Read invisibility, virtual world consistency and permissiveness are compatible

Tyler Crain, Damien Imbs, Michel Raynal

Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1958 – novembre 2010

Read invisibility, virtual world consistency and permissiveness are compatible

Tyler Crain[*] , Damien Imbs[**] , Michel Raynal[***]
tyler.crain@irisa.fr, damien.imbs@irisa.fr, raynal@irisa.fr

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 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, a STM system has to allow transactions to execute concurrently. Consequently, due to the underlying STM concurrency management, a transaction commits or aborts.

This paper studies the relation between two STM properties (read invisibility and permissiveness) and two consistency conditions for STM systems, namely, opacity and virtual world consistency. Both conditions ensures that any transaction (be it a committed or an aborted transaction) reads values from a consistent global state, a noteworthy property if one wants to prevents abnormal behavior from concurrent transactions that behave correctly when executed alone. A read operation issued by a transaction is invisible if it does not entail shared memory modifications. This is an important property that favors efficiency and privacy. An STM system is permissive with respect to a consistency condition if it accepts every history that satisfies the condition. This is a crucial property as a permissive STM system never aborts a transaction "for free". The paper first shows that read invisibility, permissiveness and opacity are incompatible, which means that there is no permissive STM system that implements opacity while ensuring read invisibility. It then shows that invisibility, permissiveness and opacity are compatible. To that end the paper describes a new STM protocol called IR_VWC_P. This protocol presents additional noteworthy features: it uses only base read/write objects and locks which are used only at commit time; it satisfies the disjoint access parallelism property; and, in favorable circumstances, the cost of a read operation is $O(1)$.

Key-words: Asynchronous shared memory system, Commit/abort, Concurrent object, Consistency condition, Opacity, Permissiveness, Serializability, Software transactional memory, Transaction, Virtual world consistency.

Les lectures invisibles, la cohérence des mondes virtuels et la permissivité sont compatibles

Résumé : *Ce rapport montre que les lectures invisibles, la cohérence des mondes virtuels et la permissivité sont compatibles. Il présente un algorithme qui réunit ces conditions.*

Mots clés : *Mémoire partagée, Commit/abort, Objet concurrent, Condition de cohérence, Opacité, Permissivité, Serialisabilité, Mémoire transactionnelle, Transaction, Cohérence des mondes virtuels.*

---

[*] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
[**] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes
[***] Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

1 Introduction

1.1 Software transactional memory (STM) systems

The aim of an STM system is to simplify the design and the writing of concurrent programs by discharging the programmer from the explicit management of synchronization entailed by concurrent accesses to shared objects. This means that, when faced to synchronization, a programmer has to concentrate on where atomicity is required and not on the way it is realized.

More explicitly, an STM is a middleware approach that provides the programmers with the *transaction* concept [11, 21]. This concept is close but different from the notion of transactions encountered in databases [6, 9, 10]. 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, a STM system has to do "its best" to execute and commit 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 a new transaction). Abort is the price that has to be paid by transactional systems to cope with concurrency in absence of explicit synchronization mechanisms (such as locks or event queues).

1.2 Two consistency conditions for STM systems

The opacity consistency condition The classical consistency criterion for database transactions is serializability [18] (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 a 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. 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.

Informally suggested in [5], and formally introduced and investigated in [8], 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. 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 its read prefix appear as if they have been executed sequentially and (ii) this sequence respects the transaction real-time occurrence order.

Virtual world consistency This consistency condition, introduced in [15], 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 [18] (so they all have the same "witness sequential execution") or linearizable [12] (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 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.

In addition to the fact that it can allow more transactions to commit than opacity, one of the main advantages of virtual world consistency lies in the fact that, as opacity, it prevents bad phenomena (as described previously) from occurring without requiring all the transactions (committed or aborted) to agree on the very same witness execution. Let us assume that each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.) when, executed alone, it reads values from a consistent global state. 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.

1.3 Desirable properties for STM systems

Invisible read operation A read operation issued by a transaction is *invisible* if it does not entail the modification of base shared objects used to implement the STM system [17]. This is a desirable property for both efficiency and privacy.

Base operations and underlying locks The use of expensive base synchronization operations such as Compare&Swap() or the use of underlying locks to implement an STM system can make it inefficient and prevent its scalability. Hence, an STM systems should use synchronization operations sparingly (or even not at all) and the use of locks should be as restricted as possible.

Disjoint access parallelism Ideally, an STM system should allow transactions that are on distinct objects to execute without interference, i.e., without accessing the same base shared variables. This is important for efficiency and restricts the number or unnecessary aborts.

Permissiveness The notion of permissiveness has been introduced in [7] (in some sense, it is a very nice generalization of the notion of *obligation* property [14]). It is on transaction abort. Intuitively, an STM system is *permissive* "if it never aborts a transaction unless necessary for correctness" (otherwise it is *non-permissive*). More precisely, an STM system is permissive with respect to a consistency condition (e.g., opacity) if it accepts every history that satisfies the condition.

Some STM systems are randomized in the sense that the commit/abort point of a transaction depends on a random coin toss. Probabilistic permissiveness is suited to such systems. A randomized STM system is *probabilistically permissive* with respect to a consistency condition if every history that satisfies the condition is accepted with positive probability [7].

As indicated in [7], an STM system that checks at commit time that the values of the objects read by a transaction have not been modified (and aborts the transaction if true) cannot be permissive with respect to opacity.

1.4 Content of the paper

This paper is on permissive STM systems with invisible reads. It has several contributions.

- It first shows that an STM system that satisfies read invisibility and opacity cannot be permissive.
- The paper then presents an STM system (called IR_VWC_P) that satisfies read invisibility, virtual world consistency and permissiveness. The STM IR_VWC_P protocol presents additional noteworthy properties.
  - It uses only base read/write operations and locks, each associated with a shared object. Moreover, a lock is used at most once by a transaction at the end of a transaction (when it executes an operation called try_to_commit()).
  - it satisfies the disjoint access parallelism property.
  - It allows for *fast* read operations. More precisely, the first read of an object X by a transaction T can cost up to $O(|lrs_T|)$ shared memory accesses (where lrs_T is the current read set of T) in the worst case and $O(1)$ shared memory accesses in favorable cases. The cost of all the following reads of X by T is $O(1)$.

The paper is made up of 9 sections. Section 2 presents the computation model. Section 3 presents the opacity and virtual world consistency conditions. Section 4 shows that opacity, read invisibility and permissiveness are incompatible. The IR_VWC_P protocol is then described incrementally. Section 5 presents first a base version that satisfies the VWC property and invisibility of read operations. This protocol is proved in Section 6 and improved in Section 7 (these improvements concern additional features such as garbage collection of useless cells and fast read operations). Finally, Section 8 enriches it to obtain a permissive protocol. Section 9 concludes the paper.

2 STM computation model and base definitions

2.1 Processes and atomic shared objects

An application is made up of an arbitrary number of processes and m 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 consists of a sequence of transactions (that are not known in advance).

Each of the m shared objects is an atomic read/write object. 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 $op1(X)$ terminates before $op2(X)$ starts, $op1$ appears before $op2$ in the witness sequence associated with X).

2.2 Transactions and object operations

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. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or write any shared 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 shared objects is a *read-only* transaction, otherwise it is an *update* transaction. A transaction that issues only write operations is a *write-only* transaction.

Transaction are assumed to b dynamically defined. The important point is here that the underlying STM system does not know in advance the transactions. It is an on-line system (as a scheduler).

Operations issued by a transaction We denote operations on shared objects in the following way. A read operation by transaction T on object X is denoted $X.\mathsf{read}_T()$. Such an operation returns either the value v read from X or the value $abort$. When a value v is returned, the notation $X.\mathsf{read}_T(v)$ is sometimes used. Similarly, a write operation by transaction T of value v into object X is denoted $X.\mathsf{write}_T(v)$ (when not relevant, v is omitted). Such an operation returns either the value ok or the value $abort$. The notations $\exists\ X.\mathsf{read}_T(v)$ and $\exists\ X.\mathsf{write}_T(v)$ are used as predicates to state whether a transaction T has issued a corresponding read or write operation.

If it has not been aborted during a read or write operation, a transaction T invokes the operation $\mathsf{try\_to\_commit}_T()$ when it terminates. That operation returns it $commit$ or $abort$.

Incremental snapshot As in [3], we assume that the behavior of a transaction T can be decomposed in three sequential steps: 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 $\mathsf{read\_modify\_write}()$ operation that is dynamically defined by a process. (This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model.)

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*. 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 inconsistent its incremental snapshot, the transaction is directed to abort. If the transaction is not aborted during its read phase, T issues local computations. Finally, if the transaction is an update transaction, and its write operations can be issued in such a way that the transaction appears as being executed atomically, the objects of its write set are updated and the transaction commits. Otherwise, it is aborted.

Read prefix of an aborted transaction A read prefix is associated with every transaction that aborts. This read prefix contains all its read operations if the transaction has not been aborted during its read phase. If it has been aborted during its read phase, its read prefix contains all read operations it has issued before the read that entailed the abort. Let us observe that the values obtained by the read operations of the read prefix of an aborted transaction are mutually consistent (they are from a consistent global state).

3 Opacity and virtual world consistency

This section defines formally opacity [8] and virtual world consistency [15]. First, we define some properties of STM executions. Then, based on these definitions, opacity and virtual world consistency are defined.

3.1 Base definitions

Preliminary remark Some of the notions that follow can be seen as read/write counterparts of notions encountered in message-passing systems (e.g., partial order and happened before relation [16], consistent cut, causal past and observation [2, 22]).

Strong **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 including all committed transactions plus all aborted transactions (each reduced to its read prefix).
- $T1 \rightarrow_{PO} T2$ (we say "$T1$ precedes $T2$") if one of the following is satisfied:
  1. Strong process order. $T1$ and $T2$ have been issued by the same process, with $T1$ first.
  2. Read_from order. $\exists\ X.\mathsf{write}_{T1}(v)\ \wedge\ \exists\ X.\mathsf{read}_{T2}(v)$. This is denoted $T1 \stackrel{X}{\rightarrow}_{rf} T2$. (There is an object X whose value v written by $T1$ has been read by $T2$.)
  3. Transitivity. $\exists T : (T1 \rightarrow_{PO} T) \wedge\ (T \rightarrow_{PO} T2)$.

Weak **transaction history** The definition of a weak transaction history is the same as the one of a strong transaction history except for the "process order" relation that is weakened as follows:

- Weak process order. $T1$ and $T2$ have been issued by the same process with $T1$ first, and $T1$ is a committed transaction.

This defines a less constrained transaction history. In a weak transaction history, no transaction "causally depends" on an aborted transaction (it has no successor in the partial order).

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.

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, when $\widehat{PO}$ is a weak transaction history, an aborted transaction T is the only aborted transaction contained in its causal past $past(T)$. Differently, in a strong transaction history, an aborted transaction always causally precedes the next transaction issued by the same process. As we will see, this apparently small difference in the definition of strong and weak transaction partial orders has a strong influence on the properties of the corresponding STM systems.

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 $\rightarrow_{PO}$").

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 $X.\mathsf{read}_T(v)$ operation, there is a committed transaction T' such that:

- $T' \rightarrow_S T$ and $\exists\ X.\mathsf{write}_{T'}(v)$, and
- $\nexists\ T''$ such that $T' \rightarrow_S T'' \rightarrow_S T$ and $\exists\ X.\mathsf{write}_{T''}()$.

If all 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.

Real time order Let $\rightarrow_{RT}$ be the *real time* relation defined as follows: $T1 \rightarrow_{RT} T2$ if $T1$ has terminated before $T2$ starts. This relation (defined either on the whole set of transactions, or only on 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.

A linear extension $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ is real time-compliant if $\forall T, T' \in S$: $(T \rightarrow_{RT} T') \Rightarrow (T \rightarrow_S T')$.

3.2 Opacity and virtual world consistency

Both opacity and virtual world consistency ensures that no transaction reads from an inconsistent global state. If each transaction taken alone is correct, this prevents bad phenomena such as the ones described in the Introduction (e.g., entering an infinite loop). Their main difference lies in the fact that opacity considers strong transaction histories while virtual world consistency considers weak transaction histories.

Definition 1 *A strong transaction history satisfies the opacity consistency condition if it has a real time-compliant legal linear extension.*

Examples of protocols implementing the opacity property, each with different additional features, can be found in [5, 13, 15, 20].

Definition 2 *A weak transaction history satisfies the virtual world consistency condition if (a) all its committed transactions have a legal linear extension and (b) the causal past of each aborted transaction has a legal linear extension.*

A protocol implementing virtual world consistency can be found in [15] where it is also shown that any opaque history is virtual world consistent. In contrast, a virtual world consistent history is not necessarily opaque.

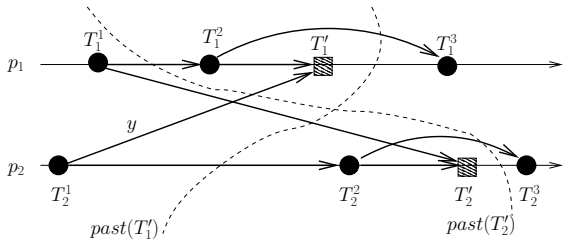

Figure 1: Examples of causal pasts

To give a better intuition of the virtual world consistency condition, let us consider the execution depicted on Figure 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 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 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.

4 Invisible reads, opacity and permissiveness are incompatible

Theorem 1 *Read invisibility, opacity and permissiveness (or probabilistic permissiveness) are incompatible.*

Proof Let us first consider permissiveness. The proof follows from a simple counter-example where three transactions T_1, T_2 and T_3 issue sequentially the following operations (depicted in Figure 2).

1. T_3 reads object X.
2. Then T_2 writes X and terminates. If the STM system is permissive it has to commit T_2. This is because if (a) the system would abort T_2 and (b) T_3 would be made up of only the read of X, aborting T_2 would make the system non-permissive. Let us notice that, at the time at which T_2 has to be committed or aborted, the future behavior of T_3 is not known and T_1 does not yet exist.
3. Then T_1 reads X and Y. Let us observe that the STM system has not to abort T_1. This is because when T_1 reads X there is no conflict with another transaction, and similarly when T_1 reads Y.

4. Finally, T_3 writes Y and terminates. let us observe that T_3 must commit in a permissive system where read operations (issued by other processes) are invisible. This is because, due to read invisibility, T_3 does not know that T_1 has previously issued a read of Y. Moreover, T_1 has not yet terminated and terminates much later than T_3. Hence, whatever the commit/abort fate of T_1, due to read invisibility, no information on the fact that T_1 has accessed Y has been passed from T_1 to T_3: when the fate of T_3 has to de decided, T_3 is not aware of the existence of T_1.

   (Let us remark that, a non-permissive system would abort T_3. This is because when, at which T_3 terminates, it tries to validate the read of X -in order to check if the read of X and the write of Y by T_3 can appear as being executed atomically-, it would discover thanks to the visible read of T_1 that its read of X has been overwritten .)

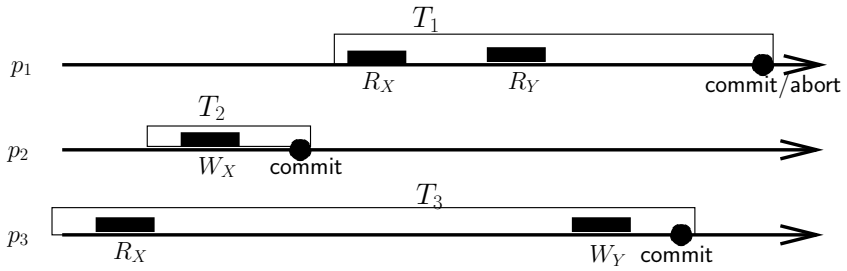

Figure 2: Invisible reads, opacity and permissiveness are incompatible

The strong transaction history $\widehat{PO} = (\{T_1, T_2, T_3\}, \rightarrow_{PO})$ associated with the previous execution is such that:

- $T_3 \rightarrow_{PO} T_2$ (follows from the fact that T_2 overwrites the value of X read by T_3).
- $T_2 \rightarrow_{PO} T_1$ (follows from the fact that T_1 reads the value of X written by T_2). Let us observe that this is independent from the fact that T_1 will be later aborted or committed. (If T_1 is aborted it is reduced to its read prefix "X.read(); Y.read()" that obtained values from a consistent global state.)
- Due to the sequential accesses on Y that is read by T_1 and then written by T_3, we have $T_1 \rightarrow_{PO} T_3$.

If follows from the previous item that $T_1 \rightarrow_{PO} T_1$. A contradiction from which we conclude that there is no protocol with invisible read operations that both is permissive and satisfies opacity.

Let us now consider probabilistic permissiveness. Actually, the same counter-example and the same reasoning as before applies. As none of T_2 and T_3 violates opacity, a probabilistic STM system that implements opacity with invisible read operations has a positive probability of committing both of them. As read operations are invisible, there is positive probability that both read operations on X and Y issued by T_1 be accepted by the STM system. It then follows that the strong transaction history $\widehat{PO} = (\{T_1, T_2, T_3\}, \rightarrow_{PO})$ associated with the execution in which T_2 and T_3 are committed while T_1 is aborted has a positive probability to be accepted. It is trivial to see that this execution is the same as in the non-probabilistic case for which it has been shown that this history is not opaque.

$\square_{Theorem\ 1}$

Remarks The previous proof shows that opacity is a too strong consistency condition when one wants both read invisibility and permissiveness. Differently, when considering the previous execution, the virtual world consistency protocol IR_VWC_P presented in this paper will abort transaction T_1. It is easy to see that the corresponding weak transaction history is virtual world consistent: The read prefix "$X.\mathsf{read}_{T_1}()$; $Y.\mathsf{read}_{T_1}()$" of the aborted transaction T_1 can be ordered after T_2 (and T_3 does not appear in its causal past).

5 Step 1: Ensuring virtual world consistency with read invisibility

As announced in the Introduction, the protocol IR_VWC_P is built is two steps. This section presents the first step, namely, a protocol that ensures virtual consistency with invisible read operations. The second step (Section 8) will enrich this base protocol to obtain permissiveness.

5.1 Base objects, STM interface, incremental reads and deferred updates

The underlying system on top of which is built the STM system is made up of base shared read/write variables (also called registers) and locks. Some of the base variables are used to contain pointer values. As we will see, not all the base registers are required to be atomic. There is an exclusive lock per shared object.

The STM system provides the process that issues a transaction T with four operations. The operations $X.\mathsf{read}_T()$, $X.\mathsf{write}_T()$, and $\mathsf{try\_to\_commit}_T()$ have been already presented. The operation $\mathsf{begin}_T()$ is invoked by a transaction T when it starts. It initializes local control variables.

The proposed STM system is based on the incremental reads and deferred update strategy. 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 explained, this is the *incremental snapshot* strategy.

When T invokes $X.\mathsf{write}_T(v)$, it writes v into its working space (and does not access the shared memory) and always returns ok. 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.)

5.2 The underlying data structures

Implementing a transaction-level shared object Each transaction-level shared object X is implemented by a list. Hence, at the implementation level, there is a shared array $PT[1..m]$ such that $PT[X]$ is a pointer to the list associated with X. This list is made up of cells. Let $CELL(X)$ be such a cell. It is made up of the following fields (see Figure 3).

- $CELL(X).value$ contains the value v written into X by some transaction T.
- $CELL(X).begin$ and $CELL(X).end$ are two dates (real numbers) such that the right-open time interval $[CELL(X).begin..CELL(X).end[$ defines the lifetime of the value kept in $CELL(X).value$. Operationally, $CELL(X).begin$ is the commit time of the transaction that wrote $CELL(X).value$ and $CELL(X).end$ is the date from which $CELL(X).value$ is no longer valid.
- $CELL(X).last\_read$ contains the commit date of the latest transaction that read object X and returned the value $v = CELL(X).value$.
- $CELL(X).next$ is a pointer that points to the cell containing the first value written into X after $v = CELL(X).value$. $CELL(X).prev$ is a pointer in the other direction.

  It is important to notice that none of these pointers are used in the protocol (Figure 4) that ensures virtual world consistency and read invisibility. $CELL(X).next$ is required only when one wants to recycle e inaccessible cells (see Section 7.1). Differently, $CELL(X).next$ will be used to obtain permissiveness (see Section 8).

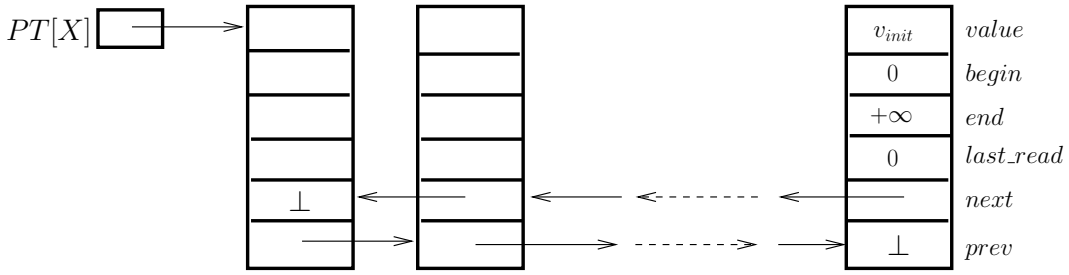

Figure 3: List implementing a transaction-level shared object X

No field of a cell is required to be an atomic read/write register of the underlying shared memory. Moreover, all fields (but $CELL(X).last\_read$) are simple write-once registers. Initially $PT[X]$ points to a list made up of a single cell containing the tuple $\langle v_{init}, 0, +\infty, 0, \perp, \perp\rangle$, where v_{init} is the initial value of X.

Locks A exclusive access lock is associated with each read/write shared object X. These locks are used only in the $\mathsf{try\_to\_commit}()$ operation, which means that neither $X.\mathsf{read}_T()$ nor $X.\mathsf{write}_T()$ is lock-based.

Variables local to each process Each process p_i manages a local variable denoted $last\_commit_i$ whose scope is the entire computation. This variable (initialized to 0) contains the commit date associated with the last transaction committed by p_i. Its aim is to ensure that the transactions committed by p_i are serialized according to their commit order.

In addition to $last\_commit_i$, a process p_i manages the following local variables whose scope is the duration of the transaction T currently executed by process p_i.

- $window\_bottom_T$ and $window\_top_T$ are two local variables that define the time interval during which transaction T could be committed. This interval is $]window\_bottom_T..window\_top_T[$ (which means that its bounds do not belong to the interval). It is initially equal to $]last\_commit_i.. + \infty[$. Then, it can only shrink. If it becomes empty (i.e., $window\_bottom_T \geq window\_top_T$), transaction T has to be aborted.
- lrs_T (resp., lws_T) is the read (resp., write) set of transaction T. Incrementally updated, it contains the identities of the transaction-level shared objects X that T has read (resp., written) up to now.
- $lcell(X)$ is a local cell whose aim is to contain the values that have been read from the cell pointed to by $PT[X]$ or will be added to that list if X is written by T. In addition to the six previous fields, it contains an additional field denoted $lcell(X).origin$ whose meaning is as follows. If X is read by T, $lcell(X).origin$ contains the value of the pointer $PT[X]$ at the time X has been read. If X is only written by T, $lcell(X).origin$ is useless.

Notation for pointers $PT[X]$, $cell(X).next$ and $lcell(X).origin$ are pointer variables. The following pointer notations are used. Let PTR be a pointer variable. $PTR\downarrow$ denotes the variable pointed to by PTR. Let VAR be a non-pointer variable. $\uparrow VAR$ denotes a pointer to VAR. Hence, $PTR \equiv \uparrow (PTR\downarrow)$ and $VAR \equiv (\uparrow VAR)\downarrow$).

operation $\mathsf{begin}_T()$:
(01) $window\_bottom_T \leftarrow last\_commit_i$; $window\_top_T \leftarrow +\infty$; $lrs_T \leftarrow \emptyset$; $lws_T \leftarrow \emptyset$.
==========
operation $X.\mathsf{read}_T()$:
(02) **if** ($\nexists$ local cell associated with the R/W shared object X) **then**
(03) allocate local space denoted $lcell(X)$;
(04) $x\_ptr \leftarrow PT[X]$;
(05) $lcell(X).value \leftarrow (x\_ptr\downarrow).value$;
(06) $lcell(X).begin \leftarrow (x\_ptr\downarrow).begin$;
(07) $lcell(X).origin \leftarrow x\_ptr$;
(08) $window\_bottom_T \leftarrow \max(window\_bottom_T, lcell(X).begin)$;
(09) $lrs_T \leftarrow lrs_T \cup X$;
(10) **for each** $(Y \in lrs_T)$ **do** $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin\downarrow).end\big)$ **end for**;
(11) **if** $(window\_bottom_T \geq window\_top_T)$ **then** $\mathsf{return}(abort)$ **end if**
(12) **end if**;
(13) return $(lcell(X).value)$.
==========
operation $X.\mathsf{write}_T(v)$:
(14) **if** ($\nexists$ local cell associated with the R/W shared object X) **then** allocate local space $lcell(X)$ **end if**;
(15) $lws_T \leftarrow lws_T \cup X$;
(16) $lcell(X).value \leftarrow v$;
(17) $\mathsf{return}(ok)$.
==========
operation $\mathsf{try\_to\_commit}_T()$:
(18) lock all the objects in $lrs_T \cup lws_T$;
(19) **for each** $(Y \in lrs_T)$ **do** $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin\downarrow).end\big)$ **end for**;
(20) **for each** $(Y \in lws_T)$ **do** $window\_bottom_T \leftarrow \max((PT[Y]\downarrow).last\_read, window\_bottom_T)$ **end for**;
(21) **if** $(window\_bottom_T \geq window\_top_T)$ **then** release all locks and disallocate all local cells; $\mathsf{return}(abort)$ **end if**;
(22) $commit\_time_T \leftarrow$ select a (random/heuristic) time value $\in]window\_bottom_T..window\_top_T[$;
(23) **for each** $(X \in lws_T)$ **do** $(PT[X]\downarrow).end \leftarrow commit\_time_T$ **end each**;
(24) **for each** $(X \in lws_T)$ **do**
(25) allocate in shared memory a new cell for X denoted $CELL(X)$;
(26) $CELL(X).value \leftarrow lcell(X).value$; $CELL(X).last\_read \leftarrow commit\_time_T$;
(27) $CELL(X).begin \leftarrow commit\_time_T$; $CELL(X).end \leftarrow +\infty$;
(28) $PT[X] \leftarrow \uparrow CELL(X)$
(29) **end for**;
(30) **for each** $(X \in lrs_T)$ **do**
(31) $(lcell(X).origin\downarrow).last\_read \leftarrow \max((lcell(X).origin\downarrow).last\_read, commit\_time_T)$
(32) **end for**;
(33) release all locks and disallocate all local cells; $last\_commit_i \leftarrow commit\_time_T$;
(34) $\mathsf{return}(commit)$.

Figure 4: Algorithm for the operations of the protocol

5.3 The $\mathsf{read}_T()$ and $\mathsf{write}_T()$ operations

When a process p_i invokes a new transaction T, it first executes the operation $\mathsf{begin}_T()$ which initializes the appropriate local variables.

The $X.\mathsf{read}_T()$ operation The algorithm implementing $X.\mathsf{read}_T()$ is described in Figure 4. When p_i invokes this operation, it returns the value locally saved in $lcell(X).value$ if $lcell(X)$ exists (lines 02 and 13). If $lcell(X)$ has not yet been allocated, p_i does it (line 03) and updates its fields $value$, $begin$ and $origin$ with the corresponding values obtained from the shared memory (lines 04-07). Process p_i then updates $window\_bottom_T$ and $window\_top_T$. These updates are as follows.

- The algorithm defines the commit time of transaction T as a point of the time line such that T could have executed all its read and write operations instantaneously as that time. Hence, T cannot be committed before a committed transaction T' that wrote the value of a shared object X read by T. According to the algorithm implementing the $\mathsf{try\_to\_commit}_T()$ operation (see line 27), the commit point of such a transaction T' is the time value kept in $lcell(X).begin$. Hence, p_i updates $window\_bottom_T$ to $\max(window\_bottom_T, lcell(X).begin)$ (line 08). X is then added to $lrst_T$ (line 09).
- Then, p_i updates $window\_top_T$ (the top side of T's commit window, line 10). If there is a shared object Y already read by T (i.e., $Y \in lrs_T$) that has been written by some other transaction T'' (where T'' is a transaction that wrote Y after T read Y), then $window\_top_T$ has to be set to $commit\_time_{T''}$ if $commit\_time_{T''} < window\_top_T$. According to the algorithm implementing the $\mathsf{try\_to\_commit}_T()$ operation, the commit point of such a transaction T'' is the date kept in $(lcell(Y).origin \downarrow).end$. Hence, for each $Y \in lrs_T$, p_i updates $window\_bottom_T$ to $\min\big(window\_top_T, (lcell(Y).origin \downarrow).end$ (line 10).

Then, if the window becomes empty, the $X.\mathsf{read}_T()$ operation entails the abort of transaction T (line 11). If T is not aborted, the value written by T' (that is kept in $lcell(X).value$) is returned (line 13).

The $X.\mathsf{write}_T(v)$ operation The algorithm implementing this operation is described at lines 14-17 of Figure 4. If there is no local cell associated with X, p_i allocates one (line 14) and adds X to lws_T (line 15). Then it locally writes v into $lcell(X).value$ (line 16) and return ok (line 17). Let us observe that no $X.\mathsf{write}_T()$ operation can entail the abort of a transaction.

5.4 The $\mathsf{try\_to\_commit}_T()$ operation

The algorithm implementing this operation is described in Figure 4 (lines 18-34). A process p_i that invokes $\mathsf{try\_to\_commit}_T()$ first locks all transaction-level shared objects X that have been accessed by transaction T (line 18). The locking of shared objects is done in a canonical order in order to prevent deadlocks.

Then, process p_i computes the values that define the last commit window of T (lines 19-20). The update of $window\_top_T$ is the same as described in the $\mathsf{read}_T()$ operation. The update of $window\_top_T$ is as follows. For each register Y that T is about to write in the shared memory (if T is not aborted before), p_i computes the date of the last read of Y, namely the date $(PT[Y] \downarrow).last\_read$. In order not to invalidate this read (whose issuing transaction has been committed), p_i updates $window\_bottom_T$ to $\max((PT[Y] \downarrow).last\_read, window\_bottom_T)$. If the commit window of T is empty, T is aborted (line 21). All locks are then released and all local cells are freed.

If T's commit window is not empty, it can be safely committed. To that end p_i defines T's commit time as a finite value randomly chosen in the current window $]window\_bottom_T..window\_top_T[$ (let us remind that the bounds are outside the window, line 22). This time function is such that no two processes obtain the same time value.

Then, before committing, p_i has to (a) apply the writes issued by T to the shared objects and (b) update the "last read" dates associated with the shared objects it has read.

a. First, for every shared object $X \in lws_T$, process p_i updates $(PT[X] \downarrow).overwrite$ with T's commit date (line 23). When all these updates have been done, for every shared object $X \in lws_T$, p_i allocates a new shared memory cell $CELL(X)$ and fills in the four fields of $CELL(X)$ (lines 25-28). Process p_i also has to update the pointer $PT[X]$ to its new value (namely $\uparrow CELL(X)$) (line 28).

b. For each register X that has been read by T, p_i updates the field $last\_read$ to the maximum of its previous value and $commit\_time_T$ (lines 30-32). (Actually, this base version of the protocol remains correct when $X \in lrs_T$ is replaced by $X \in (lrs_T \setminus lws_T)$. (As this improvement is no longer valid in the final version of the $\mathsf{try\_to\_commit}_T()$ algorithm described in Figure 7, we do not consider it in this base protocol.)

Finally, after these updates of the shared memory, p_i releases all its locks, frees the local cells it had previously allocated (line 33) and returns the value $commit$ (line 34).

On the random selection of commit points It is important to notice that, choosing randomly commit points (line 22, Figure 4), there might be "best/worst" commit points for committed transactions, where "best point" means that it allows more concurrent conflicting transactions to commit. Random selection of a commit point can be seen as an inexpensive way to amortize the impact of "worst" commit points (inexpensive because it eliminates the extra overhead of computing which point is the best).

6 Proof of the algorithm for VWC and read invisibility

Let $\mathcal{C}$ and $\mathcal{A}$ be the set of committed transactions and the set of aborted transactions, respectively. The proof consists of two parts. First, we prove that the set $\mathcal{C}$ is serializable. We then prove that that the causal past $past(T)$ of every transaction $T \in \mathcal{A}$ is serializable. In the following, in order to shorten the proofs, we abuse notations in the following way: we write "transaction T executes action A" instead of "the process that executes transaction T executes action A" and we use "$X.\mathsf{write}_T()$" as the predicate "T is a committed transaction and the operation $X.\mathsf{write}_T()$ belongs to the execution".

6.1 Proof that $\mathcal{C}$ is serializable

In order to show that $\mathcal{C}$ is serializable, we have to show that the partial order $\rightarrow_{PO}$ restricted to $\mathcal{C}$ accepts a legal linear extension. More precisely, we have to show that there exists an order $\rightarrow_S$ on the transactions of $\mathcal{C}$ such that the following properties hold:

1. $\rightarrow_S$ is a total order,
2. $\rightarrow_S$ respects the process order between transactions,
3. $\forall T1, T2 \in \mathcal{C} : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_S T2$ and,
4. $\forall T1, T2 \in \mathcal{C}, \forall X : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_S T3 \rightarrow_S T2)$.

In the following proof, $\rightarrow_S$ is defined according to the value of the $commit\_time$ variables of the committed transactions. If two transactions have the same $commit\_time$, they are ordered according to the identities of the processes that issued them.

Lemma 1 *The order $\rightarrow_S$ is a total order.*

Proof The proof follows directly from the fact that $\rightarrow_S$ is defined as a total order on the commit times of the transactions of $\mathcal{C}$. $\square_{Lemma\ 1}$

Lemma 2 *The total order $\rightarrow_S$ respects the process order between transactions.*

Proof Consider two committed transactions T and T' issued by the same process, T' being executed just after T. The variable $window\_bottom_{T'}$ of T' is initialized at $commit\_time_T$ and can only increase (during a $\mathsf{read}()$ operation at line 08, or during its $\mathsf{try\_to\_commit}()$ operation at line 20). Because $window\_bottom_{T'} > commit\_time_T$ (line 31), we have $T \rightarrow_S T'$. By transitivity, this holds for all the transactions issued by a process. $\square_{Lemma\ 2}$

Lemma 3 $\forall T1, T2 \in \mathcal{C} : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_S T2$.

Proof Suppose that we have $T1 \xrightarrow{X}_{rf} T2$ ($T2$ reads the value of X written by $T1$. After the read of X by $T2$, $window\_bottom_{T2} \geq commit\_time_{T1}$ (line 08). We then have $T1 \rightarrow_S T2$. $\square_{Lemma\ 3}$

Because a transaction locks all the objects it accesses before committing (line 27), we can order totally the committed transactions that access a given object X. Let $\xrightarrow{X}_{lock}$ denote such a total order.

Lemma 4 $X.\mathsf{write}_T() \wedge X.\mathsf{write}_{T'}() \wedge T \xrightarrow{X}_{lock} T' \Rightarrow T \rightarrow_S T'$.

Proof W.l.o.g., consider that there is no transaction T'' such that $w_{T''}(X)$ and $T \rightarrow_S T'' \rightarrow_S T'$. Because $T \xrightarrow{X}_{lock} T'$, when T' executes line 20, T has already updated $PT[x]$ and the corresponding $CELL(X)$ (because there is no T'', at this time $PT[X] \downarrow = CELL(X)$). Because $X \in lws_{T'}$, $window\_bottom_{T'} \geq (PT[X] \downarrow).last\_read \geq commit\_time_T$ (line 20). We then have $commit\_time_{T'} > commit\_time_T$ and thus $T \rightarrow_S T'$. $\square_{Lemma\ 4}$

Corollary 1 $X.\mathsf{write}_T() \wedge X.\mathsf{write}_{T'}(X) \wedge T \rightarrow_S T' \Rightarrow T \xrightarrow{X}_{lock} T'$.

Proof The corollary follows from the fact that $\rightarrow_S$ is a total order. $\square_{Corollary\ 1}$

Lemma 5 $\forall T1, T2 \in \mathcal{C}, \forall X : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_S T3 \rightarrow_S T2$.

Proof By way of contradiction, suppose that such a $T3$ exists. Again by way of contradiction, suppose that $T2 \xrightarrow{X}_{lock} T1$. This is not possible because $T2$ reads X before committing, and $T1$ writes X at the time of its commit (line 28). Thus $T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \xrightarrow{X}_{lock} T2$.

By Corollary 1, $X.\mathsf{write}_{T1}() \wedge X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_S T3 \Rightarrow T1 \xrightarrow{X}_{lock} T3$. We then have two possibilities: (1) $T3 \xrightarrow{X}_{lock} T2$ and (2) $T2 \xrightarrow{X}_{lock} T3$.

- Case $T3 \xrightarrow{X}_{lock} T2$. Let $lcell(X)$ be the local cell of $T2$ representing X. When $T2$ executes line 19), $T3$ has already updated the field end of the cell pointed by $lcell(X).origin$ with $commit\_time_{T3}$. $T2$ will then update $window\_top_{T2}$ at a smaller value than $commit\_time_{T3}$,contradicting the original assumption $T3 \rightarrow_S T2$.

- Case $T2 \xrightarrow{X}_{lock} T3$. When $T3$ executes line 20, $T2$ has already updated the field $last\_read$ of the cell pointed by $PT[X]$. $T3$ will then update $window\_bottom_{T3}$ at a value greater than $commit\_time_{T2}$, contradicting the original assumption $T3 \rightarrow_S T2$, which completes the proof of the lemma.

$\square_{Lemma\ 5}$

6.2 Proof that the causal past of each aborted transaction is serializable

In order to show that, for each aborted transaction T, the partial order $\rightarrow_{PO}$ restricted to $past(T)$ admits a legal linear extension, we have to show that there exists a total order $\rightarrow_T$ such that the following properties hold:

1. the order $\rightarrow_T$ is a total order,
2. $\rightarrow_T$ respects the process order between transactions,
3. $\forall T1, T2 \in past(T) : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$ and,
4. $\forall T1, T2 \in past(T), \forall X : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 \in past(T) : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_T T3 \rightarrow_T T2$.

The order $\rightarrow_T$ is defined as follows:
(1) $\forall T1, T2 \in past(T) \setminus \{T\} : T1 \rightarrow_T T2$ if $T1 \rightarrow_S T2$ and,
(2) $\forall T' \in past(T) \setminus \{T\} : T' \rightarrow_T T$.

Lemma 6 *The order $\rightarrow_T$ is a total order.*

Proof The proof follows directly from the fact that $\rightarrow_T$ is defined from the total order $\rightarrow_S$ for the committed transactions in $past(T)$ (part 1 of its definition) and the fact that all these transactions are defined as preceding T (part 2 of its definition).

$\square_{Lemma\ 6}$

Lemma 7 *The total order $\rightarrow_T$ respects the process order between transactions.*

Proof The proof follows from Lemma 2 and the definition of $\rightarrow_T$. $\square_{Lemma\ 7}$

Lemma 8 $\forall T1, T2 \in past(T) : T1 \rightarrow_{rf} T2 \Rightarrow T1 \rightarrow_T T2$.

Proof Because no transaction can read a value from T, we necessarily have $T1 \neq T$. When $T2 \neq T$, the proof follows from the definition of $\rightarrow_T$ and Lemma 3. When $T2 = T$, the proof follows from directly from the definition of $\rightarrow_T$. $\square_{Lemma\ 8}$

In the following lemma, we use the dual notion of the causal past of a transaction: the *causal future* of a transaction. Given a partial order $\widehat{PO}$ defined on a set of transactions, the causal future of a transaction T, denoted $future(T)$, is the set including T and all the transactions T' such that $T \rightarrow_{PO} T'$. The partial order $\widehat{PO}$ used here is the one defined in Section 3.1.

Lemma 9 $\forall T1, T2 \in past(T) : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 \in past(T) : X.\mathsf{write}_{T3}() \wedge T1 \rightarrow_T T3 \rightarrow_T T2$.

Proof For the same reasons as in Lemma 8, we only need to consider the case when $T2 = T$.

By way of contradiction, suppose that such a transaction $T3$ exists. Let it be the first such transaction to write X. Let $T4$ be the transaction in $future(T3) \cap \{T' | T' \rightarrow_{rf} T\}$ that has the biggest $commit\_time$ value. $T4$ is well defined because otherwise, $T3$ wouldn't be in $past(T)$. Let Y be the object written by $T4$ and read by T.

When T reads Y from $T4$, it updates $window\_bottom_T$ such that $window\_bottom_T \geq commit\_time_{T4}$ (line 08). From the fact that $T4 \in future(T3)$, we then have that $window\_bottom_T \geq commit\_time_{T3}$.

Either T reads Y from $T4$ and then reads X from $T1$, or the opposite. Let $last\_op$ be the latest of the two operations. During $last\_op$, T updates $window\_bottom_T$. Due to the fact that $T3 \in past(T)$, $T3$ has already updated the pointer $PT[Z]$ for some object Z (line 28), and thus has already updated the field end (line 23) of the cell pointed by $lcell(X).origin$ ($lcell(X)$ being the local cell of T representing X). T will then observe $window\_bottom_T \geq window\_top_T$ (line 11) and will not complete $last\_op$, again a contradiction, which completes the proof of the lemma. $\square_{Lemma\ 9}$

6.3 VWC and read invisibility

Theorem 2 *The algorithm presented in Figure 4 satisfies virtual world consistency and implements invisible read operations*

Proof The proof that the algorithm presented in Figure 4 satisfies virtual world consistency follows from Lemmas 1, 2, 3, 5, 6, 7, 8 and 9.

The fact that, for any shared object X and any transaction T, the operation $X.\mathsf{read}_T()$ is invisible follows from a simple examination of the text of the algorithm implementing that operation (lines 01-13 of Figure 4): there is no write into the shared memory. $\square_{Theorem\ 2}$

7 Improving the base protocol described in Figure 4

This section considers three improvements of the base protocol previously presented and proved. Those concern the following points: how the useless cells can be collected, how read operations can be made fast and how serializability can be replaced by linearizability.

7.1 Garbage collecting useless cells

This section presents a relatively simple mechanism that allows shared memory cells that have become inaccessible to be collected for recycling. This mechanism is based on the pointers $next$, two additional shared arrays, the addition of new statements to both $X.\mathsf{read}_T()$ and the $\mathsf{try\_to\_commit}_T()$, and a background task BT.

Additional arrays The first is an array of atomic variables denoted $LAST\_COMMIT[1..m]$ (remember that m is the number of sacred objects). This array is such that $LAST\_COMMIT[X]$ (which is initialed to 0) contains the date of the last committed transaction that has written into X. Hence, the statement "$LAST\_COMMIT[X] \leftarrow commit\_time_T$" is added in the **do** ... **end** part of line 23.

The second array, denoted $MIN\_READ[1..n]$, is made up of one-writer/one-reader atomic registers (let us recall that n is the total number of processes). $MIN\_READ[i]$ is written by p_i only and read by the background task BT only. It is initialized to $+\infty$ and reset to its initial value value when p_i terminates $\mathsf{try\_to\_commit}_T()$ (i.e., just before returning at line 21 or line 34 of Figure 4). When $MIN\_READ[i] \neq +\infty$, its value is the smallest commit date of a value read by the transaction T currently executed by p_i. Moreover, the following statement has to be added after line 03 of the $X.\mathsf{read}_T()$ operation:

$$MIN\_READ[i] \leftarrow \min(MIN\_READ[i], LAST\_COMMIT[X]).$$

Managing the *next* pointers When a process executes the operation $\mathsf{try\_to\_commit}_T()$ and commits the corresponding transaction T, it has to update a pointer $next$ in order to establish a correct linking of the cells implementing X. To that end, p_i has to execute $(PT[X]\uparrow).next \leftarrow\uparrow CELL[X]$ just before updating $PT[X]$ at line 28.

The background task BT This sequential task, denoted BT, is described in Figure 5. It uses a local array denoted $last_valid\_pt[1..m]$ such that $last\_valid\_pt[X]$ is a pointer initialized to $PT[X]$ (line 01). Then its value is a pointer to the cell containing the oldest value of X that is currently accessed by a transaction (this is actually a conservative value).

The body of task BT is an infinite loop (lines 02-12). BT first computes the smallest commit date still useful (line 03). Then, for every shared object X, BT scans the list from $last\_valid\_pt[X]$ and releases the space occupied by all the cells containing values of X that are no longer accessible (lines 06-09), after which it updates $last\_valid\_pt[X]$ to its new pointer value (line 10). Lines 06 and 07 uses two consecutive $next$ pointers. Those are due to the maximal concurrency allowed by the algorithm, more specifically, they prevent an $X.\mathsf{read}_T()$ operation from accessing a released cell.

It is worth noticing that the STM system and task BT can run concurrently without mutual exclusion. Hence, BT allows for maximal concurrency. The reader can also observe that such a maximal concurrency has a price, namely (as seen in line 06 where the last two cells with commit time smaller than min_useful are kept) for any shared object X, task BT allows all -but at most one- useless cells to be released.

```
(01)  init: for every X do last_valid_pt[X] ← PT[X] end for.

background task BT:
(02)  repeat forever
(03)      min_useful ← min({MIN_READ[i]}_{1≤i≤n});
(04)      for every X do
(05)          last ← last_valid_pt[X];
(06)          while ((last ≠ PT[X]) ∧ (last ↓).next ↓).next ≠ ⊥)
(07)                ∧[(((last ↓).next ↓).next ↓).commit_time < min_useful])
(08)            do temp ← last; last ← (last ↓).next; release the cell pointed to by temp
(09)          end while;
(10)          last_valid_pt[X] ← last
(11)      end for
(12)  end repeat.
```

Figure 5: The cleaning background task BT

7.2 Expediting read operations

Let us consider the invocations $X.\mathsf{read}_T()$ (those are issued by T). From the second one, none of these invocations access the shared memory. As described in Figure 4, the first invocation $X.\mathsf{read}_T()$ entails the execution of line 10 whose costs is $O(|lrs_T|)$. Interestingly, it is possible to define "favorable circumstances" in which the execution of this line can be saved when $X.\mathsf{read}_T()$ is invoked for the first time by T. Its cost becomes then $O(1)$. To that end, each process p_i is required to manage an additional local variable denoted $earliest\_read_T$ (that is initialized to $+\infty$ at line 01 of Figure 4). Line 10 of $X.\mathsf{read}_T()$ is then replaced by:

if $(lcell(X).commit\_time > earliest\_read_T)$
then Code of line 10 **else** $earliest\_read_T \leftarrow lcell(X).commit\_time$ **end if**.

Hence, the protocol allows for *fast* read operations in favorable circumstances. (It is even possible, at the price of another additional control variable, to refine the predicate used in the previous statement in order to increase the number of favorable cases. Such an improvement is described in Appendix A. It is important to notice that, while these *fast* read operations are possible when the consistency condition is VWC, they are not when it is opacity.)

7.3 From serializability to linearizability

The IR_VWC_P protocol guarantees that the committed transactions are serializable. A simple modification of the protocol allows it to ensures the stronger "linearizability" condition [12] instead of the weaker "serializability"condition. The modification assumes a common global clock that processes can read by invoking the operation $\mathsf{System.get\_time}()$. It is as follows.

- The statement $window\_bottom_T \leftarrow last\_commit_i$ at line 01 of $\mathsf{begin}_T()$ is replaced by the statement $window\_bottom_T \leftarrow \mathsf{System.get\_time}()$.
- The following statement is added just between line 19 and line 20 of $\mathsf{try\_to\_commit}_T()$ (Figure 4):
  if $(window\_top_T = +\infty)$ **then** $window\_top_T \leftarrow \mathsf{System.get\_time}()$ **end if**.

It is easy to see that these modifications force the commit time of a transaction to lie between its starting time and its end time. Let us observe that now the disjoint access parallelism property remains to be satisfied but for the accesses to the common clock.

8 Step 2: adding probabilistic permissiveness to the protocol

This section presents the final IR_VWC_P protocol that ensures virtual world consistency, read invisibility and probabilistic permissiveness. The first part describes the protocol while the second part proves its correctness.

8.1 The IR_VWC_P protocol

To obtain a protocol that additionally satisfies probabilistic permissiveness, only the operation $\mathsf{try\_to\_commit}_T()$ has to be modified. The algorithms implementing the operations $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T()$ are exactly the same as the ones described in Figure 4. The algorithm implementing the new version of the operation $\mathsf{try\_to\_commit}_T()$ is described in Figure 7. As we are about to see, it is not a new protocol but an appropriate enrichment of the previous $\mathsf{try\_to\_commit}_T()$ protocol.

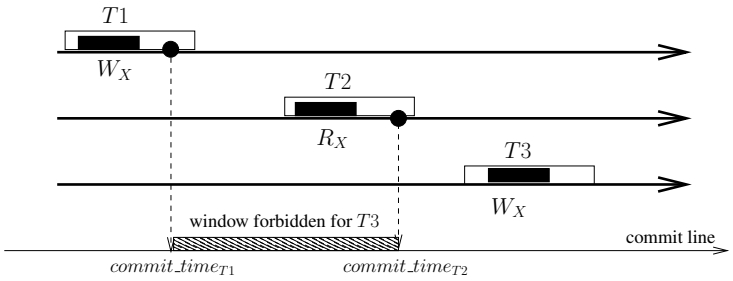

Figure 6: Commit intervals

A set of intervals for each transaction Let us consider the execution depicted in Figure 6 made up of three transactions: $T1$ that writes X, $T2$ that reads X and obtains the value written by $T1$, and $T3$ that writes X. When we consider the base protocol described in Figure 4, the commit window of $T3$ is $]commit\_time_{T2}.. + \infty[$. As the aim is not to abort a transaction if it can be appropriately serialized, it is easy to see that associating this window to $T3$ is not the best choice that can be done. Actually $T3$ can be serialized at any point of the commit line as long as the read of X by $Y2$ remains valid. This means that the commit point of $T3$ can be any point in $]0..commit\_time_{T1}[\cup]commit\_time_{T2}.. + \infty[$.

This simple example shows that, if one wants to ensure probabilistic permissiveness, the notion of continuous commit window of a transaction is a too restrictive notion. It has to be replaced by a set of time intervals in order valid commit times not to be a priori eliminated from random choices.

Additional local variables According to the previous discussion, two new variables are introduced at each process p_i. The set $commit\_set_T$ is used to contain the intervals in which T will be allowed to commit. To compute its final value, the set $forbid_T$ is used to store the windows in which T cannot be committed.

The enriched $\mathsf{try\_to\_commit}_T()$ **operation** The new $\mathsf{try\_to\_commit}_T()$ algorithm is described in Figure 7. In a very interesting way, this $\mathsf{try\_to\_commit}_T()$ algorithm has the same structure as the one described in Figure 4. The lines with the same number are identical in both algorithms, while the number of the lines of Figure 4 that are modified are postfixed by a letter. The new/modified parts are the followings.

- Lines 20.A-20.I replace line 20 of Figure 4 that was computing the value of $window\_bottom_T$. These new lines compute instead the set of intervals that constitute $commit\_set_T$. To that end they suppress from the initial interval $]window\_bottom_T, window\_top_T[$, all the time intervals that would invalidate values read by committed transactions. This is done for each object $X \in lws_T$ (line 20.H; see Appendix C for an example). If $commit\_set_T$ is empty, the transaction T is aborted (line 21.A).
- The commit time of a transaction T is now selected from the intervals in $commit\_set_T$ (line 22.A).
- Line 23 of Figure 4 was assigning, for each $X \in lws_T$, its value to $(PT[X] \downarrow).end$, namely the value $commit\_time_T$. This is now done by the new lines 23.A-23.E. Starting from $PT[X]$, these statements use the pointer $prev$ to find the cell (let us denote it say CX) of the list implementing X whose field $CX.end$ has to be assigned the value $commit\_time_T$. Let us remember that $CX.end$ defines the end of the lifetime of the value kept in $CX.value$. This cell CX is the first cell (starting from $PT[X]$) such that $CX.begin < commit\_time_T$.
- Line 24 of Figure 4 assigned its new value to every object $X \in lws_T$. Now such an object X has to be assigned its new value only if $commit\_time_T > (PT[X] \downarrow).begin$. This is because when $commit\_time_T < (PT[X] \downarrow).begin$, the value v to be written is not the last one according to the serialization order. Let us remember that the serialization order, that is defined by commit times, is not required to be real time-compliant (which would be required if we wanted to have linearizability instead of serializability, see Section 7.3). An example is given in Appendix D.

  Finally, the pointer $prev$ is appropriately updated (line 28.A). (Starting from $(PT[X] \downarrow).next$, these pointers allows for the traversal of the list implementing X.)

8.2 Proof of the probabilistic permissiveness property

In order to show that the protocol is probabilistically permissive with respect to virtual world consistency, we have to show the following. Given a transaction history that contains only committed transactions, if the partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ accepts a legal linear extension (as defined in Section 3), then the history is accepted (no operation returns abort) with positive probability. As in [7], we

```
operation try_to_commit_T():
(18)    lock all the objects in lrs_T ∪ lws_T;
(19)    for each (Y ∈ lrs_T) do window_top_T ← min(window_top_T, (lcell(Y).origin ↓).end) end for;
(20.A)  commit_set_T ← { ]window_bottom_T, window_top_T[ };
(20.B)  for each (X ∈ lws_T) do
(20.C)    x_ptr ← PT[X]; x_forbid_T[X] ← ∅;
(20.D)    while ((x_ptr ↓).last_read > window_bottom_T) do
(20.E)          x_forbid_T[X] ← x_forbid_T[X] ∪ { [(x_ptr ↓).begin, (x_ptr ↓).last_read] };
(20.F)          x_ptr ← (x_ptr ↓).prev
(20.G)    end while
(20.H)  end for;
(20.I)  commit_set_T ← commit_set_T \ ⋃_{X∈lws_T} (x_forbid_T[X]);
(21.A)  if (commit_set_T = ∅) then release all locks and disallocate all local cells; return(abort) end if;
(22.A)  commit_time_T ← select a (random/heuristic) time value ∈ commit_set_T;
(23.A)  for each (X ∈ lws_T) do
(23.B)    x_ptr ← PT[X];
(23.C)    while ((x_ptr ↓).begin > commit_time_T) do x_ptr ← (x_ptr ↓).prev end while;
(23.D)    (x_ptr ↓).end ← min((x_ptr ↓).end, commit_time_T)
(23.E)  end for;
(24.A)  for each (X ∈ lws_T) such that (commit_time_T > (PT[X] ↓).begin) do
(25)          allocate in shared memory a new cell for X denoted CELL(X);
(26)          CELL(X).value ← lcell(X).value; CELL(X).last_read ← commit_time_T;
(27)          CELL(X).begin ← commit_time_T; CELL(X).end ← +∞;
(28.A)        CELL(X).prev ← PT[X]; PT[X] ← ↑ CELL(X)
(29.A)  end for;
(30)    for each (X ∈ lrs_T) do
(31)      (lcell(X).origin ↓).last_read ← max((lcell(X).origin ↓).last_read, commit_time_T)
(32)    end for;
(33)    release all locks and disallocate all local cells; last_commit_i ← commit_time_T;
(34)    return(commit).
```

Figure 7: Algorithm for the try_to_commit() operation of the permissive protocol

consider that operations are executed in isolation. It is important to notice here that only operations, not transactions, are isolated. Different transactions can still be interlaced.

Let $\rightarrow_S$ be the order on transactions defined by the protocol according to the $commit\_time_T$ variable of each transaction T (this order has already been defined in Section 6).

Lemma 10 *Let T and T' be two committed transactions such that $T \not\rightarrow_{PO} T'$ and $T' \not\rightarrow_{PO} T$. If there is a legal linear extension of $\widehat{PO}$ in which T precedes T', then there is a positive probability that $T \rightarrow_S T'$.*

Proof Because $T \not\rightarrow_{PO} T'$, the set $past(T) \backslash past(T')$ is not empty ($past(T)$ does not contain T'). Let $biggest\_ct_{T,T'}$ be the biggest value of the $commit\_time$ variables (chosen at line 22.A) of the transactions in the set $past(T) \cap past(T')$ if it is not empty, or 0 otherwise. Suppose that every transaction in $past(T) \backslash past(T')$ chooses the smallest value possible for its $commit\_time$ variable. These values cannot be constrained (for their lower bound) by a value bigger than $biggest\_ct_{T,T'}$, thus they can all be smaller than $biggest\_ct_{T,T'} + \epsilon$ for any given ϵ.

Suppose now that T' chooses a value bigger than $biggest\_ct_{T,T'} + \epsilon$ for its $commit\_time$ variable. This is possible because, for a given transaction $T1$, the upper bound on the value of $commit\_time_{T1}$ can only be fixed by a transaction $T2$ that overwrites a value read by $T1$ (lines 10 and 19). Suppose now that $T1$ is T'. If there was such a transaction $T2$ in $past(T)$, then there would be no legal linear extension of $\rightarrow_{PO}$ in which T precedes T'. Thus if there is a legal linear extension of $\rightarrow_{PO}$ in which T precedes T', then there is a positive probability that $T \rightarrow_S T'$. $\square_{Lemma\ 10}$

Lemma 11 *Let $\widehat{PO} = (PO, \rightarrow_{PO})$ be a partial order that accepts a legal linear extension. Every operation of each transaction in PO does not return $abort$ with positive probability.*

Proof $X.\mathsf{write}_T()$ operations cannot return $abort$. Therefore, we will only consider the operations $X.\mathsf{read}_T()$ and $\mathsf{try\_to\_commit}_T()$.

Let $\rightarrow_{legal}$ be a legal linear extension of $\rightarrow_{PO}$. Let op be an operation executed by a transaction. Let $\mathcal{C}_{op}$ be the set of transactions that have ended their try_to_commit() operation before operation op is executed (because we consider that the operations are executed in isolation, this set is well defined). Let $\rightarrow_{op}$ be the total order on these transactions defined by the protocol.

From Lemma 10, two transactions that are not causally related can be totally ordered in any way that allows a legal linear extension of $\rightarrow_{PO}$. There is then a positive probability that $\rightarrow_{op} \subset \rightarrow_{legal}$. Suppose that it is true. Let T be the transaction executing op. Let $T1$

and $T2$ be the transactions directly preceding and following T in $\rightarrow_{legal}$ restricted to $\mathcal{C}_{op} \cup \{T\}$ if they exist. If $T1$ does not exist, then $window\_bottom_T = 0$ at the time of all the operation of T, and thus T can execute successfully all its operations. Similarly, if $T2$ does not exist, then $window\_top_T = +\infty$ at the time of all the operation of T, and thus T can execute successfully all its operations. We will then consider that both $T1$ and $T2$ exist.

Because $\rightarrow_{legal}$ is a legal linear extension of $\rightarrow_{PO}$, any transaction from which T reads a value is either $T1$ or a transaction preceding $T1$ in $\rightarrow_{op}$ (line 08) resulting in a value for $window\_bottom_T$ that is at most $commit\_time_{T1}$. Similarly, any transaction that overwrote a value read by T at the time of op is either $T2$ or a transaction following $T2$ in $\rightarrow_{op}$ (line 10) resulting in a value for $window\_top_T$ that is at least $commit\_time_{T2}$. All the read operations of T will then succeed (line 11).

Let us now consider the case of the try_to_commit() operation. Because all read operations have succeeded, the set $commit\_set_T =]window\_b$ (line 20.A) is not empty and must contain the set $]commit\_time_{T1}..commit\_time_{T2}[$. Because $\rightarrow_{legal}$ is a legal linear extension of $\rightarrow_{PO}$, if T writes to an object X then T cannot be placed between two transactions $T3$ and $T4$ such that $T3$ reads a value of object X written by $T4$.

Because these intervals (represented by $x\_forbid_T[X]$, lines 20.B to 20.H) are the only ones removed from $commit\_set_T$ (line 20.I) and because there is a legal linear extension of $\rightarrow_{PO}$ which includes T, $commit\_set_T$ is not empty and the transaction can commit successfully, which ends the proof of the lemma. $\square_{Lemma\ 11}$

Theorem 3 *The algorithm presented in Figure 4, where the* try_to_commit() *operation has been replaced by the one presented in Figure 7, is permissive with respect to virtual world consistency.*

Proof From Lemma 11, all transactions of a history can commit with positive probability if the history is virtual world consistent, which proves the theorem. $\square_{Theorem\ 3}$

9 Conclusion

This paper has investigated the relation linking read invisibility, permissiveness and two consistency conditions, namely, opacity and virtual world consistency. It has shown that read invisibility, permissiveness and virtual world consistency are compatible. To that end an appropriate STM protocol has been designed and proved correct. It has also been shown that this protocol allows for fast read operations while opacity does not. Interestingly enough, this new STM protocol has additional noteworthy features: (a) it uses only base read/write operations and a lock per object that is used at commit time only and (b) satisfies the disjoint access parallelism property.

The proposed IR_VWC_P protocol uses multiple versions (kept in a list) of each shared object X. Multi-version systems for STM systems have been proposed several years ago [4] and have recently received a new interest, e.g., [1, 19]. In contrast to our work, none of these papers consider virtual wold consistency as consistency condition. Moreover, both papers consider a different notion of permissiveness called multi-version permissiveness that states that read-only transactions are never aborted and an update transaction can be aborted only when in conflict with other transactions writing the same objects. More specifically, paper [19] studies inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions. This paper presents also a protocol with visible read operations that recovers useless versions. Paper [1] shows that multi-version permissiveness can be obtained from single-version. The STM protocol presented in this paper satisfies the disjoint access parallelism property, requires visible read operations and uses k-Compare&single-swap operations.

References

[1] Attiya H. and Hillel E., Single-version STM Can be Multi-version Permissive. *Proc. 12th Int'l Conference on Distributed Computing and Networking (ICDCN'11)*, Springer-Verlag, LNCS #abcd, pp. aaa-bbb, 2011.

[2] 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.

[3] 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.

[4] Cachopo J. and Rito-Silva A., Versioned Boxes as the Basis for Transactional Memory. *Science of Computer Progr.*, 63(2):172-175, 2006.

[5] 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.

[6] 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.

[7] Guerraoui R., Henzinger T.A., Singh V., Permissiveness in Transactional Memories. *Proc. 22th Int'l Symposium on Distributed Computing (DISC'08)*, Springer-Verlag, LNCS #5218, pp. 305-318, 2008.

[8] 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.

[9] 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.

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

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

[12] 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.

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

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

[15] Imbs D. and Raynal M., A versatile STM protocol with Invisible Read Operations that Satisfies the Virtual World Consistency Condition. *16th Colloquium on Structural Information and Communication Complexity (SIROCCO'09)*, Springer Verlag LNCS, #5869, pp. 266-280, 2009.

[16] Lamport L., Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558-565, 1978.

[17] Marathe V.J., Spear M.F., Heriot C., Acharya A., Eisentatt D., Scherer III W.N. and Scott M.L., Lowering the Overhead of Software Transactional Memory. *Proc 1rt ACM SIGPLAN Workshop on Languages, Compilers and Hardware Support for Transactional Computing (TRANSACT'06)*, 2006.

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

[19] Perelman D., Fan R. and Keidar I., On Maintaining Multiple versions in STM. *Proc. 29th annual ACM Symposium on Principles of Distributed Computing (PODC'10)*, ACM Press, pp. 16-25, 2010.

[20] 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.

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

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

A More on fast read operations

Virtual world consistency vs opacity Unlike opacity, a live transaction satisfying the VWC consistency criterion only has to be concerned with its causal past in order not to violate consistency. When a new transaction commits in an opaque STM a live transaction has always to consider this transaction. In contrast in a VWC STM, a live transaction needs to consider only it if it is in the live transaction's causal past.

We can take advantage of this observation in order to increase the number of times a transaction performs fast reads. This is not without a trade off though, allowing a transaction to only consider its causal past means that in certain cases transactions with no realtime-compliant legal linear extension with previously committed transactions will have their abort operation delayed. Or in other words a transaction that is doomed to abort could possibly be allowed to stay alive longer. In our STM we also have the cost of using an additional control variable. How much efficiency is gained or lost by this will certainly depend on the execution.

The method described below is not the only way to take advantage of VWC for fast reads. It has to be seen as one among several possible enhancements. The idea is that when a live transaction T reads a value written by some other transaction $T_1 \notin past(T)$,then T_1's causal past is added to $past(T)$. But if the commit time of the transaction in $past(T_1)$ with the largest commit time is smaller then the commit time of all transactions that T has read from then it is impossible for any of the transactions in $past(T_1)$ to overwrite any of T's reads. In this case only the transaction T_1 itself could overwrite a value read by T causing T to not be VWC, but this is only possible if T_1 has overwritten a value that was previously written by a transaction with commit time later then the commit time of the earliest transaction that T has read from. Thus using some extra control variables allows us to perform fast read operations in these cases.

An implementation This part discusses an implementation of the previous idea. While a transaction is live it keeps a local variable called $latest\_read_T$ initialized as $commit\_time_i$ and updated during each $X.read_T()$ operation to the largest commit time of the transactions it has read from so far. When T executes line 20 of the try_to_commit$_T()$ operation, the variable $latest\_read_T$ is (possibly) increased to the largest commit time of the transaction for the values T is overwriting. A boolean control variable, $overwrites\_latest\_read_T$ (initialized to *false*) is set to *true* if $latest\_read_T$ is modified. To that end line20 is replaced by the code described in Figure 8.

```
(13) for each (Y ∈ lws_T) do
(14)    window_bottom_T ← max((PT[Y] ↓).last_read, window_bottom_T);
(15)    if ((PT[Y] ↓).commit_time ≥ latest_read_T) then
(16)        latest_read_T ← (PT[Y] ↓).commit_time; overwrites_latest_read_T ← true
(17)    end if
(18) end for.
```

Figure 8: Fast read: Code to replace line 20 of Figure 4

Moreover, when a transaction commits, the values of $latest\_read_T$ and $overwrites\_latest\_read_T$ have now to be stored in shared memory along with the other values in $CELL(X)$ for each variable X written by T.

Finally, line 10 of $X.\mathsf{read}_T()$ is replaced by the following statement:

```
(a)  if ((CELL(X).latest_read > earliest_read_T) ∨
(b)      (CELL(X).latest_read = earliest_read_T ∧ CELL(X).overwrites_latest_read))
         then Code of line 10 end if;
     earliest_read_T ← min(lcell(X).commit_time, earliest_read_T);
     latest_read_T ← max(lcell(X).commit_time, latest_read_T).
```

Discussion It can be seen that there are two cases when update_window_top$_T()$ will be required to be executed.

- The first corresponds to line (a), i.e., when the predicate $CELL(X).latest\_read > earliest\_read_T$ (let T_1 be the transaction that wrote $CELL(X)$) is satisfied, meaning that there is at least one transaction in the causal past of T that has a commit time earlier than either a transaction in the causal past of T_1, or a transaction that has written a value overwritten by T_1. Thus a value read by T might have been overwritten by T_1 or $past(T_1)$ and update_window_top$_T()$ needs to be executed.
- The second corresponds to line (b), i.e., when the predicate $CELL(X).latest\_read = earliest\_read_T \wedge cell(X).overwrites\_latest\_read$ is true. First, as each transaction has a unique commit time, when $CELL(X).latest\_read = earliest\_read_T$ then the transactions described by $CELL(X).latest\_read$ and $earliest\_read_T$ are actually the same transaction, call this transaction T_2. So if the boolean variable $CELL(X).overwrites\_latest\_read$ is false, then T_1 just reads a value from T_2 resulting in there being no possibility of a read of T being overwritten. Otherwise if $CELL(X).overwrites\_latest\_read$ is satisfied, then T_1 is overwritten a value written by T_2 and update_window_top$_T()$ needs to be executed. In all other cases a fast read is performed.

Fast read operations and opacity As mentioned previously performing fast reads in this way is only possible in VWC. We give below a counterexample (Figure 9) in order to show that they are not compatible with opacity.

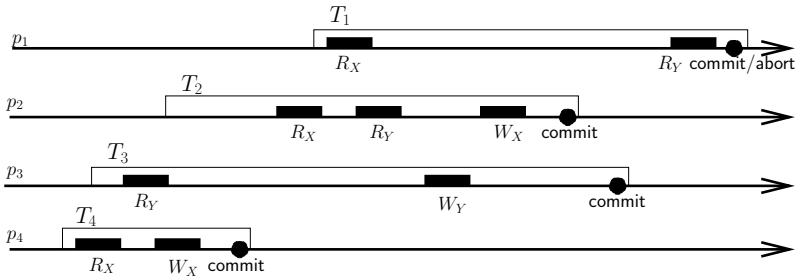

Figure 9: Example of fast reads that would violate opacity

In this figure we have T_4 committing first, then T_2 commits and must be serialized after T_4 because it reads the value of X written by T_4. Next T_3 commits and must be serialized after T_2 because it overwrites the value of Y read by T_2. Finally we have T_1 which first reads

the value of X written by T_4 then reads the value of Y written by T_3. Now T_1 violates opacity because it must be serialized before T_2 (T_2 overwrites the value of X it read) and after T_3 (it reads the value of Y written by T_3), but we already know that T_2 is serialized before T_3. On the other hand, T_1 is VWC because its causal past does not contain T_2. Now we just need to show that $\mathsf{update\_window\_top}_T()$ is not executed during the execution of $Y.\mathsf{read}_{T_1}()$. After T_1 performs $X.\mathsf{read}_{T_1}()$ the variable $earliest\_read_{T_1}$ is set to the commit time of T_4. The value $latest\_read_{T_3}$ is the commit time of transaction T_0 so when T_3 commits we have $CELL((Y).latest\_read$ also set to this value. During $Y.\mathsf{read}_{T_1}()$ we have $CELL(Y).latest\_read < earliest\_read_{T_1}$ and $\mathsf{update\_window\_top}_{T_1}()$ is not executed.

B Making read operations invisible at commit time

Read invisibility vs $\mathsf{try\_to\_commit}_T()$ **invisibility** Let us observe that read invisibility requires that read operations be invisible when they are issued by a transaction T, but does not require they remain invisible at commit time. This means that the shared memory is not modified during the read operation, but a $\mathsf{try\_to\_commit}_T()$ operation is allowed to modify shared memory locations associated with base objects read by transaction T. Interestingly though this does not always need to be the case.

Similarly to fast read operations where line 10 of $X.\mathsf{read}_T()$ does not need to be always executed, a read does not always need to be made visible during the $\mathsf{try\_to\_commit}_T()$ operation. There are two locations in $\mathsf{try\_to\_commit}_T()$ that modify shared memory with respect to objects $X \in lrs_T \setminus lws_T$. The first is on line 31 where the value of $cell(X).last\_read$ is updated for each object $X \in lrs_T$. The second is on line 18 where each object $X \in lrs_T$ is locked. Concerning efficiency and scalability, not only can locking and writing to shared memory be considered expensive operations, but is is desirable for reads to be completely invisible.

Discussion: When write to shared memory and lock can be avoided Assume some shared variable Y read by transaction T. First consider the write to $cell(Y).last\_read$. In the algorithm, the only time the $last\_read$ field of a cell is read is on line 20 of the $\mathsf{try\_to\_commit}_T()$ operation. Since $last\_read$ is only read for objects in lws_T, and T locks all objects in lws_T, T is guaranteed to be accessing the most recent cell. This means that, during a $\mathsf{try\_to\_commit}_T()$ operation on line 31, if $lcell(Y).origin$ is not the latest cell for Y, then the value of $last\_read$ written will never be accessed and there is no reason to write the value.

So the write on line 31 is not necessary when $lcell(Y).origin$ is not the latest cell, what about the lock associated with Y? When is it not necessary to lock objects in lrs_T? This turns out to not be necessary in the same case. If line 31 is not executed for some variable $Y \in lrs_T$, then the only other place a cell of Y is accessed in the $\mathsf{try\_to\_commit}_T()$ operation is on line 19. Here the only shared memory accessed is $(lcell(Y).origin \downarrow).end$. From the construction of the algorithm $(lcell(Y).origin \downarrow).end$ will be updated at most once, therefore if the loop iteration for Y on line 10 of $\mathsf{X.read}_T()$ (where X and Y may or may not be the same variable) had been executed previously with $(lcell(Y).origin \downarrow).end \neq +\infty$ (meaning $(lcell(Y).origin \downarrow).end$ had been updated previously), then the loop iteration for Y does not need to be executed again which results in Y not needing to be locked.

An implementation It follows from the previous discussion that a read operation of a shared variable Y can be made invisible at commit time if $(lcell(Y).origin \downarrow).end$ had not equal to $+\infty$ during the loop iteration for Y on line 10 for any execution of $\mathsf{X.read}_T()$. Implementing this in the algorithm becomes easy, line 10 must be replaced by the following.

for each $(Y \in lrs_T)$ **do**
  $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin \downarrow).end\big)$;
  if $((lcell(Y).origin \downarrow).end \neq +\infty)$ **then** $lrs_T \leftarrow lrs_T \setminus \{Y\}$ **end if**
end for.

Using this improvement there is a possibility for a transaction T (that performs at least one read) to have some or all of its reads to be invisible during the $\mathsf{try\_to\_commit}_T()$ operation. Consequently a read only transaction has a possibility to be completely invisible at commit time, meaning that the $\mathsf{try\_to\_commit}_T()$ operation will just immediately return $commit$ without doing any work.

Additional Benefits It is also worth noting that this improvement can also improve the performance of the read operations performed by the STM. This is because when the one of the new lines added is executed an object is removed from lrs_T, and the cost of the read operation depends on the size of lrs_T. In the best case this can cause the cost of a read operation to be $O(1)$ instead of $O(|lrs_T|)$ (note that this works concurrently along with fast reads as described in Appendix A).

C Substraction on sets of intervals (line 20.H of Figure 7)

The subtraction operation on sets of intervals of real numbers $commit\_set_T \setminus x\_forbid_T[X]$ has the usual meaning, which is explained with an example in Figure 10.

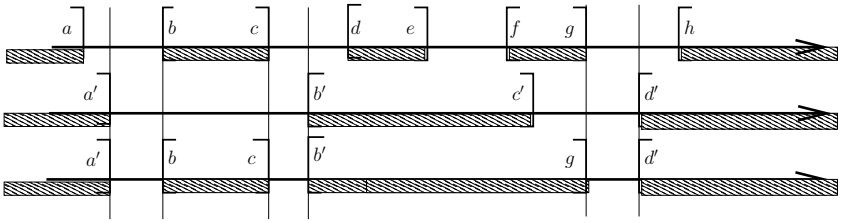

Figure 10: Subtraction on sets of intervals

The top line represents the value of $commit\_time_T$ that is made up of 4 intervals, $commit\_time_T = \{\ ]a..b[,\ ]c..d[,\ ]e..f[,\ ]g..h[\ \}$. The black intervals denote the time intervals in which T cannot be committed. The set $x\_forbid_T[X]$ is the set of intervals in which T cannot commit due to the access to X issued by T and other transactions. This set is depicted in the second line of the where we have $x\_forbid_T[X] = \{\ [0..a'],\ [b'..c'], [d'.. + \infty[\ \}$. The last line of the figure, show that we have $commit\_time_T \backslash x\_forbid_T[X] = \{\ ]a'..b[,\ ]c..b'[,\ ]g..d'[\ \}$.

D About the predicate of line 24.A of Figure 7

This section explains the meaning of the predicate used at line 24.A: $commit\_time_T > (PT[X]\ \downarrow).begin$. This predicate controls the physical write in a shared memory cell of the value v that T wants to write into X. It states that the value is written only if $commit\_time_T > (PT[X]\ \downarrow).begin$. This is due to the following reason.

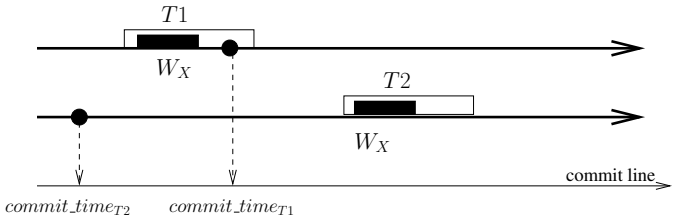

Figure 11: Predicate of line 24.A of Figure 7

Let us remember that a transaction is serialized at a random point that belongs to its current set of intervals $current\_set_T$. Moreover as we are looking for serializable transactions, the serialization points of two transactions $T1$ and $T2$ are not necessarily real time-compliant, they depend only of their sets $current\_set_{T1}$ and $current\_set_{T1}$, respectively.

An example is described in Figure 11. Transaction $T1$, that invokes $X.\mathsf{write}_{T1}()$, executes first (in real time), commits and is serialized at (logical) time $commit\_time_{T1}$ as indicated on the Figure. Then (according to real time) transaction $T2$, that invokes also $X.\mathsf{write}_{T2}()$, is invoked and then commits. Moreover, its commit set and the random selection of its commit time are such that $commit\_time_{T2} < commit\_time_{T1}$. It follows that $T2$ is serialized before $T1$. Consequently, the last value of X (according to commit times) is the one written by $T1$, that has overwritten the one written by $T2$. The predicate $commit\_time_T > (PT[X]\downarrow).begin$ prevents the committed transaction $T2$ to write its value into X in order write and read operations on X issued by other transactions be in agreement with the serialization order defined by commit times.