Publications Internes de l'IRISA
ISSN : 2102-6327
PI 1971 – Mars 2011

Towards a universal construction for transaction-based multiprocess programs*

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) system is to discharge the programmer from the explicit management of synchronization issues. The programmer's job resides in the design of multiprocess programs in which processes are made up of transactions, each transaction being an atomic execution unit that accesses concurrent objects. The important point is that the programmer has to focus her/his efforts only on the parts of code which have to be atomic execution units without worrying on the way the corresponding synchronization has to be realized.

Non-trivial STM systems allow transactions to execute concurrently and rely on the notion of commit/abort of a transaction in order to solve their conflicts on the objects they access simultaneously. In some cases, the management of aborted transactions is left to the programmer. In other cases, the underlying system scheduler is appropriately modified or an underlying contention manager is used in order that each transaction be ("practically always" or with high probability) eventually committed.

This paper presents a deterministic STM system in which (1) every invocation of a transaction is executed exactly once and (2) the notion of commit/abort of a transaction remains unknown to the programmer. This system, which imposes restriction neither on the design of processes nor or their concurrency pattern, can be seen as a step in the design of a deterministic universal construction to execute transaction-based multiprocess programs on top of a multiprocessor. Interestingly, the proposed construction is lock-free (in the sense that it uses no lock).

Key-words: Abort/commit, Asynchronous system, Atomic execution unit, Compare&Swap, Concurrency management, Fetch&Increment, Lock-freedom, Shared memory system, STM system, Transaction, Universal construction.

Une construction universelle pour les programmes parallèles fondés sur les transactions

Résumé : *Ce rapport présente une construction universelle pour les programmes parallèles fondés sur les transactions.*

Mots clés : *Système transactionnel, construction universelle.*

---

* This research is part of the Marie Curie ITN project TRANSFORM funded by the European Union FP7 Program (grant 238639).
** 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
**** Membre senior de l'Institut Universitaire de France. Projet ASAP: équipe commune avec l'INRIA, le CNRS, l'université Rennes 1 et l'INSA de Rennes

1 Introduction

Lock-based concurrent programming A *concurrent object* is an object that can be concurrently accessed by different processes of a multiprocess program. It is well known that the design of a concurrent program is not an easy task. To that end, base synchronization objects have been defined to help the programmer solve concurrency and process cooperation issues. A major step in that direction has been (more than forty years ago!) the concept of *mutual exclusion* [4] that has given rise to the notion of a *lock* object. Such an object provides the processes with two operations (lock and unlock) that allows a single process at a time to access a concurrent object. Hence, from a concurrent object point of view, the lock associated with an object allows transforming concurrent accesses to that object into sequential accesses. Interestingly, all the books on synchronization and operating systems have chapters on lock-based synchronization. According to the abstraction level supplied to the programmer, a lock may be encapsulated into a linguistic construct such as a *monitor* [21] or a *serializer* [20].

Unfortunately locks have drawbacks. One is related to the granularity of the object protected by a lock. More precisely, if several data items are encapsulated in a single concurrent object, the inherent parallelism the object can provide can be drastically reduced. This is for example the case of a queue object for which concurrent executions of enqueue and dequeue operations should be possible as long as they are not on the same item. Of course a solution could consist in considering each item of the queue as a concurrent object, but in that case, the operations enqueue and dequeue can become very difficult to design and implement. More severe drawbacks associated with locks lie in the fact that lock-based operations are deadlock-prone and cannot be easily composed.

Hence the question: how to ease the job of the programmer of concurrent applications? A (partial) solution consists in providing her/him with an appropriate library where (s)he can find correct and efficient implementations of the most popular concurrent data structures (e.g., [18, 24]). Albeit very attractive, this approach does not solve entirely the problem as it does not allow the programmer to define specific concurrent objects that take into account her/his particular synchronization issues.

The Software Transactional Memory approach The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory has first been proposed (nearly twenty years ago!) by Herlihy and Moss to implement concurrent data structures [17]. It has then been implemented in software by Shavit and Touitou [28] and has recently gained great momentum as a promising alternative to locks in concurrent programming [8, 13, 22, 25]. Interestingly enough, it is important to also observe that the recent advent of multicore architectures has given rise to what is called the *multicore revolution* [15] that has rang the revival of concurrent programming.

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

More generally, STM is a middleware approach that provides the programmers with the *transaction* concept (this concept is close but different from the notion of transactions encountered in databases [8]). A process is designed as (or decomposed into) a sequence of transactions, with each transaction being a piece of code that, while accessing concurrent objects, always appears as being executed atomically[1]. The job of the programmer is only to state which units of computation have to be atomic. He does not have to worry about the fact that the objects accessed by a transaction can be concurrently accessed. Except when (s)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.

Let us observe that the "spirit/design philosophy" that has given rise to STM systems is not new: it is related to the notion of *abstraction level*. More precisely, the aim is here to allow the programmer to focus and concentrate only on the problem (s)he has to solve and not on the base machinery needed to solve it. As we can see, this is the approach that has replaced assembly languages by high level languages and programmer-defined garbage collection by automatic garbage collection. STM can be seen as a new concept that takes up this challenge by addressing synchronization issues.

The state of affairs and (a few) related work (Due to page limitation, see also appendix A) Of course, a solution in which a single transaction at a time is executed trivially implements transaction atomicity but is inefficient on a multiprocessor system (as it allows only non-transactional code to execute in parallel). So, an STM system has to allow several transactions to execute concurrently. But, as soon as several transactions can execute concurrently, it is possible that they access the same concurrent objects in a conflicting manner. In that case, some of these transactions might have to be aborted. Hence, in a classical STM system, there is an *abort/commit* notion associated with transactions.

Several approaches have been proposed to cope with aborted transactions. In some systems, the management of aborted transactions is left to the application programmer (similarly to exception handling encountered in some systems). Other systems offer "best effort semantics" (which means that there is no provable strong guarantee). An interesting example of this approach is the technique called

[1] Actually, while the word "*transaction*" has historical roots, it seems that "*atomic procedure*" would be more appropriate because "transactions" of STM systems are computer science objects that are different from database transactions. We nevertheless continue using the word "*transaction*" for historical reasons.

steal-on-abort [1]. Its base principle is the following one. If a transaction $T1$ is aborted due to a conflict with a transaction $T2$, $T1$ is assigned to the processor that executed $T2$ in order to prevent a new conflict between $T1$ and $T2$.

Another approach consists in designing schedulers that perform particularly well in appropriate workloads. As an example, the case of read-dominated workloads is deeply investigated in [2]. Yet another approach consists in enriching the system with a contention manager, the aim of which is to improve performance and ensure (best effort or provable) progress guarantees. An associated theory is described in [10]. Failure detector-based contention managers (and corresponding lower bounds) are described in [11]. A construction to execute parallel programs made up of *atomic blocks* that have to be dispatched to queues accessed by threads (logical processors) is presented in [29].

Aim of the paper The previous observations show that, contrarily to the initial hope, STM systems proposed so far do not entirely free the programmer from the management of synchronization-related issues or they require specific adaptation of both the compiler and the underlying scheduler. (Actually, the management of aborted transactions can be seen as the transactional counterpart of the deadlock management techniques - prevention/resolution - encountered in lock-based systems.) Ideally, an STM system should be such that, from the programmer point of view, each transaction invoked by a process is executed exactly once. Said differently, this means that, at the programming level, there should be no notion of abort/commit associated with a transaction. In that way, the programmer would really be concerned neither by the way synchronization is implemented, nor by the way transactions are executed[2].

This paper is a step in that direction. From a conceptual point of view, it advocates that the abstraction level offered to the programmer and the technicalities needed at the implementation level have to be totally dissociated[3]: the programmer needs not to be aware of the way the underlying STM system works (i.e., similarly to the fact that deadlock management and parameter passing mechanisms associated with procedure calls are hidden to the programmer, abort/commit is an implementation-related notion that has to remain confined to the transaction implementation level).

Content of the paper The paper addresses the previous issue by presenting a construction (STM system) such that, at the programming level, every transaction invocation is executed exactly once. The notion of abort/commit is relegated to the implementation and not known to the programmer. As already indicated, the job of a programmer is to write her/his concurrent program in terms of cooperating sequential processes, each process being made up of a sequence of transactions (plus possibly some non-transactional code). At the programming level, any transaction invoked by a process is executed exactly once (similarly to a procedure invocation in sequential computing). Moreover, from a global point of view, any execution of the concurrent program is linearizable [19], meaning that all the transactions appear as if they have been executed one after the other in an order compatible with their real-time occurrence order. Hence, from the programmer point of view, the progress condition associated with an execution is a very classical one, namely, *starvation-freedom*.

The proposed construction can be seen as a *universal construction* for transaction-based concurrent programs. In addition to providing such a construction, an important aim of the paper is to investigate the design principles this construction relies on (efficiency issues are not part of our study). Interestingly, the fact that there is no abort/commit notion at the programming level and the very existence of this construction constitute an interesting feature which feeds the debate on the "liveness" of STM systems. Moreover, the proposed construction has a noteworthy feature: it is lock-free (in the sense it uses no lock).

In order to build such a construction, the paper assumes an underlying multiprocessor where the processors communicate through a shared memory that provides them with atomic read/write registers, compare&swap registers and fetch&increment registers.

As we will see, the underlying multiprocessor system consists of m processors and each processor is in charge of a subset of the n processes of the multiprocess program defined by the programmer. We say that a processor *owns* the corresponding processes in the sense that it has the responsibility of their individual progress. Given that, at the implementation level, a transaction may abort, the processor P_x owning the corresponding process p_i can require the help of the other processors in order that transaction be eventually committed. The implementation of this helping mechanism is at the core of the construction (similarly to the helping mechanism used to implement wait-free operations despite any number of process crashes [14]). As we will see, the main technical difficulties lie in ensuring that (1) the helping mechanism allows a transaction to be committed exactly once and (2) each processor P_x ensures the individual progress of each process p_i that it owns. As we can see, from a global point of view, the m processors have to cooperate in order to ensure a correct execution/simulation of the n processes.

Roadmap The paper is made up of 5 sections. Section 2 presents the model offered to the programmer and the underlying multiprocessor model on which the STM system is built. Section 3 presents the universal construction (STM system) for transaction-based concurrent programs. Section 4 proves its correctness. Finally, Section 5 concludes the paper by discussing additional features of the proposed construction.

[2]It is nevertheless important to notice that while a transaction can access local variables and concurrent objects, we assume here that it does not issue inputs/outputs. Those are done in the non-transactional code of the corresponding process. Hence a transaction is somewhat restricted in what it can do.

[3]It is important to notice that such an approach is adopted in some systems where the abort/commit notion is unknown to the programmer and the pair "compiler + scheduler" is designed in such a way that every transaction is "practically always" eventually executed [6].

2 Computation models

This section presents the programming model offered to the programmers and the underlying multiprocessor model on top of which the universal STM system is built.

2.1 The user programming model

The program written by the user is made up of n sequential processes denoted p_1, ..., p_n. Each process is a sequence of transactions in which two consecutive transactions can be separated by non-transactional code. Both transactions and non-transactional code can access concurrent objects.

Transactions A transaction is an atomic unit of computation (atomic procedure) that can access concurrent objects called t-objects. "Atomic" means that (from the programmer's point of view) a transaction appears as being executed instantaneously at a single point of the time line (between its start event and its end event) and no two transactions are executed at the same point of the time line. It is assumed that, when executed alone, a transaction always terminates.

Non-transactional code Non-transactional code is made up of statements for which the user does not require them to appear as being executed as a single atomic computation unit. This code usually contains input/output statements (if any). Non-transactional code can also access concurrent objects. These objects are called nt-objects.

Concurrent objects Concurrent objects shared by processes (user level) are denoted with small capital letters. It is assumed that a concurrent object is either an nt-object or a t-object (not both). Moreover, each concurrent object is assumed to be linearizable.

The atomicity property associated with a transaction guarantees that all its accesses to t-objects appear as being executed atomically. As each concurrent object is linearizable (i.e., atomic), the atomicity power of a transaction is useless if the transaction accesses once a single t-object. Hence encapsulate accesses to concurrent objects in a single transaction is "meaningful" only if that transaction accesses several objects or accesses the same object several times (as in a Read/Modify/Write operation).

As an example let us consider a concurrent queue (there are very efficient implementation of such an object, e.g., [24]). If the queue is always accessed independently of the other concurrent objects, its accesses can be part of non-transactional code and this queue instance is then an nt-object. Differently, if the queue is used with other objects (for example, when moving an item from a queue to another queue) the corresponding accesses have to be encapsulated in a transaction and the corresponding queue instances are then t-objects.

Semantics As already indicated the properties offered to the user are (1) linearizability (safety) and (2) the fact that each transaction invocation entails exactly one execution of that transaction (liveness).

2.2 The underlying system model

The underlying system is made up of m processors (simulators) denoted P_1, ..., P_m. We assume $n \geq m$. The processors communicate through shared memory that consists of single-writer/multi-reader (1WMR) atomic registers, compare&swap registers and fetch&increment registers.

Notation The objects shared by the processors are denoted with capital italic letters. The local variables of a processor are denoted with small italic letters.

Compare&swap register A compare&swap register X is an atomic object that provides processors with a single operation denoted $X.\mathsf{Compare\&Swap}()$. This operation is a conditional write that returns a boolean value. Its behavior can be described by the following statement:

operation $X.\mathsf{Compare\&Swap}(old, new)$:
if $X = old$ **then** $X \leftarrow new$; $\mathsf{return}(true)$ **else** $\mathsf{return}(false)$ **end if**.

Fetch&increment register A fetch&increment register X is an atomic object that provides processors with a single operation, denoted $X.\mathsf{Fetch\&Increment}()$, that adds 1 to X and returns its new value.

3 A universal construction for STM systems

This section describes the proposed universal construction. It first introduces the control variables shared by the m processors and then describes the construction. As already indicated, its design is based on simple principles: (1) each processor is assigned a subset of processes for which it is in charge of their individual progress; (2) when a processor does not succeed in executing and committing a transaction issued by a process it owns, it requires help from the other processors; (3) the state of the t-objects accessed by transactions is represented by a list that is shared by the processors (similarly to [14]).

Without loss of generality, the proposed construction considers that the concurrent objects shared by transactions (t-objects) are atomic read/write objects. Extending to more sophisticated linearizable concurrent objects is possible. We limit our presentation to atomic read/write objects to keep it simpler.

3.1 Control variables shared by the processors

Each processor has local variables. Those will be described when presenting the construction. This section presents the shared variables used by the processors to execute the multiprocess program.

Pointer notation Some variables manipulated by processors are pointers. The following notation is associated with pointers. Let PT be a pointer variable. $\downarrow PT$ denotes the object pointed to by PT. let OB be an object. $\uparrow OB$ denotes a pointer to OB. Hence, $\uparrow (\downarrow PT) = PT$ and $\downarrow (\uparrow OB) = OB$.

Process ownership Each processor P_x is assigned a set of processes for which it has the responsibility of ensuring individual progress. A process p_i is assigned to a single processor. We assume here a static assignment. (It is possible to consider a dynamic process assignment. This would require an appropriate underlying scheduler. We do not consider such a possibility here in order to keep the presentation simple.)

The process assignment is defined by an array $OWNED\_BY[1..m]$ such that $OWNED\_BY[x]$ contains the identities of the processes "owned" by processor P_x. As we will see below the owner P_x of process p_i can ask other processors to help it execute the last transaction issued by p_i.

Representing the state of the t-objects As previously indicated, at the processor (simulation) level, the state of t-objects program is represented by a list of descriptors such that each descriptor is associated with a transaction that has been committed.

$FIRST$ is a compare&swap register containing a pointer to the first descriptor of the list. Initially $FIRST$ points to a list containing a single descriptor associated with a fictitious transaction that gives an initial value to each t-object. Let $DESCR$ be the descriptor of a (committed) transaction T. It has the following four fields.

- $DESCR.next$ and $DESCR.prev$ are pointers to the next and previous items of the list.
- $DESCR.tid$ is the identity of T. It is a pair $\langle i, t\_sn\rangle$ where i is the identity of the process that issued the transaction and t_sn is its sequence number (among all transactions issued by p_i).
- $DESCR.ws$ is a set of pairs $\langle$X,$v\rangle$ stating that T has written v into the concurrent object X.
- $DESCR.local\_state$ is the local state of the process p_i just before the execution of the transaction that follows T in its code.

Helping mechanism: the array $LAST\_CMT[1..m, 1..n]$ This array is such that $LAST\_CMT[x, i]$ contains the sequence number of p_i's last committed transaction as known by processor P_x. $LAST\_CMT[x, i]$ is written only by P_x. Is initial value is 0.

Helping mechanism: logical time $CLOCK$ is an atomic fetch&increment register initialized to 0. It is used by the helping mechanism to associate a logical date with a transaction that has to be helped. Dates define a total order on these transactions. They are used to ensure that any helped transaction is eventually committed.

Helping mechanism: the array $STATE[1..n]$ This array is such that $STATE[i]$ describes the current state of the execution (simulation) of process p_i. It has four fields.

- $STATE[i].tr\_sn$ is the sequence number of the next transaction to be issued by p_i.
- $STATE[i].local\_state$ contains the local state of p_i immediately before the execution of its next transaction (whose sequence number is currently kept in $STATE[i].tr\_sn$).
- $STATE[i].help\_date$ is an integer (date) initialized to $+\infty$. The processor P_x (owner of process p_i) sets $STATE[i].help\_date$ to the next value of $CLOCK$ when it requires help from the other processors in order the last transaction issued by p_i be eventually committed.
- $STATE[i].last\_ptr$ contains a pointer to a descriptor of the transaction list (its initial value is $FIRST$). $STATE[i].last\_ptr = pt$ means that, if the transaction identified by $\langle i, STATE[i].tr\_sn\rangle$ belongs to the list of committed transactions, it appears after the transaction pointed to by pt.

3.2 How the t-objects and nt-objects are represented

Let us remember that the t-objects and nt-objects are the objects accessed by the processes of the application program. The nt-objects are directly implemented in the memory shared by the processors and consequently their operations access directly that memory.

Differently, the values of the t-objects are kept in the ws field of the descriptors associated with committed transactions (these descriptors define the list pointed to by $FIRST$). More precisely, we have the following.

- A write of a value v into a t-object X by a transaction appears as the pair $\langle \mathrm{X}, v\rangle$ contained in the field ws of the descriptor that is added to the list when the corresponding transaction is committed.
- A read of a t-object X by a transaction is implemented by scanning downwards (from the end towards $FIRST$) the descriptor list until encountering the first pair $\langle \mathrm{X}, v\rangle$, the value v being then returned by the read operation. It is easy to see that the values read by a transaction are always mutually consistent (if the values v and v' are returned by the reads of X and Y issued by the same transaction, then the first value read was not overwritten when the second one was read).

3.3 Behavior of a processor: initialization

Initially a processor P_x executes the code of each process p_i it owns until its first transaction and then initializes accordingly the atomic register $STATE[i]$. Then P_x invokes $\mathsf{select}(OWNED\_BY[x])$ that returns it the identity of a process it owns and assigns it to its local variable my_next_proc. It also initializes local variables whose role will be explained later. This is described in Figure 1.

The function $\mathsf{select}(set)$ is *fair* in the following sense: if it is invoked infinitely often with $i \in set$, then i is returned infinitely often (this can be easily implemented). Moreover, $\mathsf{select}(\emptyset) = \bot$.

for each $i \in OWNED\_BY[x]$ **do**
  execute p_i until the beginning of its first transaction;
  $STATE[i] \leftarrow \langle 1, p_i$'s current local state, $+\infty, FIRST\rangle$
end for;
$my\_next\_proc \leftarrow \mathsf{select}(OWNED\_BY[x])$;
$k1\_counter \leftarrow 0$; my_last_cmt is a pointer initialized to $FIRST$.

Figure 1: Initialization for processor P_x ($1 \leq x \leq m$)

3.4 Behavior of a processor: main body

The behavior of a processor P_x is described in Figure 2. This behavior consists of a while loop that terminates when all transactions issued by the processes owned by P_x have been successfully executed. This behavior can be decomposed into 4 parts.

Select the next transaction to execute (Lines 01-11) Processor P_x first reads (asynchronously) the current progress of each process and selects accordingly a process (lines 01-02). The procedure $\mathsf{select\_next\_process}()$ (whose details will be explained later) returns it the identity i of the process for which P_x has to execute the next transaction. This process p_i can be a process owned by P_x or a process whose owner P_y requires the other processors to help it execute its next transaction.

Next, P_x initializes local variables in order to execute p_i's next transaction in the appropriate correct context (lines 03-05). Before entering a speculative execution of the transaction, P_x first looks to see if it has not yet been committed (lines 07-11). To that end, P_x scans the list of committed transactions. Thanks to the pointer value kept in $STATE[i].last\_ptr$, it is useless to scan the list from the beginning: instead the scan may start from the transaction descriptor pointed to by $current = state[i].last\_ptr$. If P_x discovers that the transaction has been previously committed it sets the boolean $committed$ to $true$.

It is possible that, while the transaction is not committed, P_x loops forever in the list because the predicate $(\downarrow current).next = \bot$ is never true. This happens when new committed transactions (different from P_x's transactions) are repeatedly and infinitely added to the list. The procedure $\mathsf{prevent\_endless\_looping}()$ (line 07) is used to prevent such an infinite looping. Its details will be explained later.

Speculative execution of the selected transaction (Lines 12-17) The identity of the transaction selected by P_x is $\langle i, i\_tr\_sn\rangle$. If, from P_x's point of view, this transaction is not committed, P_x simulates locally its execution (lines 13-17). The set of concurrent t-objects read by p_i is saved in P_x's local set lrs, and the pairs $\langle \mathrm{Y}, v\rangle$ such that the transaction issued $\mathrm{Y}.\mathsf{write}(v)$ are saved in the local set ws. This is a transaction's speculative execution by P_x.

Try to commit the transaction (Lines 18-32) If P_x has performed a speculative execution of p_i's last transaction, it tries to commit it by adding it to the descriptor list if certain conditions are satisfied. To that end, P_x enters a loop (lines 19-24). There are two reasons not to try to commit the transaction.

- The first is when the transaction has already been committed. If this is the case, the transaction appears in the list of committed transactions (scanned by the pointer $current$, lines 21-22).

```
(01) state[1..n] ← [STATE[1], ··· , STATE[n]];
(02) i ← select_next_process();
(03) i_local_state ← state[i].local_state; i_tr_sn ← state[i].tr_sn;
(04) current ← state[i].last_ptr; committed ← false;
(05) k2_counter ← 0; after_my_last_cmt ← false;
(06) while ( ((↓ current).next ≠ ⊥) ∧ (¬committed) ) do
(07)       prevent_endless_looping(i);
(08)       if ((↓ current).tid = ⟨i, i_tr_sn⟩)
(09)         then committed ← true; i_local_state ← (↓ current).local_state end if;
(10)       current ← (↓ current).next
(11) end while;
(12) if (¬committed) then
       % — Simulation phase ————————————————————————————
(13)   execute the i_tr_sn-th transaction of p_i: the value of X.read() is obtained
(14)   by scanning downwards the transaction list (starting from current);
(15)   p_i's local variables are read from (written into) P_x's local memory (namely, i_local_state);
(16)   The set of shared objects read by the current transaction are saved in the set lrs;
(17)   The pairs ⟨Y,v⟩ such that the transaction issued Y.write(v) are saved in the set ws;
       % — Try to commit phase ——————————————————————————
(18)   overwritten ← false;
(19)   while ( (↓ current).next ≠ ⊥) ∧ (¬committed) ) do
(20)         prevent_endless_looping(i);
(21)         current ← (↓ current).next;
(22)         same as lines 08 and 09;
(23)         if (∃ X ∈ lrs : ⟨X, −⟩ ∈ (↓ current).ws) then overwritten ← true end if
(24)   end while;
(25)   if (¬committed ∧ (¬overwritten ∨ ws = ∅))
(26)     then allocate a new transaction descriptor DESCR;
(27)          DESCR ← ⟨⊥, current, ⟨i, i_tr_sn⟩, ws, i_local_state⟩;
(28)          committed ← Compare&Swap((↓ current).next, ⊥, ↑ DESCR);
(29)          if (¬committed) then disallocate DESCR end if
(30)   end if
(31) end if;
(32) if (committed) then LAST_CMT[x, i] ← i_tr_sn end if;
       % — End of transaction ———————————————————————————
(33) if (i ∈ OWNED_BY[x]) then
(34)   if (¬committed)
(35)     then if (state[i].help_date = +∞) then
(36)            helpdate ← Fetch&Incr(CLOCK);
(37)            STATE[i] ← ⟨state[i].tr_sn, state[i].local_state, helpdate, state[i].last_ptr⟩
(38)          end if
(39)     else execute non-transactional code of p_i (if any) in the local context i_local_state;
(40)          if (end of p_i's code)
(41)            then OWNED_BY[x] ← OWNED_BY[x] \ {i}
(42)            else STATE[i] ← ⟨i_tr_sn + 1, i_local_state, +∞, current⟩
(43)          end if;
(44)          my_last_cmt ← ↑ DESCR; my_next_proc ← select(OWNED_BY[x])
(45)   end if
(46) end if
end while.
```

Figure 2: Algorithm for processor P_x ($1 \leq x \leq m$)

- The second is when the transaction is not an update transaction and it has read a t-object that has then been overwritten (by a committed transaction). This is captured by the predicate at line 23.

Then, if (a) the transaction has not yet been committed (as far as P_x knows) and (b1) no t-object read has been overwritten or (b2) the transaction is read-only, then P_x tries to commit its speculative execution of this transaction (line 25). To that end, it first creates a new descriptor $DESCR$, updates its fields with the data obtained from its speculative execution (line 27) and then tries to add it to the list. To perform the commit, P_x issues Compare&Swap$((\downarrow current).next, \bot, \uparrow DESCR)$. It is easy to see that this invocation succeeds if and only if $current$ points to the last descriptor of the list of committed transactions (line 28).

Finally, P_x updates $LAST\_CMT[x, i]$ if the transaction has been committed (line 32).

Use the ownership notion to ensure the progress of each process (Lines 33-46) The last part of the description of P_x's behavior concerns the case where P_x is the owner of the process p_i that issued the current transaction selected by P_x (determined at line 03). This means that P_x is responsible for guaranteeing the individual progress of p_i. There are two cases.

- If $committed$ is equal to $false$, P_x requires help from the other processors in order p_i's transaction be eventually committed. To that end, it assigns (if not yet done) the next date value to $STATE[i].help\_date$ (lines 35-38). Then, P_x proceeds to the next loop iteration. (Let us observe that, in that case, my_next_proc is not modified.)
- If $committed$ is equal to $true$, as P_x is responsible of p_i's progress, its executes the non-transactional code (if any) that appears after the transaction (line 39). Then, if p_i has terminated, i is suppressed from $OWNED\_BY[x]$ (line 41). Otherwise, P_x updates $STATE[i]$ in order it contains the information required to execute the next transaction of p_i (line 42). Finally, before re-entering the main loop, P_x updates the pointer my_last_cmt (see below) and my_next_proc in order to ensure the progress of the next process it owns (line 44).

3.5 Behavior of a processor: starvation prevention

Any transaction issued by a process has to be eventually executed by a processor and committed. To that end, the helping mechanism introduced previously has to be enriched in order no processor (a) neither permanently helps only processes owned by other processors (b) nor loops forever in an internal while loop (lines 06-11 or 19-24). The first issue is solved by procedure select_next_process() while the second issue is solved by the procedure prevent_endless_looping().

Each of these procedures uses an integer value (resp., $K1$ and $K2$) as a threshold on the length of execution periods. These periods are measured with counters (resp., $k1\_counter$ and $k2\_counter$). When one of these periods attains its threshold, the corresponding processor requires help for its pending transaction. The values $K1$ and $K2$ can be arbitrary. They could be tuned or even defined dynamically in order to make the construction more efficient.

The procedure select_next_process() This operation is described in Figure 3. It is invoked at line 02 of the main loop and returns a process identity. Its aim is to allow the invoking processor P_x to eventually make progress for each of the processes it owns.

The problem that can occur is that a processor P_x can permanently help other processors execute and commit transactions of the processes they own, while none of the processes owned by P_x is making progress. To prevent this bad scenario from occurring, a processor P_x that does not succeed in having its current transaction executed and committed during a "too long" period, requires help from the other processors.

```
procedure select_next_process() returns (process id) =
(101)  let set = { i | (state[i].help_date ≠ +∞) ∧
                       ( (∀y : LAST_CMT[y,i] < state[i].tr_sn) ∨ (i ∈ OWNED_BY[x]) )};
(102)  if (set = ∅)
(103)    then i ← my_next_proc; k1_counter ← 0
(104)    else i ← min(set) computed with respect to transaction help dates;
(105)         if (i ∈ OWNED_BY[x])
(106)           then k1_counter ← 0
(107)           else k1_counter ← k1_counter + 1;
(108)                if (k1_counter ≥ K1) then
(109)                  let j = my_next_proc;
(110)                  if (state[j].help_date = +∞) then
(111)                    helpdate ← Fetch&Incr(CLOCK);
(112)                    STATE[j] ← ⟨state[j].tr_sn, state[j].local_state, helpdate, state[j].last_ptr⟩
(113)                  end if;
(114)                  k1_counter ← 0
(115)                end if
(116)         end if
(117)  end if;
(118)  return(i).
```

Figure 3: The procedure select_next_process()

This is realized as follows. P_x first computes the set set of processes p_i for which help has been required (those are the processes whose help date is $\neq +\infty$) and (as witnessed by the array $LAST\_CMT$) no processor has yet publicized the fact that their last transactions have been committed or p_i is owned by P_x (line 101). If set is empty (no help is required), select_next_process() returns the identity of the next process owned by P_x (line 103). If $set \neq \emptyset$, there are processes to help and P_x selects the identity i of the process with the oldest help date (line 104). But before returning the identity i (line 118), P_x checks if it has been waiting for a too long period before having its next transaction executed. There are then two cases.

- If $i \in OWNED\_BY[x]$, P_x has already required help for the process p_i for which it strives to make progress. It then resets the counter $k1\_counter$ to 0 and returns the identity i (line 106).
- If $i \notin OWNED\_BY[x]$, P_x first increases $k1\_counter$ (line 107) and checks if it attains its threshold $K1$. If this is the case, the logical period of time is too long (line 109) and consequently (if not yet done) P_x requires help for the last transaction of

the process p_j (such that $my\_next\_proc = j$). As we have seen, '"require help" is done by assigning the next clock value to $STATE[j].help\_date$ (lines 109-113). In that case, P_x also resets $k1\_counter$ to 0 (line 114).

```
procedure prevent_endless_looping(i);
(201)  if (i ∈ OWNED_BY[x]) then
(202)    if (current has bypassed my_last_cmt) then k2_counter ← k2_counter + 1 end if;
(203)    if ((k2_counter > K2) ∧ (state[i].help_date = +∞))
(204)      then helpdate ← Fetch&Incr(CLOCK);
(205)           STATE[i] ← ⟨state[i].tr_sn, state[i].local_state, helpdate, state[i].last_ptr⟩
(206)    end if
(207)  end if.
```

Figure 4: Procedure prevent_endless_looping()

The procedure prevent_endless_looping() As indicated, the aim of this procedure, described in Figure 4, is to prevent a processor P_x from endless looping in an internal while loop (lines 05-09 or 17-21).

The time period considered starts at the last committed transaction issued by a process owned by P_x. It is measured by the number of transactions committed since then. The beginning of this time period is determined by P_x's local pointer my_last_cmt (which is initialized to $FIRST$ and updated at line 44 of the main loop after the last transaction of a process owned by P_x has been committed.)

The relevant time period is measured by processor P_x with its local variable $k2\_counter$. If the process p_i currently selected by select_next_process() is owned by P_x (line 201), P_x requires help for p_i when this period attains $K2$ (lines 203-206). In that way, the transaction issued by that process will be executed and committed by other processors and (if not yet done) this will allow P_x to exit the while loop because its local boolean variable $committed$ will then become true (line 09 of the main loop).

4 Proof of the STM construction

Let $PROG$ be a transaction-based n-process concurrent program. The proof of the universal construction consists in showing that a simulation of $PROG$ by m processors that execute the algorithms described in Figures 1-4 generates an execution of $PROG$. Due to page limitation, the proof is given in Appendix B.

5 A short discussion to conclude

The aim of the universal construction that has been presented was to demonstrate and investigate this type of construction for transaction-based multiprocess programs. (Efficiency issues would deserve a separate investigation.) To conclude, we list here a few additional noteworthy properties of the proposed construction.

- The construction is for the family of transaction-based concurrent programs that are time-free (i.e., the semantics of which does not depend on real-time constraints).
- The construction is lock-free and works whatever the concurrency pattern (i.e., it does not require concurrency-related assumption such as obstruction-freedom). It works for both finite and infinite computations and does not require specific scheduling assumptions. Moreover, it is independent of the fact that processes are transaction-free (they then share only nt-objects), do not have non-transactional code (they then share only t-objects accessed by transactions) or have both transactions and non-transactional code.
- The helping mechanism can be improved by allowing a processor to require help for a transaction only when some condition is satisfied. These conditions could be general or application-dependent. They could be static or dynamic and be defined in relation with an underlying scheduler or a contention manager. The construction can also be adapted to benefit from an underlying scheduling allowing the owner of a process to be dynamically defined.

  It could also be adapted to take into account *irrevocable* transactions [27, 31]. Irrevocability is an implementation property which can be demanded by the user for some of its transactions. It states that the corresponding transaction cannot be aborted (this can be useful when one wants to include inputs/outputs inside a transaction; notice that, in our model, inputs/outputs appear in non-transactional code).
- We have considered a failure-free system. It is easy to see that, in a crash-prone system, the crash of a processor entails only the crash of the processes it owns. The processes owned by the processors that do not crash are not prevented from executing.

In addition to the previous properties, the proposed construction helps better understand the atomicity feature offered by STM systems to users in order to cope with concurrency issues. Interestingly this construction has some "similarities" with general constructions proposed to cope with the net effect of asynchrony, concurrency and failures, such as the BG simulation [3] (where there are simulators that execute processes) and Herlihy's universal construction to build wait-free objects [14] (where an underlying list of consensus objects used to represent the state of the constructed object lies at the core of the construction). The study of these similarities would deserve a deeper investigation.

References

[1] Ansar M., Luján M., Kotselidis Ch., Jarvis K., Kirkham Ch. and Watson Y., Steal-on-abort: Dynamic Transaction Reordering to Reduce Conflicts in Transactional Memory. *4th Int'l ACM Sigplan Conference on High Performance Embedded Architectures and Compilers (HiPEAC'09)*, ACM Press, pp. 4-18, 2009.

[2] Attiya H. and Milani A., Transactional Scheduling for Read-Dominated Workloads. *13th Int'l Conference on Principles of Distributed Systems (OPODIS'09)*, Springer Verlag LNCS #5923, pp. 3-17, 2009.

[3] Borowsky E. and Gafni E., Generalized FLP Impossibility Results for t-Resilient Asynchronous Computations. *Proc. 25th ACM Symposium on Theory of Computing (STOC'93)*, ACM Press, pp. 91-100, 1993.

[4] Dijkstra E.W.D., Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, 8(9):69, 1968.

[5] Dragojević A., Guerraoui R. and Kapałka M., Stretching Transactional Memory. *Proc. Int'l 2009 ACM SIGPLAN conference on Programming language design and implementation (PLDI '09)*, ACM Press, pp. 155-165, 2009.

[6] Felber P., Compiler Support for STM Systems. Lecture given at the *TRANSFORM Initial Training School*, University of Rennes 1 (France), 7-11 February 2011.

[7] Felber P., Fetzer Ch. and Riegel T., Dynamic Performance Tuning of Word-Based Software Transactional Memory. *Proc. 13th Int'l ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)* , ACM Press,pp. 237-246, 2008.

[8] 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.

[9] Guerraoui R., Henzinger Th. A. and Singh V., Permissiveness in Transactional Memories. *Proc. 22nd Int'l Symposium on Distributed Computing (DISC '08)* , Springer-Verlag, LNCS#5218, pp. 305-319, 2008.

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

[11] Guerraoui R., Kapałka M. and Kouznetsov P., The Weakest Failure Detectors to Boost Obstruction-freedom. *Distributed Computing*, 20(6):415-433, 2008.

[12] Guerraoui R. and Kapałka M., Principles of Transactional Memory. *Synthesis Lectures on Distributed Computing Theory*, Morgan & Claypool Publishers, 180 pages, 2010.

[13] 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.

[14] Herlihy M.P., Wait-Free Synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124-149, 1991.

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

[16] Herlihy M., Luchangco V., Moir M. and Scherer III W.M., Software Transactional Memory for Dynamic-Sized Data Structures. *Proc. 22nd Int'l ACM Symposium on Principles of Distributed Computing (PODC'03)*, ACM Press, pp. 92-101, 2003.

[17] 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 Architecture (ISCA'93)*, ACM Press, pp. 289-300, 1993.

[18] Herlihy M.P. and Shavit N., The Art of Multiprocessor Programming. *Morgan Kaufmann Pub.*, San Francisco (CA), 508 pages, 2008.

[19] 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.

[20] Hewitt C.E. and Atkinson R.R., Specification and Proof Techniques for Serializers. *IEEE Transactions on Software Engineering*, SE5(1):1-21, 1979.

[21] Hoare C.A.R., Monitors: an Operating System Structuring Concept. *Communications of the ACM*, 17(10):549-557, 1974.

[22] Larus J. and Kozyrakis Ch., Transactional Memory: Is TM the Answer for Improving Parallel Programming? *Communications of the ACM*, 51(7):80-89, 2008.

[23] Maldonado W., Marlier P., Felber P., Lawall J., Muller G. and Revière E., Deadline-Aware Scheduling for Software Transactional Memory. *41th IEEE/IFIP Int'l Conference on Dependable Systems and Networks 'DSN'11)*, IEEE CPS Press, June 2011.

[24] Michael M.M. and Scott M.L., Simple, Fast and Practical Blocking and Non-Blocking Concurrent Queue Algorithms. *Proc. 15th Int'l ACM Symposium on Principles of Distributed Computing (PODC'96)*, ACM Press, pp. 267-275, 1996.

[25] Raynal M., Synchronization is Coming Back, But Is It the Same? Keynote Speech. *IEEE 22nd Int'l Conference on Advanced Information Networking and Applications (AINA'08)*, pp. 1-10, 2008.

[26] Rosenkrantz D.J., Stearns R.E. and Lewis Ph.M., System Level Concurrency Control for Distributed Database Systems. *ACM Transactions on Database Systems*, 3(2): 178-198, 1978.

[27] Spear M.F., Silverman M., Dalessandro L., Michael M.M. and Scott M.L., Implementing and Exploiting Inevitability in Software Transactional Memory. *Proc. 37th Int'l Conference on Parallel Processing (ICPP'08)*, IEEE Press, 2008.

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

[29] Wamhoff J.-T. and Fetzer Ch., The Universal Transactional Memory Construction. *Tech Report*, 12 pages, University of Dresden (Germany), 2010.

[30] Wamhoff J.-T., Riegel T., Fetzer Ch. and Felber F., RobuSTM: A Robust Software Transactional Memory. *Proc. 12th Int'l Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'10)*, Springer Verlag LNCS #6366, pp. 388-404, 2010.

[31] Welc A., Saha B. and Adl-Tabatabai A.-R., Irrevocable Transactions and their Applications. *Proc. 20th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'08)*, ACM Press, pp. 285-296, 2008.

A Related work

The paper has presented a new STM construction that ensures that every transaction issued by a process is necessarily committed and each process makes progress. To our knowledge, this is the first STM system we know of to combine these concepts, but much work has been done previously on these two "liveness"-related concepts when implemented separately. This appendix gives a short overview of that work. The presentation is not entirely fair as the major part of the papers that are cited here are efficiency-oriented while our construction is more theory-oriented.

Contention Management The idea is here to keep track of conflicts between transactions and have a separate entity, usually called *contention manager*, decide what action to take (if any). Some of these actions include aborting one or both of the conflicting transactions, stalling one of the transactions, or doing nothing. The idea was first (as far as we know) proposed in the dynamic STM system (called DSTM) [16] and much research has been done on the topic since then.

An overview of different contention managers and their performance is presented in [10]. Interestingly the authors find that there is no "best" contention manager and that the performance depends on the application. The notion of a *greedy contention manager* they propose ensures that every issued transaction eventually commits. This is done by giving each transaction a time-stamp when it is first issued and, once the time-stamp reaches a certain age, the system ensures that no other transaction can commit that will cause this transaction to abort. Similarly to the "Wait/Die" or "Wound/Wait" strategies used to solve deadlocks in some database systems [26], preventing transactions from committing is achieved by either aborting them or directing them to wait. By doing this it is obvious that processes with conflicting transactions make progress. Their progress depends on the process that issued the transaction with the oldest time-stamp to commit.

As discussed in this paper, committing every issued transaction should be ensured by an STM. Unfortunately, in order to ensure these properties, the greedy contention manager is considered by many to be too expensive to always use and still provide good performance. To get around this some modern STM's (e.g., for example TinySTM [7] or SwissSTM [5]) use less expensive contention management until a transaction has been aborted a certain number of times at which point greedy contention management is used.

Transactional Scheduling As we have seen, in order to help a transaction commit, the proposed construction allows a transaction to be executed and committed by a processor different the one it originated from. The idea to execute a transaction on a different processor has already been proposed in [1] and [2] where this idea is used to provide processes with a type of contention management refereed to as *transactional scheduling*. A very short description of these papers has been given in the introduction. Like contention managers, these schedulers can provide progress, but none of them ensure the progress of a process with a transaction that conflicts with some other transaction which has reached a point at which it must not be aborted. This means that the progress of a process still depends on the progress of another process.

Obstruction-Freedom, Lock-Freedom Many STM systems have been proposed that do not use locks. Some of them are obstruction-free (e.g., [16, 28]) or lock-free (in the sense there is no deadlock) [9]. Unfortunately, none of them provides the property that every issued transaction is committed. At most, the transactions that are helped in these papers are the ones in the process of committing. It can easily be seen that, with only this type of help, some transaction can be aborted indefinitely.

In an obstruction-free STM system a transaction that is executed alone must eventually commit. So consider some transaction that is always stalled (before its commit operation) and, while it is stalled, some conflicting transaction commits. It is easy to build an execution in which this stalled transaction never commits.

In a lock-free STM system, infinitely many transaction invocations must commit in an infinite execution. Again it is possible to build an execution in which a transaction is always stalled (before its commit operation) and is aborted by a concurrent transaction (transactions cannot wait for this stalled transaction because they do not know if it is making progress).

An interesting discussion on the possible/impossible liveness properties of a STM system is presented in [12] where is also described a general lock-free STM system. This system is based on a mechanism similar to the Compare&Swap used in this paper.

Irrevocable Transactions The aim of the concept of *irrevocable* (or *inevitable*) transaction is to provide the programmer with a special transaction type (or tag) related to its liveness or progress. Ensuring that a transaction does not abort is usually required for transactions that perform some operations that cannot be rolled back or aborted such as I/O. In order to solve this issue, certain STM systems provide irrevocable transactions which will never be aborted once they are typed irrevocable. This is done by preventing concurrent conflicting transactions from committing when an irrevocable transaction is being executed.

It is interesting to note that (a) an irrevocable transaction must be run exactly once and (b) only one irrevocable transaction can be executed at a time in the system (unless the shared memory accesses of the transaction are known ahead of time). Possible solutions are proposed and discussed in [27] and [31]. Irrevocable transactions suited to deadline-aware scheduling are presented in [23].

The STM system proposed in this paper does not support irrevocable transactions. Since an irrevocable transaction cannot be executed more than once, it is obvious that irrevocable transactions cannot benefit from the helping mechanism proposed in the paper. Our STM construction could nevertheless be extended to provide this functionality by having a process that wants to execute an irrevocable transaction issue a Compare&Swap on an irrevocability flag to the end of the list in order to prevent any other concurrent transaction from committing until it commits itself. Unfortunately, this would violate the liveness and progress properties previously guaranteed. That is why in our model, "irrevocable transactions" appear as non-transactional code.

Robust STMs Ensuring progress even when bad behavior (such as process crash) can occur has been investigated in several papers. As an example, [30] presents a robust STM system where a transaction that is not committed for a too long period eventually gets priority using locks. It is assumed that the system provides a crash detection mechanism that allows locks to be stolen once a crash is detected. This paper also presents a technique to deal with non-terminating transactions.

B Proof of the construction

Lemma 1 *Let T be the transaction invocation with the smallest help date (among all the transaction invocations not yet committed for which help has been required). Let p_i be the process that issued T and P_y a processor. If T is never committed, there is a time after which P_y issues an infinite number invocations of* select_next_process() *and they all return i.*

Proof Let us assume by contradiction that there is a time after which either P_y is blocked within an internal while loop (Figure 2) or its invocations of select_next_process() never return i. It follows from line 104 of select_next_process() that the process identity of the transaction from set with the smallest help date is returned. This means that for i to never be returned, there must always be some transaction(s) in set with a smaller help date than T. By definition we know that T is the uncommitted transaction with the smallest help date, so any transaction(s) in set with a smaller help date must be already committed. Let us call this subset of committed transactions T_{set}. Since set is finite, T_{set} also is finite. Moreover, T_{set} cannot grow because any transaction T' added to the array $STATE[1..n]$ has a larger help date than T (such a transaction T' has asked for help after T and due to the Fetcch&Increment() operation the help dates are monotonically increasing). So to complete the contradiction we need to show that (a) P_y is never blocked forever in an internal while loop (Figure 2) and (b) eventually $T_{set} = \emptyset$.

If T_{set} is not empty, select_next_process() returns the process identity j for some committed transaction $T' \in T_{set}$. On line 09, the processor P_y will see T' in the list and perform $committed \leftarrow true$. Hence, P_y cannot block forever in an internal while loop. Then, on line 32, P_y updates $LAST\_CMT[y, j]$. Let us observe that, during the next iteration of select_next_process() by P_x, T' is not be added to set (line 101) and, consequently, there is then one less transaction in T_{set}. And this continues until T_{set} is empty. After this occurs, each time processor P_y invokes select_next_process(), it obtains the process identity i, which invalidates the contradiction assumption and proves the lemma. $\square_{Lemma\ 1}$

Lemma 2 *Any invocation of a transaction T that requests help (hence it has $helpdate \neq \infty$) is eventually committed.*

Proof Let us first observe that all transactions that require help have bounded and different help dates (lines 36-37, 111-112 or 205-206). Moreover, once defined, the helping date for a transaction is not modified.

Among all the transactions that have not been committed and require help, let T be the transaction with the smallest help date. Assume that T has been issued by process p_i owned by processor P_x (hence, P_x has required help for T). Let us assume that T is never committed. The proof is by contradiction.

As T has the smallest help date, it follows from Lemma 1 that there is a time after which all the processors that call select_next_process() obtains the process identity i. Let $\mathcal{P}$ be this non-empty set of processors. (The other processors are looping in a while loop or are slow.) Consequently, given that all transactions that are not slow are trying to commit T (by performing a compare&swap() to add it to the list), that the list is not modified anywhere else, and that we assume that T never commits, there is a finite time after which the descriptor list does no longer increase. Hence, as the predicate $(\downarrow current).next = \perp$ becomes eventually true, we conclude that at least one processor $P_y \in \mathcal{P}$ cannot be blocked forever in a while loop. Because the list is no longer changing, the predicate of line 25 then becomes satisfied at P_y. It follows that, when the processors of $\mathcal{P}$ execute line 28, eventually one of them successfully executes the compare&swap that commits the transaction T which contradicts the initial assumption.

As the helping dates are monotonically increasing, it follows that any transaction T that requires help is eventually committed. $\square_{Lemma\ 2}$

Lemma 3 *No processor P_x loops forever in an internal while loop (lines 06-11 or 19-24).*

Proof The proof is by contradiction. Let P_y be a processor that loops forever in an internal while loop. Let i be the process identity it has obtained from its last call to select_next_process() (line 02) and P_x be the processor owner of p_i.

Let us first show that processor P_x cannot loop forever in an internal while loop. Let us assume the contrary. As P_x loops forever we never have $((\downarrow current).next = \bot) \vee committed$, but each time it executes the loop body, P_x invokes prevent_endless_looping(i) (at line 07 or 20). The code of this procedure is described in Figure 4. As $i \in OWNED\_BY[i]$ and P_x invokes infinitely often prevent_endless_looping(i), it follows from lines 201-203 and the current value of my_last_cmt (that points to the last committed transaction issued by a process owned by P_x, see line 44) that P_x's local variable $k2\_counter$ is increased infinitely often. Hence, eventually this number of invocations attains $K2$. When this occurs, if not yet done, P_x requires help for the transaction issued by p_i (lines 203-206). It then follows from Lemma 2, that p_i's transaction T is eventually committed. As the pointer $current$ of P_x never skips a descriptor of the list and the list contains all and only committed transactions, we eventually have $(\downarrow current).tid = \langle i, i\_tr\_sn\rangle$ (where i_tr_sn is T's sequence number among the transactions issued by p_i). When this occurs, P_x's local variable $committed$ is set to $true$ and P_x stops looping in an internal while loop.

Let us now consider the case of a processor $P_y \neq P_x$. Let us first notice that the only way for P_y to execute T is when T has requested help (line 101 of select_next_process()). The proof follows from the fact that, due to Lemma 2, T is eventually committed. As previously (but now $current$ is P_y's local variable), the predicate $(\downarrow current).tid = \langle i, i\_tr\_sn\rangle$ eventually becomes true and processor P_y sets $committed$ to $true$. P_y then stops looping inside an internal while loop (line 08 or 22) which concludes the proof of the lemma. $\square_{Lemma\ 3}$

Lemma 4 *Any invocation of a transaction T by a process is eventually committed.*

Proof Considering a processor P_x, let $i \in OWNED\_BY[x]$ be the current value of its local control variable my_next_proc. Let T be the current transaction issued by p_i. We first show that T is eventually committed.

Let us first observe that, as p_i has issued T, P_x has executed line 42 where it has updated $STATE[i]$ that now refers to that transaction. If P_x requires help for T, the result follows from Lemma 2. Hence, to show that T is eventually committed, we show that, if P_x does not succeed in committing T without help, it necessarily requires help for it. This follows from the code of the procedure select_next_proc(). There are two cases.

- select_next_process() returns i. In that case, as P_x does not loop forever in a while loop (Lemma 3), it eventually executes lines 33-38 and consequently either commits T or requires help for T at line 37.
- select_next_process() never returns i. In that case, as P_x never loops forever in a while loop (Lemma 3), it follows that it repeatedly invokes select_next_process() and, as these invocations do not return i, the counter $k1\_counter$ repeatedly increases and eventually attains the value $K1$. When this occurs P_x requires help for T (lines 107-115) and, due to Lemma 2, T is eventually committed.

Let us now observe that that, after T has been committed (by some processor), P_x executes lines 39-44 where it proceeds to the simulation of its next process (as defined by select($OWNED\_BY[x]$)). It then follows from the previous reasoning that the next transaction of the process that is selected (whose identity is kept in my_next_proc) is eventually committed.

Finally, as the function select() is fair, it follows that no process is missed forever and, consequently, any transaction invocation issued by a process is eventually committed. $\square_{Lemma\ 4}$

Lemma 5 *Any invocation of a transaction T by a process is committed at most once.*

Proof Let T be a transaction committed by a processor P_y (i.e., the corresponding Compare&Swap() at line 28 is successful). T is identified $\langle i, STATE[i].ts\_sn\rangle$. As P_y commits T, we conclude that P_y has previously executed lines 06-28.

- We conclude from the last update of $STATE[i].last\_ptr = pt$ by P_y (line 42) and the fact that P_y's $current$ local variable is initialized to $STATE[i].last\_ptr$, that T is not in the descriptor list before the transaction pointed to by pt.
- Let us consider the other part of the list. As T is committed by P_y, its pointer $current$ progresses from $STATE[i].last\_ptr = pt$ until its last value that is such that $(\downarrow current).next = \bot$. It then follows from lines 08 and 22 that P_y has never encountered a transaction identified $\langle i, STATE[i].ts\_sn\rangle$ (i.e., T) while traversing the descriptor list.

It follows from the two previous observations that, when it is committed (added to the list), transaction T was not already in the list, which concludes the proof of the lemma. $\square_{Lemma\ 5}$

Lemma 6 *Each invocation of a transaction T by a process is committed exactly once.*

Proof The proof follows directly from Lemma 4 and Lemma 5. $\square_{Lemma\ 6}$

Lemma 7 *Each invocation of non-transactional code issued by a process is executed exactly once.*

Proof This lemma follows directly from lines 39-44: once the non-transactional code separating two transaction invocations has been executed, the processor P_x that owns the corresponding process p_i makes it progress to the beginning of its next transaction (if any). $\square_{Lemma\ 7}$

Lemma 8 *The simulation is starvation-free (no process is blocked forever by the processors).*

Proof This follows directly from Lemma 3, Lemma 6, Lemma 7 and the definition of the function select(). $\square_{Lemma\ 8}$

Lemma 9 *The transaction invocations issued by the processes are linearizable.*

Proof To prove the lemma we have (a) to associate a linearization point with each transaction invocation and (b) show that the corresponding sequence of linearization points is consistent, i.e., the values read from t-objects by a transaction invocation T are uptodate (there have not been overwritten in that sequence). As far as item (a) is concerned, the linearization point of a transaction invocation is defined as follows[4].

- Update transactions (these are the transactions that write at least one t-object). The linearization point of the invocation of an update transaction is the time instant of the (successful) compare&swap statement that entails its commit.
- Read-only transactions. Let W be the set of update transactions that have written a value that has been read by the considered read-only transaction. Let τ_1 be the time just after the maximum linearization point of the invocations of the transactions in W and τ_2 be the time at which the first execution of the considered transaction has started. The linearization point of the transaction is then $\max(\tau_1, \tau_2)$.

To prove item (b) let us consider the order in which the transaction invocations are added to the descriptor list (pointed to by $FIRST$). As we are about to see, this list and the linearization order are not necessarily the same for read-only transaction invocations. Let us observe that, due to the atomicity of the compare&swap statement, a single transaction invocation at a time is added to the list.

Initially, the list contains a single fictitious transaction that gives an initial value to every t-object. Let us assume that the linearization order of all the transaction invocations that have been committed so far (hence they define the descriptor list) is consistent (let us observe that this is initially true). Let us consider the next transaction T that is committed (i.e., added to the list). As previously, we consider two cases. Let p_i be the process that issued T, P_x the processor that owns p_i and P_y the processor that commits T.

- The transaction is an update transaction (hence, $ws \neq \emptyset$). In that case, P_y has found $(\downarrow current).next = \perp$ (because the compare&sap succeeds) and at line 25, just before committing, the predicate $\neg committed \wedge \neg overwritten$ is satisfied.

  As $overwritten$ is false, it follows that none of the values read by T has been overwritten. Hence, the reads and writes on t-objects issued by T can appear as having been executed atomically at the time of the compare&swap. Moreover, the values of the t-objects modified by T are saved in the descriptor attached to the list by the compare&swap and the global state of the t-objects is consistent (i.e., if not overwritten before, any future read of any of these t-objects obtains the value written by T).

  Let us now consider the local state of p_i (the process that issued T). There are two cases.

  - $P_x = P_y$ (the transaction is committed by the owner of p_i). In that case, the local state of p_i after the execution of T is kept in P_x's local variable i_local_state (line 15). After P_x has executed the non-transactional code that follows the invocation of T (if any, line 39), it updates $STATE[i].local\_state$ with the current value of i_local_state (if p_i had not yet terminated, line 42).
  - $P_x \neq P_y$ (the processor that commits T and its owner are different processors). In that case, P_y has saved the new local state of p_i in $DESCR.local\_state$ (line 27) just before appending $DESCR$ at the end of the descriptor list.

    Next, thanks to the the predicate $i \in OWNED\_BY[x]$ in the definition of set at line 101, there is an invocation of select_next_process() by P_x that returns i. When this occurs, P_x discovers at line 09 or 22 that the transaction T has been committed by another processor. It then retrieves the local state of p_i (after execution of T) in $(\downarrow current).local\_state$, saves it in i_local_state and (as in the previous item) eventually writes it in $STATE[i].local\_state$ (line 42).

  It follows that, in both cases, the value saved in $STATE[i].local\_state$ is the local state of p_i after the execution of T and the non-transactional code that follows T (if any).

[4]The fact that a transaction invocation is *read-only* or *update* cannot always be statically determined. It can depend on the code of transaction (this occurs for example when a transaction behavior depends on a predicate on values read from t-objects). In our case, a read-only transaction is a transaction with an empty write set (which cannot be always statically determined by a compiler).

- The transaction is a read-only transaction (hence, $ws = \emptyset$). In that case, T has not modified the state of the t-objects. Hence, we only have to prove that the new local state of p_i is appropriately updated and saved in $STATE[i].local\_state$.

  The proof is the same as for the case of an update transaction. The only difference lies in the fact that now it is possible to have $overwritten \wedge ws = 0$. If $overwritten$ is true, T can no longer be linearized at the commit point. That is why it is linearization point has been defined just after the maximum linearization point of the transactions it reads from (or the start of T if it happens later), which makes it linearizable.

$\square_{Lemma\ 9}$

Lemma 10 *The simulation of a transaction-based n-process program by m processors (executing the algorithms described in Figures 1-4) is linearizable.*

Proof Let us first observe that, due to Lemma 9, The transaction invocations issued by the processes are linearizable, from which we conclude that the set of t-objects (considered as a single concurrent object TO) is linearizable. Moreover, by definition, every nt-object is linearizable.

As (a)linearizability is a *local* consistency property [19][5] and (b) TO is linearizable and every nt-object is linearizable, it follows that the execution of the multiprocess program is linearizable. $\square_{Lemma\ 10}$

Theorem 1 *Let $PROG$ be a transaction-based n-process program. Any simulation of $PROG$ by m processors executing the algorithms described in Figures 1-4 is an execution of $PROG$.*

Proof A formal statement of this proof requires an heavy formalism. Hence we only give a sketch of it. Basically, the proof follows from Lemma 8 and Lemma 10. The execution of $PROG$ is obtained by projecting the execution of each processor on the simulation of the transactions it commits and the execution of the non-transactional code of each process it owns. $\square_{Theorem\ 1}$

C The number of tries is bounded

This section presents a bound for the maximum number of times a transaction can be unsuccessfully executed by a processor before being committed, namely, $O(m^2)$. A workload that has this bound is then given.

Lemma 11 *At any time and for any processor P_x, there is at most one atomic register $STATE[i]$ with $i \in OWNED\_BY[x]$ such that the corresponding transaction (the identity of which is $\langle i, STATE[i].tr\_sn\rangle$) is not committed and $STATE[i].help\_date \neq +\infty$.*

Proof Let us first notice that the help date of a transaction invoked by a process p_i can be set to a finite value only by the processor P_x that owns p_i. There are two places where P_x can request help.

- This first location is in the prevent_endless_looping() procedure. In that case, the transaction for which help is required is the last transaction invoked by process $p_{my\_next\_proc}$.
- The second location is on line 37 after the transaction invocation T aborts. It follows from line 103 of select_next_process() that this invocation is also from the last transaction invoked by process $p_{my\_next\_proc}$.

So we only need to show that my_next_proc only changes when a transaction is committed, which follows directly from the predicates at lines 34 and 35 and the statements of line 44. $\square_{Lemma\ 11}$

Theorem 2 *A transaction T invoked by a process p_i owned by processor P_x is tried unsuccessfully at most $O(m^2)$ times before being committed.*

Proof Let us first observe that a transaction T (invoked by a process p_i) is executed once before its help date is set to a finite value (if it is not committed after that execution). This is because only the owner P_x of p_i can select T (line 103) when its help date is $+\infty$. Then, after it has executed T unsuccessfully once, P_x requests help for T by setting its help date to a finite value (line 37).

Let us now compute how many times T can be executed unsuccessfully (i.e., without being committed) after its help date has been set to a finite value. As there are m processors and all are equal (as far as helping is concerned), some processor must execute T more than $O(m)$ times in order for T to be executed more than $O(m^2)$ times. We show that this is impossible. More precisely, assuming a processor P executes T, there are 3 cases that can cause this execution to be unsuccessful and as shown below each case can cause at most $O(m)$ aborts of T at P.

[5] A property P is local if the set of concurrent objects (considered as a single object) satisfies P whenever each object taken alone satisfies P. It is proved in [19] that linearizability is a local property.

- Case 1. The first case is that some other transaction $T1$ that does not request help (its help date is $+\infty$) is committed by some other processor $P2$ causing P's execution of T to abort. Now by lines 102 and 103 after $P2$ commits $T1$, $P2$ will only be executing uncommitted transactions from the $STATE$ array with finite help dates at least until T is committed, so any subsequent abort of T caused by $P2$ cannot be caused by $P2$ committing a transaction with $+\infty$ help date. So the maximum number of times this type of abort can happen from P is $O(1)$.
- Case 2. The second case is when some other uncommitted transaction $T1$ in the $STATE$ array with a finite help date is committed by some other processor $P2$ causing T to abort. First by lemma 11 we know that there is a maximum of $m-1$ transactions that are not T that can be requesting help at this time and in order for them to commit before T they must have a help date smaller than T's. Also by lemma 5 we know that a transaction is committed exactly once so this conflict between $T1$ and T cannot occur again at $P2$. Now after committing $T1$, the next transaction (that asks for help) of a process that is owned by the same processor that owned $T1$ will have a larger help date than T so now there are only $m-1$ transactions that need help that could conflict with T. Repeating this we have at most $O(m)$ conflicts of this type for P.
- Case 3. The third case is that P's execution of T is aborted because some other process has already committed T. Then on line 08 P will see that T has been committed and not execute it again, so we have at most $O(1)$ conflicts of this type. $\square_{Theorem\ 2}$

The bound is tight The execution that is described below shows that a transaction T can be tried $O(m^2)$ times before being committed.

Let T be a transaction owned by processor $P(1)$ such that $P(1)$ executes T unsuccessfully once and requires help by setting its help date to a finite value. Now, let us assume that each of the $m-1$ other processors is executing a transaction it owns, all these transactions conflict with T and there are no other uncommitted transactions with their help date set to a finite value.

Now $P(1)$ starts executing T again, but meanwhile processor $P(2)$ commits its own transaction which causes T to abort. Next $P(1)$ and $P(2)$ each try to execute T, but meanwhile processor $P(3)$ commits its own transaction causing $P(1)$ and $P(2)$ to abort T. Next $P(1)$, $P(2)$, and $P(3)$ each try execute T, but meanwhile processor $P(4)$ commits its owns transaction causing $P(1)$, $P(2)$, and $P(3)$ to abort T. Etc. until processor $P(m-1)$ aborts all the execution of T by other processors, resulting in all m processor executing T. The transaction T is then necessarily committed by one of these final executions. So we have $1+1+2+3+\ldots+(m-1)+m$ trials of T which is $O(m^2)$.