a new/old inversion.*

6.3. 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 Fig. 5 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); $lc_i(X) \leftarrow X$; unlock(X)".

Depending on implementation choices, concurrent transactions that try to access X when it is locked could either abort or wait. A read operation does not modify the value and holds the lock only for a short time, so if X is locked because of a read operation, it could be beneficial to let the transaction wait instead of aborting it.

7. 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 an STM protocol, called VWC-Prot, 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 interesting versatility features. 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 clock" approach [29]. 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 of size k (which is bounded and much smaller than m, the number of shared objects) has a price: due to the false additional dependencies, more transactions may 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]. Moreover, as the TL2 protocol [9] is based on a scalar clock, it would be interesting to investigate if the proposed protocol and TL2 could be derived from a more general framework, with scalar clock being the appropriate mechanism for opacity, and vector clock the appropriate mechanism for virtual world consistency. Finally, evaluating the proposed STM system on a realistic benchmark constitutes an interesting direction of a more applied fundamental research.

Acknowledgments

An early draft of this paper presenting mainly the STM protocol without its proof has appeared in the proceedings of the SIROCCO'09 colloquium [19]. We thank José Ramón Mendívil for discussions on the STM consistency conditions. Last but not least, we also thank the referees for their constructive comments which helped us to improve the presentation of the paper.

References

[1] M. Ahamad, G. Neiger, J.E. Burns, P. Kohli, Ph.W. Hutto, Causal memory: definitions, implementation, and programming, Distributed Computing 9 (1) (1995) 37–49.
[2] Y. Afek, H. Attiya, D. Dolev, E. Gafni, M. Merritt, N. Shavit, Atomic snapshots of shared memory, Journal of the ACM 40 (4) (1993) 873–890.
[3] H. Attiya, Needed: foundations for transactional memory, ACM Sigact News, DC Column 39 (1) (2008) 59–61.
[4] H. Attiya, R. Guerraoui, E. Ruppert, Partial snapshot objects, in: Proc. 20th ACM Symposium on Parallel Algorithms and Architectures, SPAA'08, ACP Press, ACM Press, 2008, pp. 336–343.
[5] Ö. Babaoğlu, K. Marzullo, Distributed systems, in: Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms, in: Frontier Series, ACM Press, 1993, pp. 55–93. Chapter 4.
[6] Ph.A. Bernstein, D.W. Shipman, W.S. Wong, Formal aspects of serializability in database concurrency control, IEEE Transactions on Software Engineering SE-5 (3) (1979) 203–216.
[7] K.M. Chandy, L. Lamport, Distributed snapshots: determining global states of distributed systems, ACM Transactions on Operating Systems 3 (1) (1985) 63–75.
[8] R. Cooper, K. Marzullo, Consistent detection of global predicates, in: Proc. ACM/ONR Workshop on Parallel and Distributed Debugging, ACM Press, 1991, pp. 167–174.
[9] D. Dice, O. Shalev, N. Shavit, Transactional locking II, in: Proc. 20th Int'l Symposium on Distributed Computing, DISC'06, in: LNCS, vol. 4167, Springer-Verlag, 2006, pp. 194–208.
[10] P. Felber, Ch. Fetzer, R. Guerraoui, T. Harris, Transactions are coming back, but are they the same? ACM Sigact News, DC Column 39 (1) (2008) 48–58.
[11] R. Guerraoui, M. Kapałka, On the correctness of transactional memory, in: Proc. 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP'08, ACM Press, 2008, pp. 175–184.
[12] T. Harris, A. Cristal, O.S. Unsal, E. Ayguade, F. Gagliardi, B. Smith, M. Valero, Transactional memory: an overview, IEEE Micro 27 (3) (2007) 8–29.
[13] M.P. Herlihy, V. Luchangco, Distributed computing and the multicore revolution, ACM SIGACT News, DC Column 39 (1) (2008) 62–72.
[14] M.P. Herlihy, J.E.B. Moss, Transactional memory: architectural support for lock-free data structures, in: Proc. 20th ACM Int'l Symposium on Computer Architecture, ISCA'93, 1993, pp. 289–300.
[15] M.P. Herlihy, J.M. Wing, Linearizability: a correctness condition for concurrent objects, ACM Transactions on Programming Languages and Systems 12 (3) (1990) 463–492.
[16] D. Imbs, M. Raynal, A lock-based STM protocol that satisfies opacity and progressiveness, in: 12th Int'l Conf. On Principles Of Distributed Systems, OPODIS'08, in: LNCS, vol. 5401, Springer-Verlag, 2008, pp. 226–245.
[17] D. Imbs, M. Raynal, Provable STM properties: leveraging clock and locks to favor commit and early abort, in: Proc. 10th Int'l Conference on Distributed Computing and Networking, ICDCN'09, in: LNCS, vol. 5408, Springer-Verlag, 2009, pp. 67–78.
[18] D. Imbs, J.R. Mendívil, M. Raynal, Virutal world consistency: a new condition for STM systems, in: BA. Proc. 20th ACM Symposium on Distributed Computing, PODC'09, ACM Press, 2009, pp. 280–281. Full version: On the Consistency Conditions of Transactional Memories. Tech Report #1917, p. 23, IRISA, Université de Rennes (France), 2009.
[19] D. Imbs, M. Raynal, A versatile STM protocol with invisible read operations that satisfies the virtual world consistency condition, in: Proc. of the 16th Colloquium on Structural Information and Communication Complexity, SIROCCO'09, in: LNCS, vol. 5869, Springer-Verlag, 2009, pp. 266–280.
[20] D. Imbs, M. Raynal, Software transactional memories: an approach for multicore programming. Journal of Supercomputing, Published Online First February 2010, http://dx.doi.org/10.1007/s11227-010-0388-0.
[21] L. Lamport, Time, clocks and the ordering of events in a distributed system, Communications of the ACM 21 (7) (1978) 558–565.
[22] L. Lamport, On interprocess communication. Part 1: Models, Part 2: Algorithms, Distributed Computing 1 (2) (1986) 77–101.
[23] Ch.H. Papadimitriou, The serializability of concurrent updates, Journal of the ACM 26 (4) (1979) 631–653.
[24] M. Raynal, G. Thia-kime, M. Ahamad, From serializable to causal transactions, in: BA. Proc. 20th ACM Symposium on Distributed Computing, PODC'96, ACM Press, 1996, p. 310. Full version: From serializable to causal transactions for collaborative applications. Proc. 23th EUROMICRO Conference, IEEE Computer Press, 1997, pp. 314–321.
[25] T. Riegel, C. Fetzer, P. Felber, Time-based transactional memory with scalable time bases, in: Proc. 19th annual ACM Symposium on Parallel Algorithms and Architectures, SPAA'07, ACM Press, 2007, pp. 221–228.
[26] N. Shavit, D. Touitou, Software transactional memory, Distributed Computing 10 (2) (1997) 99–116.
[27] C. Shao, E. Pierce, J. Welch, Multi-writer consistency conditions for shared memory objects, in: Proc. 17th Int'l Symposium on Distributed Computing, DISC'03, in: LNCS, vol. 2848, Springer-Verlag, 2003, pp. 106–120.
[28] R. Schwarz, F. Mattern, Detecting causal relationship in distributed computations: in search of the holy grail, Distributed Computing 7 (1993) 149–174.
[29] F. Torres-Rojas, M. Ahamad, Plausible clocks: constant size logical clocks for distributed systems, Distributed Computing 12 (1999) 179–195.