A Lock-based Protocol for Software Transactional Memory

Damien Imbs, Michel Raynal

I R I S A
INSTITUT DE RECHERCHE EN INFORMATIQUE ET SYSTÈMES ALÉATOIRES

PUBLICATION
INTERNE
N° 1893

A LOCK-BASED PROTOCOL FOR SOFTWARE TRANSACTIONAL MEMORY

D. IMBS M. RAYNAL

ISSN 1166-8687

IRISA
CAMPUS UNIVERSITAIRE DE BEAULIEU - 35042 RENNES CEDEX - FRANCE

IRISA

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

A Lock-based Protocol for Software Transactional Memory

D. Imbs* M. Raynal**

Systèmes communicants
Projet ASAP

Publication interne n ° 1893 — Mai 2008 — 23 pages

Abstract: The aim of a software transactional memory (STM) system is to facilitate the design of concurrent programs, i.e., programs made up of processes (or threads) that concurrently access shared objects. To that end, a STM system allows a programmer to write transactions accessing shared objects, without having to take care of the fact that these objects are concurrently accessed: the programmer is discharged from the delicate problem of concurrency management. Given a transaction, the STM system commits or aborts it. Ideally, it has to be efficient (this is measured by the number of transactions processed per time unit), while ensuring that as few transactions as possible are aborted. From a safety point of view (the one addressed in this paper), a STM system has to ensure that, whatever its fate (commit or abort), each transaction always operates on a consistent state.

STM systems have recently received a great attention. Among the proposed solutions, lock-based systems and clock-based systems have been particularly investigated. This paper presents a new lock-based STM system designed from simple basic principles. Its main features are the following: it (1) does not require the shared memory to manage several versions of each object, (2) uses neither timestamps, nor version numbers, (3) aborts a transaction only when it conflicts (with some other live transaction), (4) never aborts a write only transaction, (5) employs only bounded control variables, and (6) has no centralized contention point.

Key-words: Atomic operation, Commit/abort, Concurrency control, Consistent global state, Lock, Opacity, Shared object, Software transactional memory, Transaction.

(Résumé : tsvp)

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

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

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

Un protocole pour les mémoires transactionnelles fondé sur les verrous

Résumé : Ce rapport présente un protocole fondé sur les verrous pour les mémoires transactionnelles.

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

1 Introduction

Software transactional memory Recent advances in technology, and more particularly in multicore processors, have given rise to a new momentum to practical and theoretical research in concurrency and synchronization. Software transactional memory (STM) constitutes one of the most visible domains impacted by these advances.

Being that concurrent processes (or threads) that share data structures (base objects) have to synchronize, the transactional memory concept originates from the observation that traditional lock-based solutions have inherent drawbacks. On one side, if the set of data whose accesses are controlled by a single lock is too large (large grain), the parallelism can be drastically reduced, while, on another side, the solutions where a lock is associated with each datum (fine grain), are difficult to master and error-prone. Moreover, finding solutions that are neither large grain nor fine grain-based, usually rests on tricks from which no general solution can be extracted (when looking for a solution that could provide a general framework from which simple customized solutions could be derived).

The software transactional memory (STM) approach has been proposed in [21]. Considering a set of sequential processes that accesses shared objects, it consists in decomposing each process into (a sequence of) transactions (plus possibly some parts of code not embedded in transactions). This is the job of the programmer. The job of the STM system is then to ensure that the transactions are executed as if each was an atomic operation (it would make little sense to move the complexity of concurrent programming from the fine management of locks to intricate decompositions into transactions). So, basically, the STM approach is a structuring approach. (STM borrows ideas from database transactions; there are nevertheless fundamental differences with database transactions that are examined below [10].)

Of course, as in database transactions, the fate of a transaction is to abort or commit. (According to its aim, it is then up to the issuing process to restart -or not- an aborted transaction.) The great challenge any STM system has to take up is consequently to be efficient (the more transactions are executed per time unit, the better), while ensuring that few transactions are aborted. This is the fundamental tradeoff each STM system has to address. Moreover, in the case where a transaction is executed alone (no concurrency) or in the absence of conflicting transactions, it should not be aborted. Two transactions conflict if they access the same object and one of them modifies that object.

Consistency of a STM In the past recent years, several STM concepts have been proposed, and numerous STM systems have been designed and analyzed. On the correctness side (safety), an important notion that has been introduced very recently is the concept of opacity. That concept, introduced and formalized by Guerraoui and Kapałka [12], is a consistency criterion suited to STM executions. Its aim is to render aborted transactions harmless.

The classical consistency criterion for database transactions is serializability [17] (sometimes strengthened in "strict serializability", as implemented when using the 2-phase locking mechanism). The serializability consistency criterion involves only the transactions that are committed. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting. In a STM system, the code encapsulated in a transaction can be any piece of code and consequently a transaction has to always operate on a consistent state. To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b-c$ is different from 0 in all the consistent states. If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b = c$ (and $b = c$ defines an inconsistent state). If this occurs, the transaction raises an exception that has to be managed by the process that invoked the

corresponding transaction[1]. Such bad behaviors have to be prevented in STM systems: whatever its fate (commit or abort) a transaction has to always see a consistent state of the data it accesses. The important point is here that a transaction can (a priori) be any piece of code (involving shared data), it is not restricted to predefined patterns. This also motivates the design of STM protocols that reduce the number of aborts (even if this entails a slightly lower throughput for short transactions). Roughly speaking, opacity extends serializability to all the transactions (be them committed or aborted). Of course, a committed transaction is considered entirely. Differently, only an appropriately defined subset of an aborted transaction has to be considered.

Opacity (like serializability) states only what is a correct execution, it is a safety property. It does not state when a transaction has to commit, i.e., it is not a liveness property. Several types of liveness properties are investigated in [20].

Context of the work Among the numerous STM systems have been designed in the past years, only three of them are considered here, namely, JVSTM [8], TL2 [9] and LSA-RT [19]. (These three systems are briefly described in Appendix A.) This choice is motivated by (1) the fact that (differently from a lot of other STM systems) they all satisfy the opacity property, and (2) additional properties that can be associated with STM systems.

Before introducing these properties, we first consider underlying mechanisms on which the design of STM systems are based.

- From an operational point of view, *locks* and (physical or logical) *clocks* constitute base synchronization mechanisms used in a lot of STM systems. Locks allow mutex-based solutions. Clocks allow to benefit from the progress of the (physical or logical) time in order to facilitate the validation test when the system has to decide the fate (commit or abort) of a transaction. As a clock can always increase, clock-based systems require appropriate management of the clock values.

An important design principle that differentiates STM systems is the way they implement base objects. More specifically we have the following.

- Two types of implementation of base objects can be distinguished, namely, the *single version* implementations, and the *multi-version* implementations. The aim of the latter is to allow the commit of more (mainly read only) transactions, but requires to pay a higher price from the shared memory occupation point of view.

On a "property" side, a STM implementation can be characterized by the fact it satisfies or not important additional properties. We consider here the visibility/invisibility of the read operations, and the progressiveness property.

- Let us consider the read operations on the shared objects issued by the transactions. These read operations are *invisible* if their implementation never entails updates of the underlying control variables (kept in shared memory). It follows that, when the read operations are invisible, it is not possible for an observer -that would observe the shared memory updates- to know if a transaction has read or not some object. This property can be useful from a transaction confinement point of view.
- The *progressiveness* notion, introduced in [12], is a safety property from the commit/abort termination point of view: it defines an execution pattern that forces a transaction not to abort another one.

  As already indicated, two transactions conflict if they access the same base object and one of them updates it. The STM system satisfies the progressiveness property, if it "forcefully aborts T_1 only when there is a time t at which T_1 conflicts with another concurrent transaction (let T_2) that is not committed or aborted by time t" [12].

[1] 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.

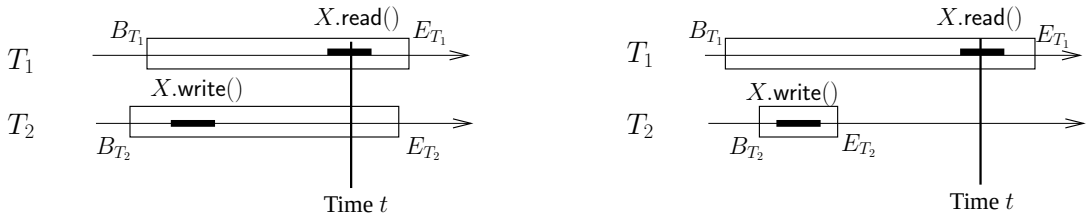

Figure 1: The progressiveness property

As an example, let us consider Figure 1 where two patterns are depicted. Both involve the same conflicting concurrent transactions T_1 that writes X, and T_2 that reads X (each transaction execution is encapsulated in a rectangle). On the left side, T_2 has not yet terminated when T_1 reads X. In that case, the progressiveness property does not prevent T_1 from being aborted due to T_2. On the right side, T_2 has terminated when T_1 reads X. In that case, if the STM system guarantees the progressiveness property, T_1 cannot be aborted due a conflict with T_2.

Finally a last criterion to compare STM systems lies in the way they cope with lower bound results related to the cost of read and write operations.

- Let k be the number of objects shared by a set of transactions. A theorem proved in [12] states the following. For any STM protocol that (1) ensures the opacity consistency criterion, (2) is based on single version objects, (3) implements invisible read operations, and (4) ensures the progressiveness property, each read/write operation issued by a transaction requires $\Omega(k)$ computation steps in the worst case. This theorem shows an inescapable cost associated with the implementation of invisible read operations as soon as we want single version objects and abort only due to conflict with a live transaction.

Considering the previous list of items (base mechanisms, number of versions, additional properties, lower bound), Table 1 indicates how each of the TL2, LSA-RT and JVSTM, behaves. While traditional comparisons of STM systems are based on efficiency measurements (usually from benchmark-based simulations), this table provides a different view to compare STM systems.

System	TL2 [9]	LSA-RT [18]	JVSTM [8]	This paper
Clock-free	no	no	no	yes
Lock-based	yes	no	yes	yes
Single version	yes	no	no	yes
Invisible read operations	yes	yes	yes	no
Progressiveness	no	yes	no	yes
Circumvent the $\Omega(k)$ lower bound	yes	no	yes	yes

Table 1: Properties ensured by protocols (that satisfy the opacity property)

Content of the paper The $\Omega(k)$ lower bound states an inherent cost for the STM systems that want to ensure invisible read operations and progressiveness while using a single version per object. When looking at Table 1, we see that, while both TL2 and JVSTM implement invisible read operations, each circumvents the $\Omega(k)$ lower bound in its own way. JVSTM uses several copies of each object and does not ensure the progressiveness property. TL2 does not ensure the progressiveness property either (it has even scenarios in which a transaction is directed to abort despite the fact it has read consistent values).

Progressiveness is a noteworthy property. As already indicated, it states circumstances where transactions must commit[2]. Considering consequently progressiveness as a first class property, this paper presents a new STM system that circumvents the $\Omega(k)$ lower bound and satisfies the progressiveness property. To that end it employs a single version per object and implements visible read operations. Moreover, differently from nearly all the STM systems proposed so far, whose designs have been driven mainly by implementation concerns and efficiency, the present paper strives for a protocol with powerful properties that can be formally proved. Its formal proof gives us a deeper understanding on how the protocol works and why it works. Combined with existing protocols, it consequently enriches our understanding of STM systems.

Finally, let us notice that the proposed protocol exhibits an interesting property related to contention management. The shared control variables associated with each object X (it is their existence that make the read operations visible) can be used by an underlying contention manager [11, 22]. If the contention manager is called when a transaction is about to commit, it can benefit from the content of these variables to decide whether to accept the commit or to abort the transaction in case this abort would entail more transactions to commit[3].

Roadmap The paper is made up of 5 sections. Section 2 describes the computation model, and the safety property we are interested in (opacity, [12]). The proposed protocol is presented incrementally. A base protocol is first presented in Section 3. This STM protocol (also called STM system in the following) associates a lock and two atomic control variables (sets) with each object X. It also uses a global control variable (a set denoted OW) that is accessed by all the update transactions (when they try to commit). This protocol is proved in Section 4. Then, Section 5 introduces the final version of the protocol. The resulting STM system has the following noteworthy features. It (1) does not require the shared memory to manage several versions of each object, (2) does not use timestamps, (3) satisfies the progressiveness property, (4) never aborts a write only transaction, (5) employs only bounded control variables, and (6) has no centralized contention point.

2 Computation model and problem specification

2.1 Computation model

Base computation model: processes, base objects, locks and atomic registers The base system (on top of which one has to build a STM system) is made up of n asynchronous sequential processes denoted $p_1, \ldots, p_n$ (a process is also sometimes denoted p) that cooperate through read/write base objects, locks, and atomic registers. The shared objects are denoted with upper case letters (e.g., the base object X). A lock, with its classical mutex semantics, is associated with each base object X.

Each process p has a local memory (a memory that can be accessed only by p). Variables in local memory are denoted with lower case letters indexed by the process id (e.g., lrs_i is a local variable of p_i).

High (user) abstraction level: transactions From a structural point of view, at the user abstraction level, each process is made up of a sequence of transactions (plus some code managing these transactions). A transaction is a sequential piece of code (computation unit) that reads/writes base objects and does local computations. At the abstraction level at which the transactions are defined, a transaction sees only base objects, it sees neither the atomic registers nor the locks. (Atomic registers and locks are used by the STM system to correctly implement transactions on top of the base model).

[2] This can be particularly attractive when there are long-lived read-only transactions

[3] Such a contention manager has to be fed with inputs (set of transactions) in order to select which transaction has to preferably be aborted. It seems more difficult to benefit from time-based STM protocols to feed a contention manager.

A transaction can be a *read-only* transaction (it then only reads base objects), or an *update* transaction (it then modifies at least one base object). A *write-only* transaction is an update transaction that does not read base objects. A transaction is assumed to be executed entirely (commit) or not at all (abort). If a transaction is aborted, it is up to the invoking process to re-issue it (as a new transaction) or not. Each transaction has its own identifier, and the set of transactions can be infinite.

2.2 Problem specification

Intuitively, the STM problem consists in designing (on top of the base computation model) protocols that ensure that, whatever the base objects they access, the transactions are correctly executed. The following property formulates precisely what "correctly executed" means in this paper.

Safety property Given a run of a STM system, let $\mathcal{C}$ be the set of transactions that commit, and $\mathcal{A}$ the set of transactions that abort. Let us assume that any transaction starts with an invocation event (B_T) and terminates with an end event (E_T).

Given $T \in \mathcal{A}$, let $T' = \rho(T)$ be the transaction built from T as follows (ρ stands for "reduced"). As T has been aborted, there is a read or a write on a base object that entailed that abortion. Let $prefix(T)$ be the prefix of T that includes all the read and write operations on the base objects accessed by T until (but excluding) the read or write that provoked the abort of T. $T' = \rho(T)$ is obtained from $prefix(T)$ by replacing its write operations on base objects and all the subsequent read operations on these objects, by corresponding write and read operations on a copy in local memory. The idea here is that only an appropriate prefix of an aborted transaction is considered: its write operations on base objects (and the subsequent read operations) are made fictitious in $T' = \rho(T)$. Finally, let $\mathcal{A}' = \{T' \mid T' = \rho(T) \ \wedge \ T \in \mathcal{A}\}$.

As announced in the Introduction, the safety property considered in this paper is *opacity* (introduced in [12] with a different formalism). It expresses the fact that a transaction never sees an inconsistent state of the base objects. With the previous notation, it can be stated as follows:

- Opacity. The transactions in $\mathcal{C} \cup \mathcal{A}'$ are linearizable (i.e., can be totally ordered according to their real-time order [13]).

This means that the transactions in $\mathcal{C} \cup \mathcal{A}'$ appear as if they have been executed one after the other, each one being executed at a single point of the time line between its invocation event and its end event.

The commit/abort vs efficiency tradeoff While both abort or commit terminate a transaction, an abort can be considered as an *unsuccessful* termination, while a commit can be considered as a *successful* termination. But, on one side, a STM system aborting all transactions would trivially satisfy the opacity property, while on another side, a STM protocol that would allow a single transaction to execute at a time would easily commit all transactions, but would be very inefficient and practically irrelevant.

Unfortunately, except for trivial cases such as concurrency-free executions, it seems very difficult to characterize in a general way (i.e., independently of the behaviors imposed by particular protocols), the cases in which a transaction should necessarily commit without compromising efficiency. The statement of such a general specification of a STM system remains an open problem [3]. It may not even be possible to find such a general formulation capturing all the cases where a transaction must commit. Hence, except for the progressiveness property, this paper restricts its analysis to protocol-dependent scenarios.

3 A lock-based STM system: base version

This section presents a base protocol that builds a STM system on top of the base system described in Section 2.1. Without ambiguity, the same identifier T is used to denote both a transaction itself and its unique name.

3.1 The STM system interface

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

- $X.\mathsf{read}_T()$ is invoked by the transaction T to read the base object X. That operation returns a value of X or the control value $abort$. If $abort$ is returned, the invoking transaction is aborted.
- $X.\mathsf{write}_T(v)$ is invoked by the transaction T to update X to the new value v. That operation never forces a transaction to immediately abort.
- If a transaction attains its last statement (as defined by the user) it executes $\mathsf{try\_to\_commit}_T()$. That operation decides the fate of T by returning $commit$ or $abort$. (Let us notice, a transaction T that invokes $\mathsf{try\_to\_commit}_T()$ has not been aborted during an invocation of $X.\mathsf{read}_T()$.)

3.2 The STM system variables

To implement the previous STM operations, the STM system uses a lock per base object X, and the following atomic control variables that are sets (all initialized to $\emptyset$).

- A read set RS_X is associated with each object X. This set contains the id of the transactions that read X. A transaction adds its id to RS_X to indicate a possibility of conflict.
- A set OW, whose meaning is the following: $T \in OW$ means that the transaction T has read an object Y and, since this reading, Y has been updated (so, there is a conflict).
- A set FBD_X per base object X (FBD_X stand for For$B$iDen). $T \in FBD_X$ means that the transaction T has read an object Y that since then has been overwritten (hence $T \in OW$), and the overwriting of Y is such that any future read of X by T will be invalid (i.e., the value obtained by T from Y and any value it will obtain from X in the future cannot be mutually consistent): reading X from the shared memory is forbidden to the transactions in FBD_X.

An example explaining the meaning of FBD_X is described in Figure 2. On the left side, the execution of three transactions are depicted (as before, each rectangle encapsulates a transaction execution). T_1 starts by reading X, executes local computation, and then reads Y. The execution of T_1 overlaps with two transactions, T_2 that is a simple write of Y, followed by T_3 that is a simple write of X. It is easy to see that the execution of these three transactions can be linearized: first T_2, then T_1 and finally T_3. In this execution, FBD_X does not include T_1.

In the execution on the right side, T_2 and T_3 are combined to form a single transaction T_4. It is easy to see that this concurrent execution of T_1 and T_4 cannot be linearized. Due to its access to X, the STM system (as we will see) will force T_4 to add T_1 to FBD_Y, entailing the abort of T_1 when T_1 will access Y (if T_1 would not access Y, it would not be aborted). Let us observe that the same thing occurs if, instead of T_4, we have (with the same timing) a transaction made up of $X.\mathsf{write}()$ followed by another transaction including $Y.\mathsf{write}()$.

The STM system also uses the following local variables (kept in the local memory of the process that invoked the corresponding transaction). lrs_T is a local set where T keeps the ids of all the objects it reads. Similarly, lws_T is a local set where T keeps the ids of all the objects it writes. Finally, $read\_only_T$ is a boolean variable initialized to $true$.

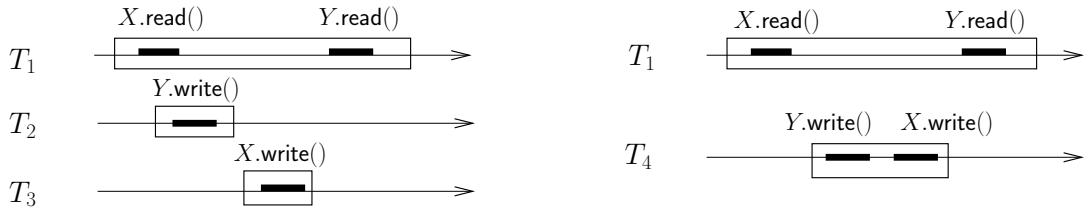

Figure 2: Meaning of the set FBD_X

The previous sets can be efficiently implemented using Bloom filters (e.g., [2, 7, 15]). In a very interesting way, the small probability of false positive on membership queries does not make the protocol incorrect (it can only affect its efficiency by entailing non-necessary aborts).

Let us recall that a process is sequential and consequently executes transactions one after the other. As local control variables are associated with a transaction, the corresponding process has to reset them to their initial values between two transactions. Similarly, if a transaction creates a local copy of an object, that copy is destroyed when the transaction terminates (a given copy of an object is meaningful for one transaction only).

3.3 The algorithms of the STM system

The three operations that constitute the STM system $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$, and $\mathsf{try\_to\_commit}_T()$, are described in Figure 3.

The operation $X.\mathsf{read}_T()$ The algorithm implementing this operation is pretty simple. If there is a local copy of X, its value is returned (lines 01 and 07). Otherwise, space for X is allocated in the local memory (line 02), X is added to the set of objects read by T (line 03), T is added to the read set RS_X of X, and the current value of X is read from the shared memory and saved in the local memory (line 04).

Due to asynchrony, it is possible that the value read by T is overwritten before T uses it. The predicate $T \in FBD_X$ is used to capture this type of read/write conflict. If this predicate is true, T is aborted (line 06). Otherwise, the value obtained from X is returned (line 07). It is easy to see that any object X is read from the shared memory at most once by a transaction.

The operation $X.\mathsf{write}_T()$ The text of the algorithm implementing the operation $X.\mathsf{write}_T()$ is even simpler than the text of $X.\mathsf{read}_T()$. The transaction first sets a flag to record that it is not a read-only transaction (line 08). If there is no local copy of X, corresponding space is allocated in the local memory (line 09); let us remark that this does not entail a reading of X from the shared memory. Finally, T updates the local copy of X (line 10), and records that it has locally written the copy of X (line 11).

It is important to notice that an invocation of $X.\mathsf{write}_T()$ is purely local: it involves no access to the shared memory, and cannot entail an immediate abort of the corresponding transaction.

The operation $\mathsf{try\_to\_commit}_T()$ This operation works a follows. If the invoking transaction is a read-only transaction, it is committed (lines 12-13). So, a read-only transaction can abort only during the invocation of a $X.\mathsf{read}_T()$ operation (line 05 of that operation).

If the transaction T is an update transaction, $\mathsf{try\_to\_commit}_T()$ first locks all the objects accessed by T (line 14). (In order to prevent deadlocks, it is assumed that these objects are locked according to a predefined total order, e.g., their identity order.) Then, T checks if it belongs to the set OW. If this is the case, there

```
operation X.read_T():
(01) if (there is no local copy of X) then
(02)    allocate local space for a copy;
(03)    lrs_T ← lrs_T ∪ {X};
(04)    lock X; local copy of X ← X; RS_X ← RS_X ∪ {T}; unlock X;
(05)    if (T ∈ FBD_X) then return(abort) end if
(06) end if;
(07) return(value of the local copy of X)
=======================================================
operation X.write_T(v):
(08) read_only_T ← false;
(09) if (there is no local copy of X) then allocate local space for a copy end if;
(10) local copy of X ← v;
(11) lws_T ← lws_T ∪ {X}
=======================================================
operation try_to_commit_T():
(12) if (read_only_T)
(13)    then return(commit)
(14)    else lock all the objects in lrs_T ∪ lws_T;
(15)         if (T ∈ OW) then release all the locks; return(abort) end if;
(16)         for each X ∈ lws_T do X ← local copy of X end for;
(17)         OW ← OW ∪ ( ∪_{X∈lws_T} RS_X );
(18)         for each X ∈ lws_T do FBD_X ← OW end for;
(19)         release all the locks;
(20)         return(commit)
(21) end if
```

Figure 3: A lock-based STM system

is a read-write conflict: T has read an object that since then has been overwritten. T consequently aborts (after having released all the locks, line 15). If the predicate $T \in OW$ is false, T will necessarily commit. But, before committing (at line 20), T has to update the control variables to indicate possible conflicts due to the objects it has written, the ids of which have been kept by T in the local set lws_T during its execution.

So, after it has updated the shared memory with the new value of each object $X \in lws_T$ (line 16), T computes the union of their read sets; this union contains all the transactions that will have a write/read conflict with T when they will read an object $X \in lws_T$. This union set is consequently added to OW (line 17), and the set FBD_X of each object $X \in lws_T$ is updated to OW (line 18). (It is important to notice that each set FBD_X is updated to OW in order not to miss the transitive conflict dependencies that have been possibly created by other transactions). Finally, before committing, T releases all its locks (line 19).

On locking As in TL2 [9], it is possible to adopt the following systematic abort strategy. When a transaction T tries to lock an object that is currentlty locked, it immediately aborts (after realeasing the locks it has, if any).

3.4 On the management of the sets RS_X, FBD_X and OW

Let us recall that these sets are kept in atomic variables.

Management of RS_X and FBD_X The set RS_X is written only at line 04 ($\mathsf{read}_T()$ operation), and (due to the lock associated with X) no two updates of RS_X can be concurrent; so, no update of RS_X is missed. Its only read (line 16) is protected by the same lock. So, there is no concurrency problem for RS_X.

The set FBD_X is read at line 06 ($\mathsf{read}_T()$ operation), and its only write (line 18, $\mathsf{try\_to\_commit}_T()$ operation) is protected by a lock. As it is an atomic variable, there is no concurrency problem for FBD_X.

Management of the set OW This set is read and written only in the operation $\mathsf{try\_to\_commit}_T()$. It is read at lines 15 and 17, and written at line 17 (its read at line 18 can benefit from a local copy saved at line 17).

Concurrent invocations of $\mathsf{try\_to\_commit}_T()$ can come from transactions accessing distinct sets of objects. When this occurs, the set OW is not protected by the locks associated with the objects and can consequently be concurrently accessed. As OW is kept in an atomic variable there is no concurrency problem for the reads. Differently, writes of OW (line 17) can be missed. Actually, when we look at the update of the atomic set variable OW, namely $OW \leftarrow OW \cup \big(\cup_{X \in lws_T} RS_X\big)$ (line 17), we can observe that this update is nothing else than a $Fetch\&Add()$ statement that has to atomically add $\cup_{X \in lws_T} RS_X$ to OW. If such an operation on a set variable is not provided by the hardware, there are several ways to implement it. One consists in using a lock to execute this operation is mutual exclusion. Another consists in using specialized hardware operations such as $Compare\&swap()$ (manipulating a pointer on the set OW, or LL/SC (load-linked/store-conditional) [14, 16]. Yet, another possible implementation consists in considering the set OW as a shared array with one entry per process, p_i being the only process that can write $OW[i]$. Moreover, for efficiency, the current value of $OW[i]$ can be saved in a local variable ow_i. A write by p_i in $OW[i]$ then becomes $ow_i \leftarrow ow_i \cup_{X \in lws_T} RS_X$ followed by $OW[i] \leftarrow ow_i$; while the atomic read of the set OW is implemented by a snapshot operation on the array $OW[1..n]$ [1] (there are efficient implementations of the snapshot operation, e.g., [4, 5]).

Differently from the pair of sets RS_X and FBD_X, associated with each object X, the set OW constitutes a global contention point. This contention point can be suppressed by replacing OW by independent boolean variables (see Section 5). We have adopted here an incremental presentation, to make the final protocol easier to understand.

3.5 Early abort and contention manager

When the predicate $T \in OW$ is satisfied, the transaction T has read an object that since then has been overwritten. This fact is not sufficient to abort T if it is a read-only transaction. Differently, if T is an update transaction, it cannot be linearized; consequently, it will be aborted when executing line 15 of $\mathsf{try\_to\_commit}_T()$. It is possible to abort such an update transaction T earlier than during the execution of $\mathsf{try\_to\_commit}_T()$. This can be simply done by adding the statement "**if** $T \in OW$ **then** $\mathsf{return}(abort)$ **end if**" just before the first line of the operation $\mathsf{write}_T()$. Similarly, the statement "**if** $T \in FBD_X$ **then** $\mathsf{return}(abort)$ **end if**" can be inserted between the first and the second line of the operation $\mathsf{read}_T()$.

Interestingly, it is important to notice that the sets RS_X, FBD_X, and OW can be used by an underlying contention manager [11, 22] to abort transactions according to predefined rules (namely, there are configurations where aborting a single transaction can prevent the abort of other transactions).

4 Proof of the base protocol

4.1 Base formalism and definitions

Events and history at the shared memory level An event is associated with the execution of each operation on the shared memory (base object, lock, set variable). We use the following notation.

- Let B_T denote the event associated with the beginning of the transaction T, and E_T the event associated with its termination. E_T can be of two types, namely A_T and C_T, where A_T is the event "abort of T" (line 05 or 15), and C_T is the event "commit of T" (line 20).
- Let $r_T(X)v$ denote the event associated with the read of X from the shared memory issued by the transaction T; v denotes the value returned by the read. Given an object X, there is a most one event $r_T(X)v$ per transaction T. If any, this read occurs at line 04 (operation $X.\mathsf{read}_T()$).
- Let $w_T(X)v$ denote the event associated with the write of the value v in X. Given an object X, there is a most one event $w_T(X)v$ per transaction T. If any, it corresponds to a write issued at line 16 in the $\mathsf{try\_to\_commit}_T()$ operation. If the value v is irrelevant $w_T(X)v$ is abbreviated $w_T(X)$.
  Without loss of generality we assume that no two writes on the same object X write the same value. We also assume that all the objects are initially written by a fictitious transaction.
- Let $AL_T(X, op)$ denote the event associated with the acquisition of the lock on the object X issued by the transaction T during an invocation of op where op is $X.\mathsf{read}_T()$ or $\mathsf{try\_to\_commit}_T()$.
  Similarly, let $RL_T(X, op)$ denote the event associated with the release of the lock on the object X issued by the transaction T during an invocation of op.

Given an execution, let H be the set of all the events generated by the shared memory accesses issued by the STM system described in Figure 3. As these shared memory accesses are atomic, the previous events are totally ordered. Consequently, at the shared memory level, an execution can be represented by the pair $\widehat{H} = (H, <_H)$ where $<_H$ denotes the total ordering on its events. $\widehat{H}$ is called a *shared memory history*.

As $<_H$ is a total order, it is possible to consider each event in H as a date of the time line. This "date" view of a sequential history on events will be used in the proof.

History at the transaction level Given an execution, let TR be the set of transactions issued during that execution. Let $\rightarrow_{TR}$ be the order relation defined on the transactions of TR as follows: $T1 \rightarrow_{TR} T2$ if $E_{T1} <_T B_{T2}$ ($T1$ has terminated before $T2$ starts). If $T1 \not\rightarrow_{TR} T2 \wedge T2 \not\rightarrow_{TR} T1$, we say that $T1$ and $T2$ are concurrent (their executions overlap in time). As the transaction level, that execution is defined by the partial order $\widehat{TR} = (TR, \rightarrow_{TR})$, that is called a *transaction level history*.

The *read-from* relation between transactions, denoted $\rightarrow_{rf}$, is defined as follows: $T1 \stackrel{X}{\rightarrow}_{rf} T2$ if $T2$ reads the value that $T1$ wrote in the object X.

A transaction history $\widehat{ST} = (ST, \rightarrow_{ST})$ is *sequential* if no two of its transactions are concurrent. Hence, in a sequential history, $T1 \not\rightarrow_{ST} T2 \Leftrightarrow T2 \rightarrow_{ST} T1$, thus $\rightarrow_{ST}$ is a total order. A sequential transaction history is *legal* if each of its read operations returns the value of the last write on the same object (because the history is sequential and transactions are executed sequentially, no two operations can overlap).

A sequential transaction history $\widehat{ST}$ is *equivalent* to a transaction history $\widehat{TR}$ if (1) $ST = TR$ (i.e., they are made of the same transactions -same invocations and same replies- in $\widehat{ST}$ and in $\widehat{TR}$), and (2) the total order $\rightarrow_{ST}$ respects the partial order $\rightarrow_{TR}$ (i.e., $\rightarrow_{TR} \subseteq \rightarrow_{ST}$).

A transaction history $\widehat{AA}$ is *linearizable* if there exists a history $\widehat{SA}$ that is sequential, legal and equivalent to $\widehat{AA}$.

Let $\widehat{\rho(TR)}$ denote the transaction history obtained from the history $\widehat{TR}$ as described in Section 2.2. This means that $\widehat{\rho(TR)}$ includes all the transactions of $\widehat{TR}$ that commit, and contains $\rho(T)$ for each transaction $T \in \widehat{TR}$ that aborts. As defined in Section 2.2, a transaction history $\widehat{TR}$ is *opaque* if there exists a transaction history $\widehat{ST}$ that is sequential, legal and equivalent to $\widehat{\rho(TR)}$.

4.2 Principle of the proof of the opacity property

According to the algorithms implementing the operations $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T(v)$ described in Figure 3, we ignore all the read operations on an object that follow another operation on the same object within the same transaction, and all the write operations that follow another write operation on the same object within the same transaction (these are operations local to the memory of the process that executes them). Building $\rho(TR)$ from TR is then a straightforward process.

To prove that the protocol described in Figure 3 satisfies the opacity consistency criterion, we need to prove that, for any transaction history $\widehat{TR}$ produced by this protocol, there is a sequential legal history $\widehat{ST}$ equivalent to $\widehat{\rho(TR)}$. This amounts to prove the following properties (where $\widehat{H}$ is the shared memory level history generated by the transaction history $\widehat{TR}$):

1. $\rightarrow_{ST}$ is a total order,
2. $\forall T \in TR : \big(T \text{ commits} \Rightarrow T \in ST\big) \wedge \big(T \text{ aborts} \Rightarrow \rho(T) \in ST\big)$,
3. $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$,
4. $T1 \xrightarrow{X}_{rf} T2 \Rightarrow \nexists T3$ such that $\big(T1 \rightarrow_{ST} T3 \rightarrow_{ST} T2\big) \wedge \big(w_{T3}(X) \in H\big)$,
5. $T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \rightarrow_{ST} T2$.

4.3 Definition of the linearization points

ST is produced by ordering the transactions according to their linearization points. The linearization point of the transaction T is denoted ℓ_T. The linearization points of the transactions are defined as follows :

- If a transaction T aborts, ℓ_T is the time just before T is added to the set OW (line 17 of the $\mathsf{try\_to\_commit}_\mathsf{T}()$ operation that entails its abort).
- If a read only transaction T commits, ℓ_T is placed at the earliest of (1) the occurrence time of the test during its last read operation (line 05 of the $X.\mathsf{read}()$ operation) and (2) the time just before it is added to OW (if it ever is). (An example is depicted in Figure 4.)
- If an update transaction T commits, ℓ_T is placed immediately after the execution of line 17 by T (update of OW).

The total order $<_H$ (defined on the events generated by $\widehat{TR}$) can be extended with these linearization points. Transactions whose linearization points happen at the same time (for example, in multi-core systems) are ordered arbitrarily.

4.4 Proof of the opacity property

Let $\widehat{TR} = (TR, \rightarrow_{TR})$ be a transaction history. Let $\widehat{ST} = (\rho(TR), \rightarrow_{ST})$ a history whose transactions are the transactions $\rho(TR)$, and such that $\rightarrow_{ST}$ is defined according to linearization points of each transaction in $\rho(TR)$. If two transactions in $\rho(TR)$ have the same linearization point, they are ordered arbitrarily. Finally, let us observe that the linearization points can be trivially added to the sequential history $\widehat{H} = (H, \rightarrow_H)$ defined on the events generated by the transaction history $\widehat{TR}$. So, we consider in the following that the set H includes the linearization points of the transactions.

Lemma 1 *$\rightarrow_{ST}$ is a total order.*

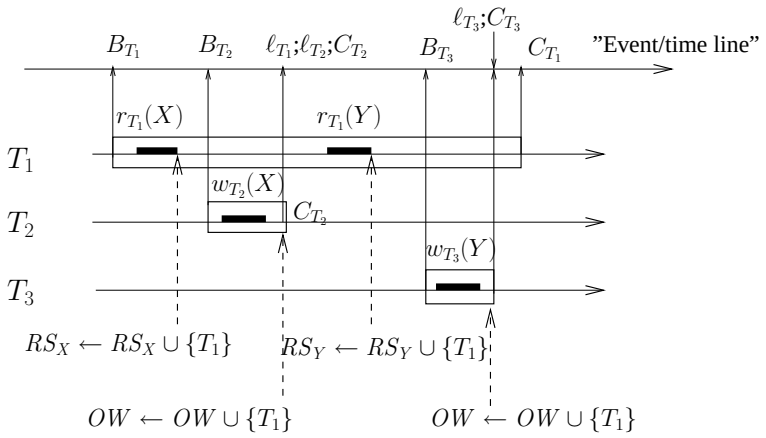

Figure 4: An example of linearization points

Proof Trivial from the ordering of the linearization points. $\square_{Lemma\ 1}$

Lemma 2 $\rightarrow_{\rho(TR)} \subseteq \rightarrow_{ST}$.

Proof This lemma follows from the fact that, given any transaction T, its linearization point is placed within its lifetime. Therefore, if $T1 \rightarrow_{\rho(TR)} T2$ ($T1$ ends before $T2$ begins), then $T1 \rightarrow_{ST} T2$. $\square_{Lemma\ 2}$

Let $ow(T,t)$ be the predicate "at time t, T belongs to OW".

Lemma 3 $ow(T,t) \Rightarrow \ell_T <_H t$.

Proof We show that the linearization point of a transaction T cannot be after the time at which the transaction's id is added to OW. There are three cases.

- By construction, if T aborts, its linearization ℓ_T is the time just before its id is added to OW, which proves the lemma.
- If T is read-only and commits, again by construction, its linearization ℓ_T point is placed at the latest just before the time at which its id is added to OW (if it ever is), which again proves the lemma.
- If T writes and commits, its linearization point ℓ_T is placed during its try_to_commit() operation, while T holds the locks of every object that it has read. If T was in OW before it acquired all the locks, it would not commit (due to line 15). Let us notice that T can be added to OW only by an update transaction holding a lock on a base object previously read by T. As T releases the locks just before committing (line 19), it follows that ℓ_T occurs before the time at which its id is added to OW (if it ever is), which proves the last case of the lemma.

$\square_{Lemma\ 3}$

Let $rs_X(T,t)$ be the predicate "at time t, T belongs to RS_X".

Lemma 4 $T_W \xrightarrow{X}_{rf} T_R \Rightarrow \nexists T'_W$ *such that* $\big(T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R\big) \wedge \big(w_{T'_w}(X) \in H\big)$.

Proof By contradiction, let us assume that there are transactions T_W, T'_W and T_R and an object X such that:

- $T_W \xrightarrow{X}_{rf} T_R$,
- $w_{T'_W}(X)v' \in H$,
- $T_W \rightarrow_{ST} T'_W \rightarrow_{ST} T_R$.

As both T_W and $T_{W'}$ write X in shared memory, they have necessarily committed (a write in shared memory occurs only at line 16 during the execution of try_to_commit(), abbreviated ttc in the following). Moreover, their linearization points ℓ_{T_W} and $\ell_{T'_W}$ occur while they hold the lock on X (before committing), from which we have the following implications:

$$\begin{array}{rcl}
T_W \rightarrow_{ST} T'_W & \Leftrightarrow & \ell_{T_W} <_H \ell_{T'_W}, \\
\ell_{T_W} <_H \ell_{T'_W} & \Rightarrow & RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T'_W}(X, \mathsf{ttc}), \\
RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T'_W}(X, \mathsf{ttc}) & \Rightarrow & w_{T_W}(X)v <_H w_{T'_W}(X)v', \\
\big(T_W \xrightarrow{X}_{rf} T_R\big) \wedge \big(w_{T_W}(X)v <_H w_{T'_W}(X)v'\big) & \Rightarrow & w_{T_W}(X)v <_H r_{T_R}(X)v <_H w_{T'_W}(X)v'.
\end{array}$$

A transaction T that reads an object X always adds its id to RS_X before releasing the lock on X. Therefore, the predicate $rs_X(T, RL_T(X, X.\mathsf{read}_T()))$ is true. Using this observation, we have the following:

$$\begin{array}{rcl}
r_{T_R}(X)v <_H w_{T'_W}(X)v' \wedge rs_X(T_R, RL_{T_R}(X, X.\mathsf{read}_{T_R}())) & \Rightarrow & rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc})), \\
rs_X(T_R, AL_{T'_W}(X, \mathsf{ttc})) \wedge \big(w_{T'_W}(X)v' \in H\big) & \Rightarrow & ow(T_R, \ell_{T'_W}), \\
\text{(Due to Lemma 3)} \quad ow(T_R, \ell_{T'_W}) & \Rightarrow & \ell_{T_R} <_H \ell_{T'_W}, \\
\text{(and finally)} \quad \ell_{T_R} <_H \ell_{T'_W} & \Leftrightarrow & T_R \rightarrow_{ST} T'_W,
\end{array}$$

which proves that, contrarily to the initial assumption, T'_W cannot precede T_R in the sequential transaction history $\widehat{ST}$. $\square_{Lemma\ 4}$

Let $fbd_X(T, t)$ be the predicate "at time t, T belongs to FBD_X".

Lemma 5 $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.

Proof The proof is made up of two parts. First it is shown that $T_W \xrightarrow{X}_{rf} T_R \Rightarrow \neg ow(T_R, \ell_{T_W})$, and then it is shown that $\neg ow(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.

Proof of $T_W \xrightarrow{X}_{rf} T_R \Rightarrow \neg ow(T_R, \ell_{T_W})$. Let us assume by contradiction that $ow(T_R, \ell_{T_W})$ is true. Due to line 18 we have

$$ow(T_R, \ell_{T_W}) \Rightarrow fbd_X(T_R, RL_{T_W}(X, \mathsf{ttc})).$$

If the read of X from shared memory by T_R is before the write by T_W, we cannot have $T_W \xrightarrow{X}_{rf} T_R$. So, in the following we consider that the read of X from shared memory by T_R is after its write by T_W. We have then $RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T_R}(X, X.\mathsf{read}_{T_R}())$, and consequently

$$fbd_X(T_R, RL_{T_W}(X, \mathsf{ttc})) \Rightarrow fbd_X(T_R, AL_{T_R}(X, X.\mathsf{read}_{T_R}())).$$

As $T_r \in FBD_X$ when it locks X, it follows that T_r aborts at line 05 and consequently we cannot have $T_W \xrightarrow{X}_{rf} T_R$. Summarizing the previous reasoning we have $(ow(T_R, \ell_{T_W}) \Rightarrow \neg(T_W \xrightarrow{X}_{rf} T_R))$, and taking the contrapositive we finally obtain

$$T_W \xrightarrow{X}_{rf} T_R \Rightarrow \neg ow(T_R, \ell_{T_W}).$$

Proof of $\neg ow(T_R, \ell_{T_W}) \wedge T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$. As defined in Section 4.3, the linearization point ℓ_{T_R} depends on the fact that T_R commits or aborts, and is a read only or an update transaction. The proof considers the three possible cases.

- If T_R is an update transaction that commits, its linearization point ℓ_{T_R} (that is defined as line 17 when it updates the set OW) occurs after its invocation of try_to_commit(). Due to this observation, the fact that T_W releases its locks after its linearization point, and $T_W \xrightarrow{X}_{rf} T_R$, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.
- If T_R is a (read only or update) transaction that aborts, its linearization point ℓ_{T_R} is the time at which T_R is added to OW. Because $T_W \xrightarrow{X}_{rf} T_R$ we have $\neg ow(T_R, \ell_{T_W})$. Moreover, due to $\neg ow(T_R, \ell_{T_W})$ and the fact that T_R aborts, we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$. It follows that $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$.
- If T_R is a read only transaction that commits, its linearization point ℓ_{T_R} is placed either at the time at which it is added to OW (then the case is the same as a transaction that aborts, see before), or at the time of the test during its last read operation (line 05). In the latter case, we have $w_{T_W}(X)v <_H \ell_{T_W} <_H RL_{T_W}(X, \mathsf{ttc}) <_H AL_{T_R}(X, X.\mathsf{read}_{T_R}()) <_H r_{T_R}(X)v <_H \ell_{T_R}$, from which we have $\ell_{T_W} <_H \ell_{T_R}$, i.e., $T_W \rightarrow_{ST} T_R$.

Hence, in all cases, we have $T_W \xrightarrow{X}_{rf} T_R \Rightarrow T_W \rightarrow_{ST} T_R$. $\square_{Lemma\ 5}$

Theorem 1 *Every transaction history $\widehat{TR}$ produced by the protocol described in Figure 3 satisfies the opacity consistency criterion.*

Proof The proof follows from the construction of the set $\rho(TR)$ (Section 4.1), the definition of the linearization points (Section 4.3), and the Lemmas 1, 2, 4 and 5. $\square_{Theorem\ 1}$

4.5 On the termination of transactions

It is easy to see that each transaction terminates. Concerning the fact that a transaction terminates successfully (commit) or not (abort), we have the following properties.

- If a transaction $T1$ entails the abort of a transaction $T2$, then $T1$ necessarily commits.
  This is because the abort of $T2$ occurs at line 05 (test $T2 \in FBD_X$) or at line 15 (test $T2 \in OW$). In both cases the addition of $T2$ to the set FBD_X or OW is done by a transaction $T1$ while it executes the lines 17 or 18, i.e., just before $T1$ commits at line 20.
- If none of the values it has read is overwritten, an update transaction cannot abort. Moreover, a write-only transaction never aborts.
- A transaction that reads an object X, is not necessarily aborted, despite the fact that it has previously read an object Y that is now overwritten, as long as the values of X and Y it has obtained belong to the same consistent state.

5 A lock-based STM system: final version

5.1 The improvements

From transaction ids to process ids While this is not necessary from a correctness point of view, it is desirable that the identities of transactions that have terminated (committed or aborted) be suppressed from

the sets RS_X, FBD_X, and OW. Moreover, the fact that the domain of these identities is unbounded can become a real drawback. As there a is a fixed number of processes, and each process issues one transaction at a time, a solution consists in using the id of the issuing process as the id of the corresponding transaction. From the point of view of transaction ids, this means that they are now recyclable. This recycling can be obtained with an appropriate update of the relevant control variables each time a transaction terminates.

Eliminating the contention point OW The set OW can actually be replaced by a set of atomic boolean variables, one per process p_i. The boolean associated with p_i, denoted OW_i, has the following meaning: $OW_i = true$ means that the current transaction issued by p_i has read an object whose value is no longer up to date.

The resulting improvement It follows from the previous improvements that all the shared control variables do have a bounded domain. They are either boolean variables, or set variables that contain at most n process ids. Let us remark that there are very efficient management algorithms for such sets.

5.2 The final (improved) STM system

The algorithms implementing the $X.\mathsf{read}_i()$, $X.\mathsf{write}_i(v)$ and $\mathsf{try\_to\_commit}_i()$ operations of the improved system are described in Figure 5. To emphasize the incremental presentation, the lines that are not modified (but for the use of the process id) have the same number in Figure 3 and Figure 5. Differently, the lines that are modified keep the same number but are postfixed with a letter.

The three operations use the id i of the invoking process p_i as the transaction id. They also use an internal operation denoted $\mathsf{init}_i()$ the aim of which is to reset control variables and suppress the id i of the invoking process from the sets it belongs to. The local variables lws_i, lrs_i and $read\_only_i$ replace their counterparts used in the base algorithms. Once this replacement has been done, the algorithms for $X.\mathsf{read}_i()$ and $X.\mathsf{write}_i(v)$ are verbatim the same as before.

The code of the $\mathsf{try\_to\_commit}_i()$ operation is the same as in Figure 3 with three modifications. The first concerns line 15: the test $T \in OW$ is replaced by the test of the boolean OW_i (line 15.a). The second modification concerns the update of the set OW (line 17 in Figure 3). This update now consists in setting to $true$ the boolean OW_j associated with each process p_j that belongs to the read sets RS_X of the objects X written by p_i. This modified update appears at line 17.a of Figure 5. The last modification concerns the sets FBD_X associated with the objects written by p_i (line 18 in Figure 3). These updates are now based on the booleans OW_j instead of the set OW; they appear at lines 18.a, 18.b and 18.c in Figure 5.

Finally, the lines A1-A3 describe the internal $\mathsf{init}_i()$ operation.

Acknowledgments

We would like to thank Pascal Felber for providing us with good advices and nice food on STM systems.

References

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

[2] Almeida P.SD., Baquero C., Preguiça N. and Hutchinson D., Scalable Bloom Filters. *Information Processing Letters*, 101(6):255-261, 2007.

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

```
operation X.read_i():
(01) if (there is no local copy of X) then
(02)    allocate local space for a copy;
(03)    lrs_i ← lrs_i ∪ {X};
(04)    lock X; local copy of X ← X; RS_X ← RS_X ∪ {i}; unlock X;
(05)    if (i ∈ FBD_X) then init_i(); return(abort) end if
(06) end if;
(07) return(value of the local copy of X)
=========================================================
operation X.write_i(v):
(08) read_only_i ← false;
(09) if (there is no local copy of X) then allocate local space for a copy end if;
(10) local copy of X ← v;
(11) lws_i ← lws_i ∪ {X}
=========================================================
operation try_to_commit_i():
(12) if (read_only_i)
(13)    then return(commit)
(14)    else lock all the objects in lrs_i ∪ lws_i;
(15.a)       if OW_i then release all the locks; init_i(); return(abort) end if;
(16)         for each X ∈ lws_i do X ← local copy of X end for;
(17.a)       for each j ∈ ( ∪_{X∈lws_i} RS_X ) do OW_j ← true end for;
(18.a)       for each X ∈ lws_i do
(18.b)          for each j such that OW_j do FBD_X ← FBD_X ∪ {j} end for
(18.c)       end for;
(19)         release all the locks;
(20.a)       init_i(); return(commit)
(21) end if
=========================================================
operation init_i():
(A1) for each X ∈ lrs_i do RS_X ← RS_X \ {i} end for;
(A2) OW_i ← false; lws_i ← ∅; lrs_i ← ∅; read_only_i ← true;
(A3) for each X such that (i ∈ FBD_X) do FBD_X ← FBD_X \ {i} end for
```

Figure 5: The improved STM system (bounded variables and no centralized contention point)

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

[5] Attiya H. and Rachman O., Atomic Snapshots in O($n \log n$) Operations. *SIAM Journal on Computing*, 27(2):319-340, 1998.

[6] Avni H. and Shavit N., Maintaining Consistent Transactional States without a Global Clock. *Proc. 15th Colloquium on Structural Information and Communication Complexity (SIROCCO'08)*, To appear, 2008.

[7] Bloom B.H., Space/Time Tradeoffs in Hash-coding with Allowable Errors. *Communications of the ACM*, 13(7):422-426, 1970.

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

[9] 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.

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

[11] Guerraoui R., Herlihy M.P. and Pochon S., Towards a Theory of Transactional Contention Managers. *Proc. 24th ACM Symposium on Principles of Distributed Computing (PODC'05)*, ACM Press, pp. 258-264, 2005.

[12] 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.

[13] 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.

[14] Jayanti P. and Petrovic S., Efficient and Practical Constructions of LL/SC Variables. *Proc. 22th ACM Symp. on Principles of Dist. Computing (PODC'03)*, ACM Press, pp. 285-294, 2003.

[15] Mitzenmatcher M., Compressed Bloom Filters. *IEEE Transaction on Networks*, 10(5):604-612, 2002.

[16] Moir M., Practical Implementation of Non-Blocking Synchronization Primitives. *Proc. 16th ACM Symposium on Principles of Distributed Computing (PODC'97)*, ACM Press, pp. 219-228, 1997.

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

[18] Riegel T., Felber P. and Fetzer C. A Lazy Snapshot Algorithm with Eager Validation. *Proc. 20th Int'l Symposium on Distributed Computing (DISC'06)*, Springer-Verlag, LNCS #4167, pp. 284-298, 2006.

[19] 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.

[20] Scott L.M., Sequential Specification of Transactional Memory Semantics. *Proc. First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Trans. Computing (TRANSACT'06)*, ACM Press, 2006.

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

[22] Scherer W.N.III and Scott M.L., Advanced Contention Management in Dynamic Software Transactional Memory. *Proc. 24th ACM Symp. on Principles of Dist. Computing (PODC'05)*, ACM Press, pp. 240-248, 2005.

A A quick view of JVSTM, TL2 and LSA-RT

Numerous STM protocols have been proposed. This section presents the principles that underlie the three of them that appear in Table 1, namely, JVSTM, TL2 and LSA-RT. All satisfy the opacity property (it is interesting to notice that they have all been defined before the opacity property has been stated). Differently, only LSA-RT satisfies the progressiveness property.

The JVSTM system [8] This system uses a logical clock to timestamp the transactions and allow them to read an appropriate version of an object. As just indicated, JVSTM satisfies the progressiveness property.

The validation phase of the transactions (i.e., the part corresponding to the try_to_commit() operation) is executed in mutual exclusion: they are encapsulated with the "`synchronized`" construct provided by JAVA, the implementation of which is lock-based. Interestingly (due to the multiplicity of versions), the read only transactions are never aborted.

Each object is implemented by a list of versions. JVSTM provides a garbage collector mechanism that discards the versions older than the oldest transaction still in the system.

The TL2 system [9] This system aims at reducing the synchronization cost due to the read, write and validation (i.e., try_to_commit()) operations. To that end, it associates a lock with each data (object) and uses a logical global clock (integer) that is read by all the transactions and increased by each writing transaction that commits. This global clock is basically used to validate the consistency of the state of the data a transaction is working on. The TL2 protocol is particularly efficient when there are few (or no) conflicts between concurrent transactions. This clock is a shared control variable that constitutes a centralized contention point. This drawback is eliminated in an improved version of the protocol described in [6], called TLC[4].

[4] In the protocol described in [6], each process p manages a local clock and maintains a clock array with an entry per process q. That entry contains the greatest clock value used by q as known by p. A process associates its current clock value (timestamp) with each value its writes. A process p updates its local clock array entry associated with q when it reads a value written by q. Let us observe that the timestamp value it reads is smaller or equal to q's actual clock value.

On the safety side, both TL2 and TLC ensure the opacity property. On the liveness side, the performance study depicted in [9] (based on a red-black tree benchmark) shows that the TL2 protocol is pretty efficient. It has nevertheless scenarios in which a transaction is directed to abort despite the fact it has read values from a consistent state[5]. Its distributed clock version [6] is less efficient, but this is not counter-intuitive as some price has to be paid for eliminating the global clock. Due to the management of the distributed clock, there are cases where this protocol directs a transaction to abort despite the fact that it is executed in a concurrency-free context.

The LSA-RT system [18, 19] This system associates a time interval with each transaction. The idea is here to use time to reason on the consistency of the data accessed by a transaction by associating a time interval with each transaction. This time interval is updated (shrunk) when the transaction accesses an object (an object keeps the date of its last update). Basically, a transaction has to abort when its interval becomes empty. The protocol described in [18] presents a lazy snapshot algorithm (called LSA) that efficiently constructs an always consistent snapshot for transactions. This protocol is based on a logical global clock that counts the number of writing transactions that have been committed so far. The protocol described in [19] (called LSA-RT) replaces the logical clock (counter) with (externally synchronized) real-time clocks. Moreover, the LSA-RT protocol manages several versions of each data in order to have invisible read operations, and abort transactions only in case of conflicts.

B A short algorithmic description of TL2

A very schematic description of the $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$ and $\mathsf{try\_to\_commit}_T()$ operations used in TL2 [9] are described in Figure 6 for an update transaction and in Figure 7 for a read only transaction (a transaction that starts as a read only transaction and finally issues a write can be aborted). As indicated previously, the protocols implementing these operations are based on a global (logical) clock and a lock per object.

The global clock is denoted $Clock$. It is increased (with an atomic $Fetch\&Increment()$ operation) each time an update transaction T invokes $\mathsf{try\_to\_commit}_T()$ (line 13). Moreover, when a transaction starts it invokes the additional operation $\mathsf{begin}_T()$ to obtain a birthdate (defined as the current value of the clock).

At the implementation level, an object X is made up of two fields: a data field $X.value$ containing the current value of the object, and a control field $X.date$ containing the date at which that value was created (see line 14 of $\mathsf{try\_to\_commit}_T()$).

The case of an update transaction Each transaction T manages two sets (as in the proposed protocol): a local read set lrs_T, and a local write set lws_T.

As far as a $X.\mathsf{read}_T()$ is concerned we have the following. If, previously, a local copy lx of the object X has been created by an invocation of $X.\mathsf{write}_T(v)$ issued by the same transaction, its value is returned (lines 01-02). Otherwise, X is read from the shared memory, and X's id is added to the local read set lrs_T (line 04). Finally, if the date associated with the current value of X is greater than the birthdate of T, the transaction is aborted (line 05). (This is because, T has possibly read other values that are no longer consistent with the value of X just obtained.) If the date associated with the current value of X is not greater than the birthdate of T, that value is returned by the $X.\mathsf{read}_T()$ operation. (In that case, the value read is consistent with the values previously read by T.)

[5] These scenarios depend on the value of the global clock. They can occur when, despite the fact that that all the values read by a transaction T are mutually consistent, one of them has been written by a transaction concurrent with T.

```
operation T.begin_T(): birthdate ← Clock
================================================================
operation X.read_T():
(01) if (there is a local copy of X)
(02)    then return (lx.value) % the local copy lx was created by a write of X %
(03)    else tlcx ← copy of X read from the shared memory;
(04)         lrs_T ← lrs_T ∪ {X};
(05)         if tlcx.date > birthdate then return (abort) else return (tlcx.value) end if
(06) end if
================================================================
operation X.write_T(v):
(07) if (there is no local copy of X) then allocate local space lx for a copy end if;
(08) lx.value ← v;
(09) lws_T ← lws_T ∪ {X}
================================================================
operation try_to_commit_T():
(10)    lock all the objects in (lrs_T ∪ lws_T);
(11)    for each X ∈ lrs_T do % the date of X is read from the shared memory %
(12)       if X.date > birthdate then release all the locks; return (abort) end if
(13)    end for;
(14)    commit_date ← Fetch&Increment(Clock);
(15)    for each X ∈ lws_T do X ← (lx.value, commit_date) end for;
(16)    release all the locks;
(17)    return (commit)
```

Figure 6: TL2 algorithm for an update transaction

The operation $X.\mathsf{write}_T(v)$ in TL2 and the one in the proposed protocol are similar. If there is no local copy of X, one is created and its value field is set to v. The local write set lws_T is also updated to remember that X has been written by T. The lifetime of the local copy lx of X created by a $X.\mathsf{write}_T(v)$ operation spans the duration of the transaction T.

When a transaction T invokes $\mathsf{try\_to\_commit}_T()$ it first locks all the objects in $lrs_T \cup lws_T$. Then, T checks if the current values of the objects X is has read are still mutually consistent, and consistent with respect to the new values it has (locally) written). This is done by comparing the current date $X.date$ of each object X that has been read to the birthdate of T. If one of these dates is greater than its birthdate, there is a possible inconsistency and consequently T is aborted (line 12). Otherwise, T can be committed. Before being committed (line 17), T has to set the objects it has written to their new values (line 15). Their control part has also to be updated: they are set to the last clock value obtained by T (line 14). Finally, T releases the locks and commits.

Remark This presentation of the $\mathsf{try\_to\_commit}_T()$ operation of TL2 does not take into account all of its aspects. As an example, if at line 10, all the locks cannot be immediately obtained, TL2 can abort the transaction (and restart it later). This can allow for more efficient behaviors. Moreover, the lock of an object is used to contain its date value (this allows for more efficient read operations.)

The case of a read only transaction If a transaction T does not modify the shared objects, the code of its $X.\mathsf{read}_T()$ and $\mathsf{try\_to\_commit}_T()$ operations simplify as shown in Figure 7.

Each time a read only transaction T reads an object X, it obtains its value from the shared memory. Then, if the date associated with the value it has read is not greater than its birthdate, the read is valid and the value is returned. Otherwise, T is aborted. Let us notice that a read only transaction does not manage a local read set.

```
operation begin_T(): birthdate ← Clock
==========================================================
operation X.read_T():
(01) lx ← copy of X read from the shared memory;
(02) if lx.date > birthdate then return (abort) else return (lx.value) end if
==========================================================
operation try_to_commit_T(): return(commit)
```

Figure 7: TL2 algorithm for a read-only transaction

Finally, the operation $\mathsf{try\_to\_commit}_T()$ always returns $commit$. This is because, if a read only transaction T has never been aborted when it read base objects, we can conclude that T has read object values that are mutually consistent. They define a consistent snapshot of the shared objects that have been accessed [1].

C A short algorithmic description of JVSTM

A very schematic description of the $X.\mathsf{read}_T()$, $X.\mathsf{write}_T(v)$ and $\mathsf{try\_to\_commit}_T()$ operations used in JVSTM [8] are described in Figure 8 for an update transaction and in Figure 9 for a read only transaction (as for TL2, a transaction that starts as a read only transaction and finally issues a write can be aborted). These protocols are very informally described in [8]. These informal descriptions include a lot of improvements that are not considered here.

As indicated previously, the protocols implementing these operations are based on a global (logical) clock and a global lock. The global clock, denoted $Clock$, is increased when an update transaction T invokes $\mathsf{try\_to\_commit}_T()$ (lines 13 and 15). Since, due to the use of a global lock, the $\mathsf{try\_to\_commit}()$ operations are executed in mutual exclusion, there are no concurrent updates of $Clock$. Moreover, (as in TL2) when a transaction starts, it invokes the additional operation $\mathsf{begin}_T()$ to obtain a birthdate (defined as the current value of the clock).

At the implementation level, an object X is a pointer on a list of records made up of three fields: a data field $X.value$ containing the current value of the object, a control field $X.date$ containing the date at which that value was created (see line 15 of $\mathsf{try\_to\_commit}_T()$) and a pointer on the next structure in the list.

If Y is a variable containing a pointer, $(\downarrow Y)$ denotes the variable pointed by Y. If X is a variable, $(\uparrow X)$ denotes a pointer on X.

The case of an update transaction Each transaction T manages two sets (as in the proposed protocol): a local read set lrs_T, and a local write set lws_T.

As far as a $X.\mathsf{read}_T()$ operation is concerned we have the following. If, previously, a local copy lx of the object X has been created by an invocation of $X.\mathsf{write}_T(v)$ issued by the same transaction, its value is returned (lines 01-02). Otherwise, X's id is added to the local read set lrs_T (line 04) and the most recent value, at least as old as the transaction's birthdate, is fetched from the list of versions of X (lines 05-06) (so the set of values read is consistent). Let us note that, due the multiplicity of versions, a $X.\mathsf{read}_T()$ operation never forces a transaction to abort.

The operation $X.\mathsf{write}_T(v)$ in JVSTM and the one in the proposed protocol are similar. If there is no local copy of X, one is created and its value field is set to v. The local write set lws_T is updated accordingly. As in the previous protocols, the lifetime of the local copy lx of X created by a $X.\mathsf{write}_T(v)$ operation spans the duration of T.

When a transaction T invokes $\mathsf{try\_to\_commit}_T()$ it first acquires the global lock (line 11). Then, T checks if the current values of the objects X has read are the most recent, and consistent with respect to the

```
operation X.begin_T(): birthdate ← Clock
================================================================================
operation X.read_T():
(01)  if (there is a local copy lx of X)
(02)  then return (lx.value)
(03)  else  lrs_T ← lrs_T ∪ {X};
(04)        x ← X; % pointer on the list of versions of the object %
(05)        while ((↓ x).date > birthdate) do x ← (↓ x).next end while;
(06)        return ((↓ x).value)
(07)  end if
================================================================================
operation X.write_T(v):
(08)  if (there is no local copy of X) then allocate space lx for a copy end if;
(09)  lx.value ← v;
(10)  lws_T ← lws_T ∪ {X}
================================================================================
operation try_to_commit_T():
(11)  acquire the lock; % commits are mutually exclusive %
(12)  for each X ∈ lrs_T do if (↓ X).date > birthdate then release lock; return (abort) end if end for;
(13)  commit_date ← Clock + 1;
(14)  for each lx ∈ lws_T do lx.next ← X; lx.date ← commit_date; X ← (↑ lx) end for;
(15)  Clock ← commit_date;
(16)  release the lock;
(17)  return (commit)
```

Figure 8: JVSTM's algorithm for an update transactions

new values it has (locally) written. This is done by comparing the current date $X.date$ of each object X that has been read to the birthdate of T (line 12). If one of these dates is greater than its birthdate, there is a possible inconsistency and consequently T is aborted (line 12). Otherwise, T can be committed. Before being committed, T has to insert the new versions of the objects it has written into their lists of versions (line 14). Their control part has also to be updated: they are set to the current clock value obtained by T (incremented by one as shown in line 13). Finally, T increments the clock, releases the global lock and commits (lines 15-17).

Remark This presentation of the try_to_commit$_T()$ operation of JVSTM does not take into account all of its aspects. Among those, there is a mechanism that keeps track of the birthdates of the transactions that are not committed yet. A transaction that commits then discards all the versions of the objects it writes that are no longer accessible (that is, that will never be read by any live transaction because they are too old). The JAVA garbage collector then frees the corresponding memory locations.

```
operation X.init_T(): birthdate ← Clock
================================================
operation X.read_T():
(01)  x ← X;
(02)  while ((↓ x).date > birthdate) do x ← (↓ x).next end while;
(03)  return ((↓ x).value);
================================================
operation try_to_commit_T(): return (commit)
```

Figure 9: JVSTM's algorithm for a read-only transactions

The case of a read only transaction If a transaction T does not modify the shared objects, the code of its $X.\mathsf{read}_T()$ and $\mathsf{try\_to\_commit}_T()$ operations can be simplified as shown in Figure 9.

Each time a read only transaction T reads an object X, it obtains (from the shared memory) the most recent value at least as old as the transaction's birthdate. A read only transaction does not manage a local read set.

Finally, the operation $\mathsf{try\_to\_commit}_T()$ always returns $commit$. This is because a read only transaction can never abort. This is due to the presence of multiple versions which always allow a read only transaction T to obtain mutually consistent versions of all the objects it has read.