Nº d'ordre : ANNÉE 2012

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informátique

Ecole doctorale Matisse

présentée par

Tyler Crain

préparée à l'unité de recherche (n^o + nom abrégé)
(Nom développé de l'unité)
(Composante universitaire)

Intitulé de la thèse: Software Transactional Memory and Concurrent Data Structures: On the performance and usability of parallell programming abstractions

Thèse soutenue à l'INRIA – Rennes le ?? Mars 2013

devant le jury composé de :

Software Transactional Memory and Concurrent Data Structures: On the performance and usability of parallel programming abstractions

Abstract

Writing parallel programs is well know difficult task and because of this there are some abstractions that can help make parallel programming easier. Two different approaches that try to solve this problem are transactional memory and concurrent data abstractions (such as sets or dictionaries). Transactional memory can be viewed as a methodology for parallel programming, allowing the programmer the ability to declare what sections of his code should be executed atomically, while concurrent data abstractions expose some specifically defined operations that can be safely executed concurrently to access and modify shared data. Importantly, even though these are separate solutions to the problem they are not mutually exclusive. They can be (and are) used together, as an example concurrent data abstractions are often used within transactions. Unfortunately they both are know to suffer from performance and (especially in the case of transactional memory) ease of use problems. This thesis examines and proposes some solutions trying to address these problems.

Transactional memory is designed to make parallel programming easier by allowing the programmer to define what blocks of code he wants to be executed atomically while leaving the difficult synchronization tasks for the underlying system. Even though the idea of a transaction is clear, how the programmer should interact with the abstraction is still not clearly defined and STM systems are known to suffer from performance problems. Three STM algorithms are presented to help study these problems.

1. One that satisfies two properties that can be considered good for efficiency.
2. One that guarantees transactions commit exactly once and hides the concepts of aborts to the programmer.
3. One that allows safe concurrent access to the same memory both inside and outside of transactions.

Concurrent data structures representing abstractions such as a set or map/dictionary are often an important component of parallel programs as they allow the programmer to access/store/and modify data safely and concurrently. Due to this they can be heavily used and highly contended in concurrent workloads leading to poor performance in programs. Some proposed solutions to this suggest weakening the abstraction, but this can make using the data structures more complicated. In this thesis a methodology is proposed that improves performance without weakening the abstraction. This methodology can also be applied to data structures used within transactions.

Acknowledgments

Contents

List of Figures

Part I

Software Transactional Memory Algorithms

Chapter 1

Introduction

Multicore architectures are changing the way we write programs. Not only are all computational devices turning multicore thus becoming inherently concurrent, but tomorrow's multicore will embed a larger amount of simplified cores to better handle energy while proposing higher performance, a technology also known as *manycore* [83]. Thus in order to take advantage of these resources programs must be written so that they can execute concurrently.

Lock-based concurrent programming 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* [32] that has given rise to the notion of a *lock* object. Such an object provides the processes with two operations (lock and unlock) that allow 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 on that object into sequential accesses. Interestingly, all the books on synchronization and operating systems have chapters on lock-based synchronization. In addition, according to the abstraction level supplied to the programmer, a lock may be encapsulated into a linguistic construct such as a *monitor* [45] or a *serializer* [44]. The type of syncronization ammitted by locks is often refered to as*pessimistic*, as each access to some location x blocks further accesses to x until the location is released. Unsurprisingly given that this concept of mutual exclusion is a straighforward way to concieve of synchronization and concurrency, locking is the by far the most widely used abstraction to implement concurrent algorithms.

Unfortunately locks have several 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 the first solution that comes to mind would be to use more fine grained locks. For example one could consider each item of the queue as a concurrent object with its own lock, allowing the operations enqueue and dequeue operations to execute concurrently. Unfortunately it is not that simple as implementing operations using fine grained locking can become very difficult to design and implement correctly.

The most common difficulty associated with using fine grained locking is avoiding deadlock. Deadlock occurs when a process A wants to obtain a lock that is already owned by process B

while concurrently process *B* wants to obtain a lock that is already owned by process *A* resulting in neither thread progressing. In order to avoid deadlock locks are often accquired in a global order, but this may result in locks being taken more often and held longer then necessary.

Other problems with locks can occur when a process holding a lock is descheduled by the operating system while a live process is trying to access the samelock (sometimes called *priority inversion*. Further problems can occur if a thread crashes or is stalled while holding a lock.

Other drawbacks associated with locks lie in the fact that lock-based operation cannot be easily composed [69, 67].

Non-blocking concurrent programming Non-blocking algorithms [87] have been introduced as an alternative to using locks, these algorithms are implemented using system level synchronization operations such as *compare_and_swap* and do not suffer some of the scalibility or progress problems that locks do. There have been many efficient and scalible non-blocking data structures proposed [95, 103, 105, 88, 89]. Unfortunately such algorithms are known to be extreemly difficult to implement and undersatnd. Frequently top research conferences will publish papers that introduce a concurrent non-blocking version of a data structure showing how difficut these algorithms can be to implement.

Hence the question: how to ease the job of the programmer of concurrent applications? A (partial) solution consists of 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., [43, 48]). Albeit very attractive, this approach does not solve entirely the problem as it does not allow the programmer to define specific concurrent executions 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 was first proposed (nearly twenty years ago!) by Herlihy and Moss as an abstraction to be used in order to easily implement lock-free concurrent data structures [12]. It has since been implemented in software by Shavit and Touitou [23] and has recently gained great momentum as a promising alternative to locks in concurrent programming [7, **?**, 46, 49]. 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* [11] that has rang the revival of concurrent programming.

Transactional memory abstracts away the complexity associated with concurrent programming by replacing locking with atomic execution units. Unlike using locks where a programmer might use several locks throughout his operations, when using transactional memory a programmer just needs to define what sections of his code should appear as if they execute atomically (i.e. all at once, leaving no possibility for interleaved concurrent operations). The transactional memory protocol then deals with the necessary syncronization to ensure that this happens. A programmer could think of it as using a single global lock where ever he wants to perform synchronization between processes. In that way, the programmer has to focus on where atomicity is required and not on the way it must be realized. The aim of an STM system is consequently to discharge the programmer from the direct management of the synchronization that is entailed by accesses to concurrent objects.

More explicitly, 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 database systems [7, 10, 11]). 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 if it was 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. The programmer is not concerned by synchronization except when (s)he defines the beginning and the end of a transaction. It is then the job of the STM system to ensure that transactions are executed 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 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 when considering synchronization issues.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is irrelevant from an efficiency point of view. So, a STM system has to do "its best" to execute and commit as many transactions per time unit as possible (a concept sometimes refered to as *optimistic synchronization*), but unfortuantely, similarly to a scheduler, a STM system is an on-line algorithm that does not know the future. Therefore, if the STM is not trivial (i.e., it allows several transactions that access the same objects to run concurrently), then conflicts between concurrent transactions may require the system to abort some transactions in order to ensure both transaction atomicity and object consistency. Hence, in a classical STM system there is an *abort/commit* notion associated with transactions. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered a new transaction). Abortion is the price that has to be paid by transactional systems to cope with concurrency in absence of explicit pessimistic synchronization mechanisms (such as locks or event queues).

Simplicity As we can see the primary goal of transactional memory is to make concurrent programming easier and more accessable, with performance following closely as a secondary goal. Unfortunately neiter of these goals have yet been realized. While the basic semantics of a transaction are widely aggreed on (that each transaction should appear to have been executed atomically with respect to other transactions), there are many other details to consider, some of which are actively debated and some of which remain as open questions. Standardizing these semantics and answering these open questions is an important step in ensuring that the primary goal of making concurrent programming easier is realized. If the semantics are either too hard to understand, or a programmer has to study too many details of one or more STM systems before being able to use them in his program then the original goal has been lost. While there is still no agreement on the full semantics of transactional memory, there has still been much research that focuses on performance first. The reason for this is because even though STM in its current form has been shown to be efficient for certain worloads [], many people argue that its performance is not good enough to replace locks, even with all the difficulties that surround using locks.

[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.

This thesis takes the approach of finding the correct semantics for transactional memory before trying to improve performance as why try to improve the performace for something that is not fully defined and might change? As such the goal is to help define the semantics of software transactional memory that a programmer can use without being an expert in the area.

Unfortunately there are far too many questions left open when dealing with the semantics of transactional memory to introduce here or let alone solve. Instead of attempting such a lofty goal this thesis will indroduce what we belived are the main areas of transactional memory research that take the ease-of-use as a primary concern. Each chapter will introduce an area before looking closely at a specific open problem from that area and suggesting an STM protocol as a solution to the problem.

The first chapter focuses on the area of STM research which takes the view that the first-class ease of use requirement is satisfied by ensuring transactions are atomic with respect to eachother and all transactions (aborted transaction included) execute in a consistent state of memory. Then once this is satisfied we can focus on improving the implementing protocol by increasing its performance or having it satsify desireable properties. Following this model, this chapter looks at the properties of *invisible reads* and *permissiveness* showing that they are not compatble with the correctness condition of *opacity* before introduing a protocol that does satisfy these properties using the correctness condition of *virtual world consistency*. See the chapter for the definition of these properties.

Following the view of transactional memory of the first chapter, issues that are not solved by ensuring transactions are atomic with eachother are left up to the programmer. Meaning he has to understand these issues and to choose an appropriate STM protocl based on his needs. The second and the third chapter looks at research that believes these sematics are not appropriate for making parallel programming easy to use. The second chapter suggests looking at ways to simplify these semantics while the thrid chapter suggests expanding these semantics to consider more issuse that might come up for the programmer.

In the second chapter we look at research that takes the view that for transactional memory to be easy to use the semantics defined in the first chapter should be simplified, meaning that, in a sense, the abstraction level should be raised. Specifically it looks at the problem of aborts in transactional memory and suggests that the programmer should not have to be aware of them, following that suggestion it introduces a protocol that ensures every transaction issued by a process commits no matter the concurrecy pattern of other threads in the system.

They simplify concurrent programming for two reasons. First, the programmer only needs to delimit regions of sequential code into transactions or to replace critical sections by transactions to obtain a safe concurrent program. Second, the resulting transactional program is reusable by any programmer, hence a programmer composing operations from a transactional library into another transaction is guaranteed to obtain new deadlock-free operations that execute atomically. Studies have shown that transactional memory is easier to program with then locks [79, 80].

1.1 STM computation model and base definitions

1.1.1 Processes and atomic shared objects

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

Each of the m shared objects is an atomic read/write object. This means that the read and write operations issued on such an object X appear as if they have been executed sequentially, and this "witness sequence" is legal (a read returns the value written by the closest write that precedes it in this sequence) and respects the real time occurrence order on the operations on X (if $op1(X)$ terminates before $op2(X)$ starts, $op1$ appears before $op2$ in the witness sequence associated with X).

1.1.2 Transactions and object operations

In this section we define our model for transactional memory and its operations that will be used when describing algorithms in this thesis.

Transaction A transaction is a piece of code that is produced on-line by a sequential process (automaton), that is assumed to be executed atomically (commit) or not at all (abort). This means that (1) the transactions issued by a process are totally ordered, and (2) the designer of a transaction does not have to worry about the management of the base objects accessed by the transaction. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or to write any shared object.

The set of the objects read by a transaction defines its *read set*. Similarly the set of objects it writes defines its *write set*. A transaction that does not write shared objects is a *read-only* transaction, otherwise it is an *update* transaction. A transaction that issues only write operations is a *write-only* transaction.

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

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

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

Incremental snapshot As in [3], we assume that the behavior of a transaction T can be decomposed in three sequential steps: it first reads data objects, then does local computations and finally writes new values in some objects, which means that a transaction can be seen as a software read_modify_write() operation that is dynamically defined by a process. (This model is for reasoning, understand and state properties on STM systems. It only requires that everything appears as described in the model.)

The read set is defined incrementally, which means that a transaction reads the objects of its read set asynchronously one after the other (between two consecutive reads, the transaction can issue local computations that take arbitrary, but finite, durations). We say that the transaction T

computes an *incremental snapshot*. This snapshot has to be *consistent* which means that there is a time frame in which these values have co-existed (as we will see later, different consistency conditions consider different time frame notions).

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

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

Chapter 2

Read Invisibility, Virtual World Consistency and Permissiveness are Compatible

2.1 Motivation

2.1.1 Software transactional memory (STM) systems

The most imporant goal of the concept of STM is to make concurrent programming easier and more accessable to any programmer. Arguably all a programmer should need to know in order to use STM abstraction is to know the syntax for writing an atomic block, likely something as simple as

$$atomic\{\dots\}$$

All the complexities and difficult synchronization is then taken care of by the underlying STM system. This means that, when faced to synchronization, a programmer has to concentrate on where atomicity is required and not on the way it is realized. (Unfortunately, in reality, it is actually not so simple for the programmer, this is discussed in detail in section **??**) Like many abstractions, even if what is exposed to the programmer is a fixed well defined interface there are many different possible implementations and different properties that exist to provide the abstraction.

Let us consider the common example of a classical map abstraction provided by a data structure. A programmer might want to access and store some data in memory and he knows that in order to do this he needs the *insert*, *delete*, and *contains* operations provided by the map abstraction. He also knows he wants to perform these operations concurrently. With this information he should be able to take a take a library providing this functionality and use it in his program without knowing any additional information.

Now even though this is all that the programmer needs to know, there are many more details to consider at a lower level. The most obvious is that there are different data stuctures that can be used to provide a map abstraction, with examples such as a skip-list, binary tree, or hash table. Chosing one of these may impact the applications performance or provide additional functionality, yet to the programmer just interested in the map abstraction he case use each of them the same.

To further complicate things for the people implementing the actual datat sturcutures, even when considering a single data sturcture there are different properties a specific implementation might ensure. For example you might have a blocking skip-list or a non-blocking skip-list which provide different guarantees of the progress of the operations. Taking this a step further, different implementations might perform differently depending on the workload. For example you might have a skip-list implementation that is memory efficient, but is slower then another implementation that is less memory efficient. Again though a programmer is not required to know these details.

Each of these points are true in transactional memory as well. At the most basic level all a programmer should need to know is where to begin and end his atomic blocks. Beneth these atomic blocks lies the STM implementation which has many different aspects. Since the introduction of transactional memory in 1993 [] dozens of different properties have emerged as well as many different STM algorithms, with each of them ensuring a greater or fewer number of these properties and being more or less concerned with performance. (In the related work section (section **??**) we give an overview of many different properties and implementations of STM)

In an ideal world there would exist a *"perfect"* STM algorithm that ensures all desirable properties without making any sacrifies. Unfortunately this algorithm has not yet been discovered (if it is even possible). In fact many of these desierable properties have been little more then introduced and many of their implications on how they affect STM algorithms or how they interact with eachother has yet to be explored. Motivated by this, this chapter examines two desiralbe properties that are concerned with performance, namely *permissiveness* and *invisible-reads*, and how they interact with two consistency criterion for STM systems, namely *opacity* and *virtual world consistency*. As previously noted, these are just a few of the many properties that have been defined for STM systems so this work is only touching on a much larger set of problems, but hopefully this work encourages the study of attitional properties.

2.1.2 Some interesting and desirable properties for STM systems

Invisible read operation A read operation issued by a transaction is *invisible* if it does not entail the modification of base shared objects used to implement the STM system [18]. This is a desirable property for both efficiency and privacy.

Permissiveness The notion of permissiveness has been introduced in [8] (in some sense, it is a very nice generalization of the notion of *obligation* property [15]). It is on transaction abort. Intuitively, an STM system is *permissive* "if it never aborts a transaction unless necessary for correctness" (otherwise it is *non-permissive*). More precisely, an STM system is permissive with respect to a consistency condition (e.g., opacity) if it accepts every history that satisfies the condition.

As indicated in [8], an STM system that checks at commit time that the values of the objects read by a transaction have not been modified (and aborts the transaction if true) cannot be permissive with respect to opacity. In fact other then the protocol introduced along with permissiveness in [8] virtually all published STM protocols abort transactions that could otherwise be safely commmitted, i.e. the protcols are not permissive.

Probabilistic Permissiveness Some STM systems are randomized in the sense that the commit/abort point of a transaction depends on a random coin toss. Probabilistic permissiveness is suited to such systems. A randomized STM system is *probabilistically permissive* with respect to a consistency condition if every history that satisfies the condition is accepted with positive probability [8].

2.2 Opacity and virtual world consistency

2.2.1 Two consistency conditions for STM systems

If you have not noticed already, a reocurring theme throughout this document is that the most important goal of transactional memory is ease of use, and is the subject of this section. We alreay have our syntax defined for the programmer ($atomic\{\dots\}$)) and a basic idea of what this means, “the code in the atomic block will appear as if it has been executeded instantaneously”, yet we need to precisely define what this means for an STM algorithm. At the heart of this we have consistency criterions. These criterions precisely define the semantics of a transaction and guide the creation of algorithms in order that the chosen criterion is satisfied. Without a clear and precisely defined consistency criterion we loose the ease of use that is the original intention of STM.

In this section we give a overview of two well known consitency criterion defined for transactional memory.

The opacity consistency condition The classical consistency criterion for database transactions is serializability [20], roughly defined as follows: “A history is serializable if it is equivalent to one in which transactions appear to execute sequentially, i.e., without interleaving.” What is important to consider when thinking about transactional memory is that the serializability consistency criterion involves only the transactions that commit. Said differently, a transaction that aborts is not prevented from accessing an inconsistent state before aborting. It should be noted that serializability is sometimes strengthened in “strict serializability”. Strict serializability has the additional constranst that the equivalent sequentail history must follow a real time order so that each transaction is placed somewhere between its invocation and response time, as implemented when using the 2-phase locking mechanism. Strict serializability is often refered to as linearizability [] when considering the operations of an object instead of the system as a while.

In contrast to database transactions that are usually produced by SQL queries, in a STM system the code encapsulated in a transaction is not restricted to particular patterns. Consequently a transaction always has to operate on a consistent state (no matter if it is eventually commited or not). To be more explicit, let us consider the following example where a transaction contains the statement $x \leftarrow a/(b-c)$ (where a, b and c are integer data), and let us assume that $b-c$ is different from 0 in all consistent states (intuitively, a consistent state is a global state that, considering only the committed transactions, could have existed at some real time instant). If the values of b and c read by a transaction come from different states, it is possible that the transaction obtains values such as $b = c$ (and $b = c$ defines an inconsistent state). If this occurs, the transaction throws a divide by zero exception that has to be handled by the process that invoked the corresponding transaction. Even worse undesirable behaviors can be obtained when reading values from inconsistent states. This occurs for example when an inconsistent state provides a transaction with values that generate infinite loops. Such bad behaviors have to be prevented in

STM systems: whatever its fate (commit or abort) a transaction has to see always a consistent state of the data it accesses. The aborted transactions have to be harmless.

Informally suggested in [6], and formally introduced and investigated in [9], the *opacity* consistency condition requires that no transaction, at any time, reads values from an inconsistent global state where, considering only the committed transactions, a *consistent global state* is defined as the state of the shared memory at some real time instant. Let us associate with each aborted transaction T its execution prefix (called *read prefix*) that contains all its read operations until T aborts (if the abort is entailed by a read, this read is not included in the prefix). An execution of a set of transactions satisfies the *opacity* condition if (i) all committed transactions plus each aborted transaction reduced to its read prefix appear as if they have been executed sequentially and (ii) this sequence respects the transaction real-time occurrence order.

Virtual world consistency This consistency condition, introduced in [16], is weaker than opacity while keeping its spirit. It states that (1) no transaction (committed or aborted) reads values from an inconsistent global state, (2) the consistent global states read by the committed transactions are mutually consistent (in the sense that they can be totally ordered) but (3) while the global state read by each aborted transaction is consistent from its individual point of view, the global states read by any two aborted transactions are not required to be mutually consistent. Said differently, virtual world consistency requires that (1) all the committed transactions be serializable [20] (so they all have the same "witness sequential execution") or linearizable [13] (if we want this witness execution to also respect real time) and (2) each aborted transaction (reduced to a read prefix as explained previously) reads values that are consistent with respect to its *causal past* only. Informally the causal past of a transaction is some valid history as viewed by the transaction, but not necessarily the same history as the one seen by other transactions i.e. some transactions might be missing or ordered differently. Causal past is defined more formally in the next section.

As two aborted transactions can have different causal pasts, each can read from a global state that is consistent from its causal past point of view, but these two global states may be mutually inconsistent as aborted transactions have not necessarily the same causal past (hence the name *virtual world* consistency).

In addition to the fact that it can allow more transactions to commit than opacity, one of the most important points of virtual world consistency lies in the fact that, as opacity, it prevents bad phenomena (as described previously) from occurring without requiring all the transactions (committed or aborted) to agree on the very same witness execution. Let us assume that each transaction behaves correctly (e.g. it does not entail a division by 0, does not enter an infinite loop, etc.) when, executed alone, it reads values from a consistent global state. As, due to the virtual world consistency condition, no transaction (committed or aborted) reads from an inconsistent state, it cannot behave incorrectly despite concurrency, it can only be aborted. This consistency condition can benefit many STM applications as, from its local point of view, a transaction cannot differentiate it from opacity.

So what does this mean for the programmer who plans to use transactional memory to write his concurrent program? Possible performance implications aside, absolutely nothing, The programmer will see no difference between an STM protocol that is opaque versus one that is virtual world consistent. Given the first requirement of transactional memory is ease of use, this is extreely important, virtual world consistency would be much less interesting as an STM consistency condition if this were not true.

2.2.2 Formal Definitions

This section defines formally opacity [9] and virtual world consistency [16]. First, we define some properties of STM executions. Then, based on these definitions, opacity and virtual world consistency are defined.

2.2.3 Base definitions

Preliminary remark Some of the notions that follow can be seen as read/write counterparts of notions encountered in message-passing systems (e.g., partial order and happened before relation [17], consistent cut, causal past and observation [2, 24]).

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

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

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

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

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

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

Causal past of a transaction Given a partial order $\widehat{PO}$ defined on a set of transactions, the *causal past* of a transaction T, denoted $past(T)$, is the set including T and all the transactions T' such that $T' \rightarrow_{PO} T$.

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

Linear extension A linear extension $\widehat{S} = (S, \to_S)$ of a partial order $\widehat{PO} = (PO, \to_{PO})$ is a topological sort of this partial order, i.e.,

- $S = PO$ (same elements),
- $\to_S$ is a total order, and
- $(T1 \to_{PO} T2) \Rightarrow (T1 \to_S T2)$ (we say "$\to_S$ respects $\to_{PO}$").

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

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

If all transactions are legal, the linear extension $\widehat{S}$ is legal.

In the following, a legal linear extension of a partial order, that models an execution of a set of transactions, is sometimes called a *sequential witness* (or witness) of that execution.

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

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

2.2.4 Opacity and virtual world consistency

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

Définition 2.1 *A strong transaction history satisfies the opacity consistency condition if it has a real time-compliant legal linear extension.*

Examples of protocols implementing the opacity property, each with different additional features, can be found in [6, 14, 16, 22].

Définition 2.2 *A weak transaction history satisfies the virtual world consistency condition if (a) all its committed transactions have a legal linear extension and (b) the causal past of each aborted transaction has a legal linear extension.*

A protocol implementing virtual world consistency can be found in [16] where it is also shown that any opaque history is virtual world consistent. In contrast, a virtual world consistent history is not necessarily opaque.

To give a better intuition of the virtual world consistency condition, let us consider the execution depicted on Figure 2.1. There are two processes: p_1 has sequentially issued T_1^1, T_1^2, T_1' and T_1^3, while p_2 has issued T_2^1, T_2^2, T_2' and T_2^3. The transactions associated with a black dot

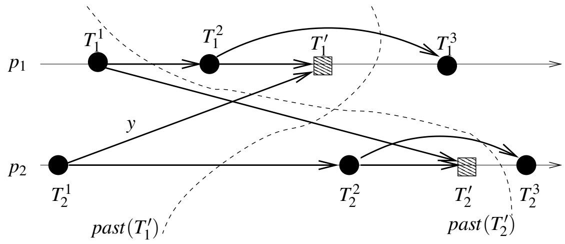

Figure 2.1: Examples of causal pasts

have committed, while the ones with a grey square have aborted. From a dependency point of view, each transaction issued by a process depends on its previous committed transactions and on committed transactions issued by the other process as defined by the read-from relation due to the accesses to the shared objects, (e.g., the label y on the dependency edge from T_2^1 to T_1' means that T_1' has read from y a value written by T_2^1). In contrast, since an aborted transaction does not write shared objects, there is no dependency edges originating from it. The causal past of the aborted transactions T_1' and T_2' are indicated on the figure (left of the corresponding dotted lines). The values read by T_1' (resp., T_2') are consistent with respect to its causal past dependencies.

2.3 Invisible reads, permissiveness, and consistency

In the previous section several interesting properties and consistency conditions for STM have been defined. In this section we examine the some of the implcations of these properties on STM algorithms.

We start by proving that an STM portocal cannot implemnt invisible reads, opactiy, and permissiveness at the same time. This is importat because it shows that no matter how well we keep track of the interatctions between transactions, an opaque STM portocol will have to abort transactions unnecessarily if invisible reads are used. We then show that by simply replacing virtual world consistency for opacity as the consistency condition no transactions need to be aborted unenessarily. Importantly changing virtual world consictency for opacity makes no difference to the meaning of a transaction for the programmer.

2.3.1 Invisible reads, opacity and permissiveness are incompatible

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

Proof Let us first consider permissiveness. The proof follows from a simple counter-example where three transactions T_1, T_2 and T_3 issue sequentially the following operations (depicted in Figure 2.2).

1. T_3 reads object X.
2. Then T_2 writes X and terminates. If the STM system is permissive it has to commit T_2. This is because if (a) the system would abort T_2 and (b) T_3 would be made up of only the read of X, aborting T_2 would make the system non-permissive. Let us notice that, at the time at which T_2 has to be committed or aborted, the future behavior of T_3 is not known and T_1 does not yet exist.

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

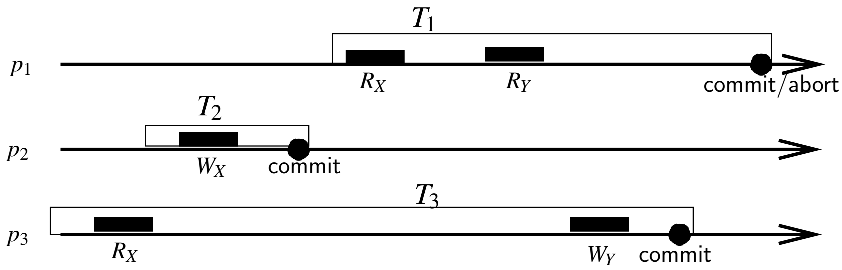

Figure 2.2: Invisible reads, opacity and permissiveness are incompatible

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

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

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

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

From this we have that read invisibility, permissiveness, and opacity are incompatible. $\square_{Theorem\ 1}$

Remark Let us observe that any opaque system with invisible reads would be required to abort T_3. When T_3 performs the try_to_commit() operation detecting that its read of X has been overwritten, it must abort (this is because T_3 has no way of knowing whether or not T_1's read exists at this point, so T_3 must abort in order to ensure safety). From this we have that read invisibility, permissiveness, and opacity are *incompatible* in the sense that any pair of properties can be satisfied only if the third is omitted.

2.3.2 Invisible reads, virtual world consistency and permissiveness are compatible

(Permissiveness is defined as follows: A transaction only aborts if by committing it violates consistency. Read operations are invisible, but committed read only transactions are visible. Other defninitions and notations are used from the paper *Read invisibility, virtual world consistency, and permissivenss are compatible*)

Theorem 2 *It is possible for a TM to implement VWC, invisible read operations and permissiveness.*

Proof

Consider some TM that has invisible read operations executing on some workload. We will show by induction that it is possible to have committed transactions that are permissive and VWC, and that aborted transactions are VWC.

The base case for the committed transactions is obvious (the first transaction to perform the *try_to_commit* operation in the shared memory history will commit). Now assume some transaction T_N is performing the *try_to_commit* operation after $n-1$ transactions have previously performed this operation in the shared memory history (either committing successfully or aborting). First notice that each operation of the shared memory history of all previously commited transactions are visible to T_N. Now assume by contridiction that this transaction commits and creates a history that is not VWC. This means that by commiting this transaction it becomes false that *all committed transaction have a realtime-complinat legal linear extension*. But since all operations of previous transactions are visible to this transacions then the transaction would have aborted if this was violated.

Now assume this transaction commits if this property is not violated. For a TM to be permissive the transaction must not abort unless by committing it violates consistency. From the above we know that the transactions are commiting as often as possible and that the committed transactions are VWC, but we do not know that the aborted tranactions are VWC.

Now consider the aborted transactions. We will prove by contradiction that the aborted transactions are VWC. For an aborted transaction T_A to not be VWC it must be false that its *causal past has a legal linear extension*. This means that some commited transaction $T_B \in past(T_A)$ and T_B must be illegal in the casual past of T_A ($past(T_A)$).

This crease the following posibilites (from the definition of a legal transaction):

1. The transaction T_C that wrote the value that was read by T_B is not in $past(T_A)$. But it is obvious that T_C must be in $past(T_A)$ because if T_B read the value from T_C then T_C must have committed before the read in the shared memory history, and will be in $past(T_A)$.

2. The transactions T_B, T_C, and T_D, are in $past(A)$. T_B read a value written by T_C. $T_C \to_S T_D \to_S T_B$ and $w_{T_D}(X) \in T_D$.

There are two cases to consider for posibility 2, first consider that $T_A \neq T_B$. So we have $T_C \rightarrow^{rf} T_B$ and $T_C \rightarrow_S T_D \rightarrow_S T_B$ and $w_{T_D}(X) \in T_D$, but this is impossible because each of these transactions are committed and are VWC (one of these transactions would have aborted when performing its *try_to_commit* operation).

Now consider that $T_A = T_B$. So we have $T_C \rightarrow^{rf} T_A$ and $w_{T_D}(X) \in T_D$ and $T_D \in past(T_A)$. There are three posibilties for T_D being in $past(T_A)$ (from the definition of casual past).

1. $T_D \rightarrow_{PO} T_A$. This is impossible because then T_A would never have read from T_C.

2. $T_D \rightarrow^{rf} T_A$. But since we also have $T_C \rightarrow^{rf} T_A$, then T_A would have aborted before it completed both of these reads.

3. $\exists(T_D \rightarrow_{PO} T) \wedge (T \rightarrow_{PO} T_A)$. First notice that in order for T_D to be in $past(T_A)$ it must commit in the shared memory history before the last read done by T_A. So assume T_D commits at a time in the shared memory history before $T_C \rightarrow^{rf} T_A$. Then it is possible that $T_D \in past(T_A)$ before $T_C \rightarrow^{rf} T_A$ in the shared memory history, but if this is true then T_A will abort when it tries to read a value from T_C. This means that T_D must be added to $past(T_A)$ after $T_C \rightarrow^{rf} T_A$ so the following must be true: $(T_D \rightarrow_{PO} T) \wedge (T \rightarrow^{rf} T_A)$ and $(T_C \rightarrow_{rf} T_A) <_H (T \rightarrow^{rf} T_A)$. But then T_A would abort when it performs the read from T.

$\square_{Theorem\ 2}$

!!!!!!!!!!!!!!SHOULD ALSO TALK ABOUT PROB PERM FOR THE PROOF!!!!!!!!!!!

Remark The previous section (section 2.3.2) shows that opacity is a too strong consistency condition when one wants both read invisibility and permissiveness while this section shows that virtual world consistency should be used instead.

Let us consider the execution in figure 2.2 that was used to show that opacity, invisible reads, and permissiveness are incompatible. Unsurprisingly this history can be made permissive and virtual world consistent even with read invisibility. In order for this to be true, a virtual world consistency protocol must then abort transaction T_1. Then it is easy to see that the corresponding weak transaction history is virtual world consistent: The read prefix "$X.\mathtt{read}_{T_1}()$; $Y.\mathtt{read}_{T_1}()$" of the aborted transaction T_1 can be ordered after T_2 (and T_3 does not appear in its causal past).

2.4 A protocol satisfying permissiveness and virtual world consistency with read invisibility

Even though the previous section shows that it is possible to have an STM protocol with read invisibility, permissiveness, and virtual world consistentency, it does not show that it is realistic to implement. In this section we introduce such a protocol that satisfies the above properties efficiently using realistic operations available in most hardware.

2.4.1 Step 1: Ensuring virtual world consistency with read invisibility

The protocol (named as IR_VWC_P) is built is two steps. This section presents the first step, namely, a protocol that ensures virtual consistency with invisible read operations. The second step (Section 2.4.4) will enrich this base protocol to obtain probabilistic permissiveness.

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

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

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

The proposed STM system is based on the incremental reads and deferred update strategy. Each transaction T uses a local working space. When T invokes $X.\mathsf{read}_T()$ for the first time, it reads the value of X from the shared memory and copies it into its local working space. Later $X.\mathsf{read}_T()$ invocations (if any) use this copy. So, if T reads X and then Y, these reads are done incrementally, and the state of the shared memory may have changed in between. As already explained, this is the *incremental snapshot* strategy.

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

2.4.1.2 The underlying data structures

Implementing a transaction-level shared object Each transaction-level shared object X is implemented by a list. Hence, at the implementation level, there is a shared array $PT[1..m]$ such that $PT[X]$ is a pointer to the list associated with X. This list is made up of cells. Let $CELL(X)$ be such a cell. It is made up of the following fields (see Figure 2.3).

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

  It is important to notice that none of these pointers are used in the protocol (Figure 2.4) that ensures virtual world consistency and read invisibility. $CELL(X).next$ is required only when one wants to recycle e inaccessible cells (see Section 2.4.6). Differently, $CELL(X).next$ will be used to obtain permissiveness (see Section 2.4.4).

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

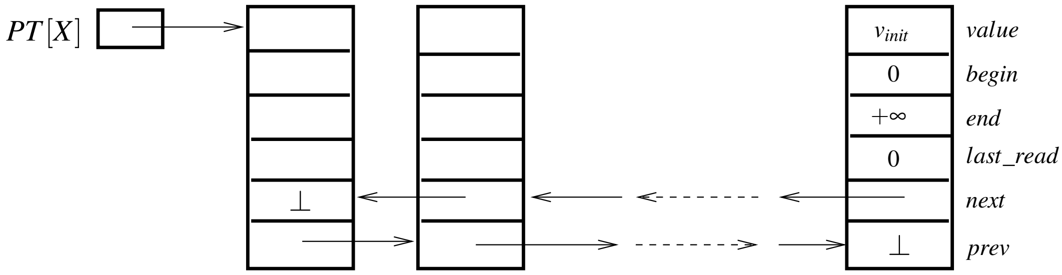

Figure 2.3: List implementing a transaction-level shared object X

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

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

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

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

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

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

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

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

Figure 2.4: Algorithm for the operations of the protocol

The $X.\mathsf{read}_T()$ **operation** The algorithm implementing $X.\mathsf{read}_T()$ is described in Figure 2.4. When p_i invokes this operation, it returns the value locally saved in $lcell(X).value$ if $lcell(X)$ exists (lines 02 and 14). If $lcell(X)$ has not yet been allocated, p_i does it (line 03) and updates its fields *value*, *begin* and *origin* with the corresponding values obtained from the shared memory (lines 04-07). Process p_i then updates $window\_bottom_T$ and $window\_top_T$. These updates are as follows.

- The algorithm defines the commit time of transaction T as a point of the time line such that T could have executed all its read and write operations instantaneously as that time. Hence, T cannot be committed before a committed transaction T' that wrote the value of a shared object X read by T. According to the algorithm implementing the $\text{try\_to\_commit}_T()$ operation (see line 28), the commit point of such a transaction T' is the time value kept in $lcell(X).begin$. Hence, p_i updates $window\_bottom_T$ to $\max(window\_bottom_T, lcell(X).begin)$ (line 08). X is then added to $lrst_T$ (line 09).

- Then, p_i updates $window\_top_T$ (the top side of T's commit window, line 11). If there is a shared object Y already read by T (i.e., $Y \in lrs_T$) that has been written by some other transaction T'' (where T'' is a transaction that wrote Y after T read Y), then $window\_top_T$ has to be set to $commit\_time_{T''}$ if $commit\_time_{T''} < window\_top_T$. According to the algorithm implementing the $\text{try\_to\_commit}_T()$ operation, the commit point of such a transaction T'' is the date kept in $(lcell(Y).origin \downarrow).end$. Hence, for each $Y \in lrs_T$, p_i updates $window\_bottom_T$ to $\min\big(window\_top_T, (lcell(Y).origin \downarrow).end$ (line 11).

Then, if the window becomes empty, the $X.\text{read}_T()$ operation entails the abort of transaction T (line 12). If T is not aborted, the value written by T' (that is kept in $lcell(X).value$) is returned (line 14).

The $X.\text{write}_T(v)$ operation The algorithm implementing this operation is described at lines 15-18 of Figure 2.4. If there is no local cell associated with X, p_i allocates one (line 15) and adds X to lws_T (line 16). Then it locally writes v into $lcell(X).value$ (line 17) and return *ok* (line 18). Let us observe that no $X.\text{write}_T()$ operation can entail the abort of a transaction.

2.4.1.4 The $\text{try\_to\_commit}_T()$ operation

The algorithm implementing this operation is described in Figure 2.4 (lines 19-35). A process p_i that invokes $\text{try\_to\_commit}_T()$ first locks all transaction-level shared objects X that have been accessed by transaction T (line 19). The locking of shared objects is done in a canonical order in order to prevent deadlocks.

Then, process p_i computes the values that define the last commit window of T (lines 20-21). The update of $window\_top_T$ is the same as described in the $\text{read}_T()$ operation. The update of $window\_top_T$ is as follows. For each register Y that T is about to write in the shared memory (if T is not aborted before), p_i computes the date of the last read of Y, namely the date $(PT[Y] \downarrow).last\_read$. In order not to invalidate this read (whose issuing transaction has been committed), p_i updates $window\_bottom_T$ to $\max((PT[Y] \downarrow).last\_read, window\_bottom_T)$. If the commit window of T is empty, T is aborted (line 22). All locks are then released and all local cells are freed.

If T's commit window is not empty, it can be safely committed. To that end p_i defines T's commit time as a finite value randomly chosen in the current window $]window\_bottom_T..window\_top_T[$ (let us remind that the bounds are outside the window, line 23). This time function is such that no two processes obtain the same time value.

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

a. First, for every shared object $X \in lws_T$, process p_i updates $(PT[X] \downarrow).overwrite$ with T's commit date (line 24). When all these updates have been done, for every shared object $X \in lws_T$, p_i allocates a new shared memory cell $CELL(X)$ and fills in the four fields of $CELL(X)$ (lines 26-29). Process p_i also has to update the pointer $PT[X]$ to its new value (namely $\uparrow CELL(X)$) (line 29).

b. For each register X that has been read by T, p_i updates the field *last_read* to the maximum of its previous value and $commit\_time_T$ (lines 31-33). (Actually, this base version of the protocol remains correct when $X \in lrs_T$ is replaced by $X \in (lrs_T \setminus lws_T)$. (As this improvement is no longer valid in the final version of the $\mathsf{try\_to\_commit}_T()$ algorithm described in Figure 2.6, we do not consider it in this base protocol.)

Finally, after these updates of the shared memory, p_i releases all its locks, frees the local cells it had previously allocated (line 34) and returns the value *commit* (line 35).

On the random selection of commit points It is important to notice that, choosing randomly commit points (line 23, Figure 2.4), there might be "best/worst" commit points for committed transactions, where "best point" means that it allows more concurrent conflicting transactions to commit. Random selection of a commit point can be seen as an inexpensive way to amortize the impact of "worst" commit points (inexpensive because it eliminates the extra overhead of computing which point is the best).

2.4.2 Proof of the algorithm for VWC and read invisibility

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

2.4.2.1 Proof that $\mathscr{C}$ is serializable

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

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

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

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

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

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

Proof Consider two committed transactions T and T' issued by the same process, T' being executed just after T. The variable $window\_bottom_{T'}$ of T' is initialized at $commit\_time_T$ and can only increase (during a read() operation at line 08, or during its try_to_commit() operation at line 21). Because $window\_bottom_{T'} > commit\_time_T$ (line 32), we have $T \rightarrow_S T'$. By transitivity, this holds for all the transactions issued by a process. $\Box_{Lemma\ 2}$

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

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

Because a transaction locks all the objects it accesses before committing (line 28), we can order totally the committed transactions that access a given object X. Let $\xrightarrow{X}_{lock}$ denote such a total order.

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

Proof W.l.o.g., consider that there is no transaction T'' such that $w_{T''}(X)$ and $T \rightarrow_S T'' \rightarrow_S T'$. Because $T \xrightarrow{X}_{lock} T'$, when T' executes line 21, T has already updated $PT[x]$ and the corresponding $CELL(X)$ (because there is no T'', at this time $PT[X] \downarrow = CELL(X)$). Because $X \in lws_{T'}$, $window\_bottom_{T'} \geq (PT[X] \downarrow).last\_read \geq commit\_time_T$ (line 21). We then have $commit\_time_{T'} > commit\_time_T$ and thus $T \rightarrow_S T'$. $\Box_{Lemma\ 4}$

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

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

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

Proof By way of contradiction, suppose that such a $T3$ exists. Again by way of contradiction, suppose that $T2 \xrightarrow{X}_{lock} T1$. This is not possible because $T2$ reads X before committing, and $T1$ writes X at the time of its commit (line 29). Thus $T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \xrightarrow{X}_{lock} T2$.

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

- Case $T3 \xrightarrow{X}_{lock} T2$. Let $lcell(X)$ be the local cell of $T2$ representing X. When $T2$ executes line 20), $T3$ has already updated the field *end* of the cell pointed by $lcell(X).origin$ with $commit\_time_{T3}$. $T2$ will then update $window\_top_{T2}$ at a smaller value than $commit\_time_{T3}$,contradicting the original assumption $T3 \rightarrow_S T2$.
- Case $T2 \xrightarrow{X}_{lock} T3$. When $T3$ executes line 21, $T2$ has already updated the field *last_read* of the cell pointed by $PT[X]$. $T3$ will then update $window\_bottom_{T3}$ at a value greater than $commit\_time_{T2}$, contradicting the original assumption $T3 \rightarrow_S T2$, which completes the proof of the lemma.

$\Box_{Lemma\ 5}$

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

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

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

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

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

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

$\square_{Lemma\ 6}$

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

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

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

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

In the following lemma, we use the dual notion of the causal past of a transaction: the *causal future* of a transaction. Given a partial order $\widehat{PO}$ defined on a set of transactions, the causal future of a transaction T, denoted $future(T)$, is the set including T and all the transactions T' such that $T \rightarrow_{PO} T'$. The partial order $\widehat{PO}$ used here is the one defined in Section 2.2.3.

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

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

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

When T reads Y from $T4$, it updates *window_bottom*$_T$ such that *window_bottom*$_T \geq$ *commit_time*$_{T4}$ (line 08). From the fact that $T4 \in$ *future*(*T3*), we then have that *window_bottom*$_T \geq$ *commit_time*$_{T3}$.

Either T reads Y from $T4$ and then reads X from $T1$, or the opposite. Let *last_op* be the latest of the two operations. During *last_op*, T updates *window_bottom*$_T$. Due to the fact that $T3 \in past(T)$, $T3$ has already updated the pointer $PT[Z]$ for some object Z (line 29), and thus has already updated the field *end* (line 24) of the cell pointed by $lcell(X).origin$ ($lcell(X)$ being the local cell of T representing X). T will then observe *window_bottom*$_T \geq$ *window_top*$_T$ (line 12) and will not complete *last_op*, again a contradiction, which completes the proof of the lemma. $\square_{Lemma\ 9}$

2.4.3.1 VWC and read invisibility

Theorem 3 *The algorithm presented in Figure 2.4 satisfies virtual world consistency and implements invisible read operations*

Proof The proof that the algorithm presented in Figure 2.4 satisfies virtual world consistency follows from Lemmas 1, 2, 3, 5, 6, 7, 8 and 9.

The fact that, for any shared object X and any transaction T, the operation $X.\mathsf{read}_T()$ is invisible follows from a simple examination of the text of the algorithm implementing that operation (lines 01-14 of Figure 2.4): there is no write into the shared memory. $\square_{Theorem\ 3}$

2.4.4 Step 2: adding probabilistic permissiveness to the protocol

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

2.4.4.1 The IR_VWC_P protocol

To obtain a protocol that additionally satisfies probabilistic permissiveness, only the operation $\mathsf{try\_to\_commit}_T()$ has to be modified. The algorithms implementing the operations $\mathsf{begin}_T()$, $X.\mathsf{read}_T()$ and $X.\mathsf{write}_T()$ are exactly the same as the ones described in Figure 2.4. The algorithm implementing the new version of the operation $\mathsf{try\_to\_commit}_T()$ is described in Figure 2.6. As we are about to see, it is not a new protocol but an appropriate enrichment of the previous $\mathsf{try\_to\_commit}_T()$ protocol.

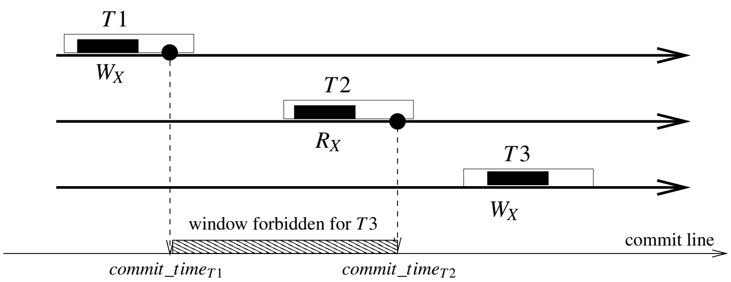

Figure 2.5: Commit intervals

A set of intervals for each transaction Let us consider the execution depicted in Figure 2.5 made up of three transactions: $T1$ that writes X, $T2$ that reads X and obtains the value written by $T1$, and $T3$ that writes X. When we consider the base protocol described in Figure 2.4, the commit window of $T3$ is $]commit\_time_{T2}..+\infty[$. As the aim is not to abort a transaction if it can be appropriately serialized, it is easy to see that associating this window to $T3$ is not the best choice that can be done. Actually $T3$ can be serialized at any point of the commit line as long as the read of X by $Y2$ remains valid. This means that the commit point of $T3$ can be any point in $]0..commit\_time_{T1}[\,\cup\,]commit\_time_{T2}..+\infty[$.

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

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

The enriched $\mathsf{try\_to\_commit}_T()$ **operation** The new $\mathsf{try\_to\_commit}_T()$ algorithm is described in Figure 2.6. In a very interesting way, this $\mathsf{try\_to\_commit}_T()$ algorithm has the same structure as the one described in Figure 2.4. The lines with the same number are identical in both algorithms, while the number of the lines of Figure 2.4 that are modified are postfixed by a letter. The new/modified parts are the followings.

- Lines 21.A-21.I replace line 21 of Figure 2.4 that was computing the value of $window\_bottom_T$. These new lines compute instead the set of intervals that constitute $commit\_set_T$. To that end they suppress from the initial interval $]window\_bottom_T, window\_top_T[$, all the time intervals that would invalidate values read by committed transactions. This is done for each object $X \in lws_T$ (line 21.H; see section 2.4.4.2 for an example). If $commit\_set_T$ is empty, the transaction T is aborted (line 22.A).
- The commit time of a transaction T is now selected from the intervals in $commit\_set_T$ (line 23.A).
- Line 24 of Figure 2.4 was assigning, for each $X \in lws_T$, its value to $(PT[X]\downarrow).end$, namely the value $commit\_time_T$. This is now done by the new lines 24.A-24.E. Starting from $PT[X]$, these statements use the pointer *prev* to find the cell (let us denote it say CX) of the list implementing X whose field $CX.end$ has to be assigned the value $commit\_time_T$. Let us remember that $CX.end$ defines the end of the lifetime of the value kept in $CX.value$. This cell CX is the first cell (starting from $PT[X]$) such that $CX.begin < commit\_time_T$.
- Line 25 of Figure 2.4 assigned its new value to every object $X \in lws_T$. Now such an object X has to be assigned its new value only if $commit\_time_T > (PT[X]\downarrow).begin$. This is because when $commit\_time_T < (PT[X]\downarrow).begin$, the value v to be written is not the last one according to the serialization order. Let us remember that the serialization order, that is defined by commit times, is not required to be real time-compliant (which would be required if we wanted to have linearizability instead of serializability, see Section 2.4.7). An example is given in section 2.4.4.3. Finally, the pointer *prev* is appropriately updated

operation try_to_commit$_T$():
(19) lock all the objects in $lrs_T \cup lws_T$;
(20) **for each** $(Y \in lrs_T)$ **do** $window\_top_T \leftarrow \min\big(window\_top_T, (lcell(Y).origin \downarrow).end\big)$ **end for**;
(21.A) $commit\_set_T \leftarrow \{\ ]window\_bottom_T, window\_top_T[\ \}$;
(21.B) **for each** $(X \in lws_T)$ **do**
(21.C) $x\_ptr \leftarrow PT[X]$; $x\_forbid_T[X] \leftarrow \emptyset$;
(21.D) **while** $((x\_ptr \downarrow).last\_read > window\_bottom_T)$ **do**
(21.E) $x\_forbid_T[X] \leftarrow x\_forbid_T[X] \cup \{\ [(x\_ptr \downarrow).begin, (x\_ptr \downarrow).last\_read]\ \}$;
(21.F) $x\_ptr \leftarrow (x\_ptr \downarrow).prev$
(21.G) **end while**
(21.H) **end for**;
(21.I) $commit\_set_T \leftarrow commit\_set_T \setminus \bigcup_{X \in lws_T} \big(x\_forbid_T[X]\big)$;
(22.A) **if** $(commit\_set_T = \emptyset)$ **then** release all locks and disallocate all local cells; return($abort$) **end if**;
(23.A) $commit\_time_T \leftarrow$ select a (random/heuristic) time value $\in$ $commit\_set_T$;
(24.A) **for each** $(X \in lws_T)$ **do**
(24.B) $x\_ptr \leftarrow PT[X]$;
(24.C) **while** $((x\_ptr \downarrow).begin > commit\_time_T)$ **do** $x\_ptr \leftarrow (x\_ptr \downarrow).prev$ **end while**;
(24.D) $(x\_ptr \downarrow).end \leftarrow \min((x\_ptr \downarrow).end, commit\_time_T)$
(24.E) **end for**;
(25.A) **for each** $(X \in lws_T)$ **such that** $(commit\_time_T > (PT[X] \downarrow).begin)$ **do**
(26) allocate in shared memory a new cell for X denoted $CELL(X)$;
(27) $CELL(X).value \leftarrow lcell(X).value$; $CELL(X).last\_read \leftarrow commit\_time_T$;
(28) $CELL(X).begin \leftarrow commit\_time_T$; $CELL(X).end \leftarrow +\infty$;
(29.A) $CELL(X).prev \leftarrow PT[X]$; $PT[X] \leftarrow \uparrow CELL(X)$
(30.A) **end for**;
(31) **for each** $(X \in lrs_T)$ **do**
(32) $(lcell(X).origin \downarrow).last\_read \leftarrow \max((lcell(X).origin \downarrow).last\_read, commit\_time_T)$
(33) **end for**;
(34) release all locks and disallocate all local cells; $last\_commit_i \leftarrow commit\_time_T$;
(35) return($commit$).

Figure 2.6: Algorithm for the try_to_commit() operation of the permissive protocol

(line 29.A). (Starting from $(PT[X] \downarrow).next$, these pointers allows for the traversal of the list implementing X.)

2.4.4.2 Substraction on sets of intervals (line 21.H of Figure 2.6)

The subtraction operation on sets of intervals of real numbers $commit\_set_T \backslash x\_forbid_T[X]$ has the usual meaning, which is explained with an example in Figure 2.7.

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

2.4.4.3 About the predicate of line 25.A of Figure 2.6

This section explains the meaning of the predicate used at line 25.A: $commit\_time_T > (PT[X] \downarrow).begin$. This predicate controls the physical write in a shared memory cell of the value v that T

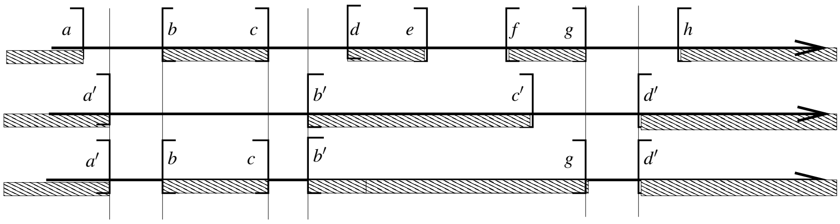

Figure 2.7: Subtraction on sets of intervals

wants to write into X (meaning that not every item in a transactions write set will be phyiscally written to memory). It states that the value is written only if $commit\_time_T > (PT[X]\downarrow).begin$. This is due to the following reason.

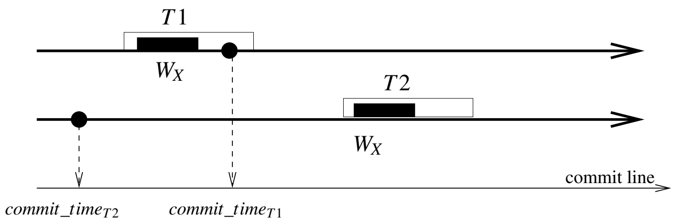

Figure 2.8: Predicate of line 25.A of Figure 2.6

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

An example is described in Figure 2.8. Transaction $T1$, that invokes $X.\mathsf{write}_{T1}()$, executes first (in real time), commits and is serialized at (logical) time $commit\_time_{T1}$ as indicated on the Figure. Then (according to real time) transaction $T2$, that invokes also $X.\mathsf{write}_{T2}()$, is invoked and then commits. Moreover, its commit set and the random selection of its commit time are such that $commit\_time_{T2} < commit\_time_{T1}$. It follows that $T2$ is serialized before $T1$. Consequently, the last value of X (according to commit times) is the one written by $T1$, that has overwritten the one written by $T2$. The predicate $commit\_time_T > (PT[X]\downarrow).begin$ prevents the committed transaction $T2$ to write its value into X in order write and read operations on X issued by other transactions be in agreement with the serialization order defined by commit times.

2.4.5 Proof of the probabilistic permissiveness property

In order to show that the protocol is probabilistically permissive with respect to virtual world consistency, we have to show the following. Given a transaction history that contains only committed transactions, if the partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ accepts a legal linear extension (as defined in Section 2.2), then the history is accepted (no operation returns abort) with positive probability. As in [8], we consider that operations are executed in isolation. It is important to notice here that only operations, not transactions, are isolated. Different transactions can still be interlaced.

Let $\rightarrow_S$ be the order on transactions defined by the protocol according to the $commit\_time_T$ variable of each transaction T (this order has already been defined in Section 2.4.2).

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

Proof Because $T \not\to_{PO} T'$, the set $past(T) \setminus past(T')$ is not empty ($past(T)$ does not contain T'). Let $biggest\_ct_{T,T'}$ be the biggest value of the *commit_time* variables (chosen at line 23.A) of the transactions in the set $past(T) \cap past(T')$ if it is not empty, or 0 otherwise. Suppose that every transaction in $past(T) \setminus past(T')$ chooses the smallest value possible for its *commit_time* variable. These values cannot be constrained (for their lower bound) by a value bigger than $biggest\_ct_{T,T'}$, thus they can all be smaller than $biggest\_ct_{T,T'} + \varepsilon$ for any given ε.

Suppose now that T' chooses a value bigger than $biggest\_ct_{T,T'} + \varepsilon$ for its *commit_time* variable. This is possible because, for a given transaction $T1$, the upper bound on the value of $commit\_time_{T1}$ can only be fixed by a transaction $T2$ that overwrites a value read by $T1$ (lines 11 and 20). Suppose now that $T1$ is T'. If there was such a transaction $T2$ in $past(T)$, then there would be no legal linear extension of $\to_{PO}$ in which T precedes T'. Thus if there is a legal linear extension of $\to_{PO}$ in which T precedes T', then there is a positive probability that $T \to_S T'$. $\square_{Lemma\ 10}$

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

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

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

From Lemma 10, two transactions that are not causally related can be totally ordered in any way that allows a legal linear extension of $\to_{PO}$. There is then a positive probability that $\to_{op} \subset \to_{legal}$. Suppose that it is true. Let T be the transaction executing op. Let $T1$ and $T2$ be the transactions directly preceding and following T in $\to_{legal}$ restricted to $\mathscr{C}_{op} \cup \{T\}$ if they exist. If $T1$ does not exist, then $window\_bottom_T = 0$ at the time of all the operation of T, and thus T can execute successfully all its operations. Similarly, if $T2$ does not exist, then $window\_top_T = +\infty$ at the time of all the operation of T, and thus T can execute successfully all its operations. We will then consider that both $T1$ and $T2$ exist.

Because $\to_{legal}$ is a legal linear extension of $\to_{PO}$, any transaction from which T reads a value is either $T1$ or a transaction preceding $T1$ in $\to_{op}$ (line 08) resulting in a value for $window\_bottom_T$ that is at most $commit\_time_{T1}$. Similarly, any transaction that overwrote a value read by T at the time of op is either $T2$ or a transaction following $T2$ in $\to_{op}$ (line 11) resulting in a value for $window\_top_T$ that is at least $commit\_time_{T2}$. All the read operations of T will then succeed (line 12).

Let us now consider the case of the try_to_commit() operation. Because all read operations have succeeded, the set $commit\_set_T =]window\_bottom_T..window\_top_T[$ (line 21.A) is not empty and must contain the set $]commit\_time_{T1}..commit\_time_{T2}[$. Because $\to_{legal}$ is a legal linear extension of $\to_{PO}$, if T writes to an object X then T cannot be placed between two transactions $T3$ and $T4$ such that $T3$ reads a value of object X written by $T4$.

Because these intervals (represented by $x\_forbid_T[X]$, lines 21.B to 21.H) are the only ones removed from $commit\_set_T$ (line 21.I) and because there is a legal linear extension of $\rightarrow_{PO}$ which includes T, $commit\_set_T$ is not empty and the transaction can commit successfully, which ends the proof of the lemma. $\Box_{Lemma\ 11}$

Theorem 4 *The algorithm presented in Figure 2.4, where the* try_to_commit() *operation has been replaced by the one presented in Figure 2.6, is probabilistically permissive with respect to virtual world consistency.*

Proof From Lemma 11, all transactions of a history can commit with positive probability if the history is virtual world consistent, which proves the theorem. $\Box_{Theorem\ 4}$

2.4.6 Garbage collecting useless cells

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

Additional arrays The first is an array of atomic variables denoted $LAST\_COMMIT[1..m]$ (remember that m is the number of sacred objects). This array is such that $LAST\_COMMIT[X]$ (which is initialed to 0) contains the date of the last committed transaction that has written into X. Hence, the statement "$LAST\_COMMIT[X] \leftarrow commit\_time_T$" is added in the **do** ... **end** part of line 24.

The second array, denoted $MIN\_READ[1..n]$, is made up of one-writer/one-reader atomic registers (let us recall that n is the total number of processes). $MIN\_READ[i]$ is written by p_i only and read by the background task BT only. It is initialized to $+\infty$ and reset to its initial value value when p_i terminates $\mathsf{try\_to\_commit}_T()$ (i.e., just before returning at line 22 or line 35 of Figure 2.4). When $MIN\_READ[i] \neq +\infty$, its value is the smallest commit date of a value read by the transaction T currently executed by p_i. Moreover, the following statement has to be added after line 03 of the $X.\mathsf{read}_T()$ operation:

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

Managing the *next* pointers When a process executes the operation $\mathsf{try\_to\_commit}_T()$ and commits the corresponding transaction T, it has to update a pointer *next* in order to establish a correct linking of the cells implementing X. To that end, p_i has to execute $(PT[X]\downarrow).next \leftarrow\uparrow CELL[X]$ just before updating $PT[X]$ at line 29.

The background task BT This sequential task, denoted BT, is described in Figure 2.9. It uses a local array denoted $last_valid\_pt[1..m]$ such that $last\_valid\_pt[X]$ is a pointer initialized to $PT[X]$ (line 01). Then its value is a pointer to the cell containing the oldest value of X that is currently accessed by a transaction (this is actually a conservative value).

The body of task BT is an infinite loop (lines 02-12). BT first computes the smallest commit date still useful (line 03). Then, for every shared object X, BT scans the list from $last\_valid\_pt[X]$ and releases the space occupied by all the cells containing values of X that are

(01) **init: for every** X **do** $last\_valid\_pt[X] \leftarrow PT[X]$ **end for**.

background task BT:
(02) **repeat forever**
(03) $min\_useful \leftarrow \min(\{MIN\_READ[i]\}_{1 \leq i \leq n})$;
(04) **for every** X **do**
(05) $last \leftarrow last\_valid\_pt[X]$;
(06) **while** $\big((last \neq PT[X]) \wedge (last \downarrow).next \downarrow).next \neq \bot)$
(07) $\wedge[(((last \downarrow).next \downarrow).next \downarrow).commit\_time < min\_useful]\big)$
(08) **do** $temp \leftarrow last$; $last \leftarrow (last \downarrow).next$; release the cell pointed to by $temp$
(09) **end while**;
(10) $last\_valid\_pt[X] \leftarrow last$
(11) **end for**
(12) **end repeat**.

Figure 2.9: The cleaning background task *BT*

no longer accessible (lines 06-09), after which it updates $last\_valid\_pt[X]$ to its new pointer value (line 10). Lines 06 and 07 uses two consecutive *next* pointers. Those are due to the maximal concurrency allowed by the algorithm, more specifically, they prevent an $X.\mathsf{read}_T()$ operation from accessing a released cell.

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

2.4.7 From serializability to linearizability

The IR_VWC_P protocol guarantees that the committed transactions are serializable. A simple modification of the protocol allows it to ensures the stronger "linearizability" condition [13] instead of the weaker "serializability"condition. The modification assumes a common global clock that processes can read by invoking the operation $\mathsf{System.get\_time}()$. It is as follows.

- The statement $window\_bottom_T \leftarrow last\_commit_i$ at line 01 of $\mathsf{begin}_T()$ is replaced by the statement $window\_bottom_T \leftarrow \mathsf{System.get\_time}()$.
- The following statement is added just between line 20 and line 21 of $\mathsf{try\_to\_commit}_T()$ (Figure 2.4):
  if $(window\_top_T = +\infty)$ **then** $window\_top_T \leftarrow \mathsf{System.get\_time}()$ **end if**.

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

2.4.8 Some additional intersting properties

[NOTE!!!!: !!!!!!!!!NEED TO INTRO/UPDATE THIS SECTION!!!!!!] Interestingly enough, this new STM protocol has additional noteworthy features: (a) it uses only base read-

/write operations and a lock per object that is used at commit time only and (b) satisfies the disjoint access parallelism property.

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

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

Multi-versioning The proposed IR_VWC_P protocol uses multiple versions (kept in a list) of each shared object X. Multi-version systems for STM systems have been proposed several years ago [4] and have recently received a new interest, e.g., [1, 21]. In contrast to our work, none of these papers consider virtual wold consistency as consistency condition. Moreover, both papers consider a different notion of permissiveness called multi-version permissiveness that states that read-only transactions are never aborted and an update transaction can be aborted only when in conflict with other transactions writing the same objects. More specifically, paper [21] studies inherent properties of STMs that use multiple versions to guarantee successful commits of all read-only transactions. This paper presents also a protocol with visible read operations that recovers useless versions. Paper [1] shows that multi-version permissiveness can be obtained from single-version. The STM protocol it presents satisfies the disjoint access parallelism property, requires visible read operations and uses k-Compare&single-swap operations.

2.5 Improving the base protocol described in Figure 2.4

The previous sections investigated the interatcions between a few different STM properties and consistency conditions. It showed that premissiveness, opacity, and invisible reads are imcompatable, but if opacity is switched for virtual world consistency then not only are the properties compatible, they can be satisfied by a realistic protocol. This work focused mostly on theory, showing concepts to be possible or not. This study of theory is important as it gives us a starting point from where to design practiacl STM algorithms, but does not provide a complete examination of STM as it leaves us with no practiacl implementations.

This section begins an investigaion towards this more practiacl study of algorithms. What we have observed is that in practice we might not want a perfectly permissive algorithm because it would likely have too much bookkeeping overhead just in order to commit a small number of transactions that otherwise would be aborted unnecessarily. Specially we noticed that there were interesting ways that the “not quite” permissive algorithm of figure 2.4 could be modified in order to make the read operations cheaper whithout impacting the correctness of the algorithm that were not compatible with the permissive algorithm of figure 2.6.

2.5.1 Expediting read operations

Let us consider the invocations $X.\mathsf{read}_T()$ (those are issued by T). From the second one, none of these invocations access the shared memory. As described in Figure 2.4, the first invocation $X.\mathsf{read}_T()$ entails the execution of line 11 whose costs is $O(|lrs_T|)$. Interestingly, it is possible to define "favorable circumstances" in which the execution of this line can be saved when $X.\mathsf{read}_T()$ is invoked for the first time by T. Its cost becomes then $O(1)$. To that end, each process p_i is required to manage an additional local variable denoted $earliest\_read_T$ (that is initialized to $+\infty$ at line 01 of Figure 2.4). Line 11 of $X.\mathsf{read}_T()$ is then replaced by:

if $(lcell(X).commit\_time > earliest\_read_T)$
then Code of line 11 **else** $earliest\_read_T \leftarrow lcell(X).commit\_time$ **end if**.

Hence, the protocol allows for *fast* read operations in favorable circumstances. (It is even possible, at the price of another additional control variable, to refine the predicate used in the previous statement in order to increase the number of favorable cases. Such an improvement is described in Appendix 2.5.2. It is important to notice that, while these *fast* read operations are possible when the consistency condition is VWC, they are not when it is opacity.)

2.5.2 More on fast read operations

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

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

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

An implementation This part discusses an implementation of the previous idea. While a transaction is live it keeps a local variable called $latest\_read_T$ initialized as $commit\_time_i$ and updated during each $X.read_T()$ operation to the largest commit time of the transactions it has read from so far. When T executes line 21 of the try_to_commit$_T()$ operation, the variable $latest\_read_T$ is (possibly) increased to the largest commit time of the transaction for the values T is overwriting. A boolean control variable, $overwrites\_latest\_read_T$ (initialized to *false*) is set to *true* if $latest\_read_T$ is modified. To that end line21 is replaced by the code described in Figure 2.10.

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

Figure 2.10: Fast read: Code to replace line 21 of Figure 2.4

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

Finally, line 11 of X.read$_T()$ is replaced by the following statement:

(a) **if** $\big((CELL(X).latest\_read > earliest\_read_T) \vee$

(b) $\quad (CELL(X).latest\_read = earliest\_read_T \wedge CELL(X).overwrites\_latest\_read)\big)$

then Code of line 11 **end if**;

$earliest\_read_T \leftarrow \min(lcell(X).commit\_time, earliest\_read_T)$;

$latest\_read_T \leftarrow \max(lcell(X).commit\_time, latest\_read_T)$.

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

- The first corresponds to line (a), i.e., when the predicate $CELL(X).latest\_read > earliest\_read_T$ (let T_1 be the transaction that wrote $CELL(X)$) is satisfied, meaning that there is at least one transaction in the causal past of T that has a commit time earlier than either a transaction in the causal past of T_1, or a transaction that has written a value overwritten by T_1. Thus a value read by T might have been overwritten by T_1 or $past(T_1)$ and update_window_top$_T()$ needs to be executed.

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

Fast read operations and opacity As mentioned previously performing fast reads in this way is only possible in VWC. We give below a counterexample (Figure 2.11) in order to show that they are not compatible with opacity.

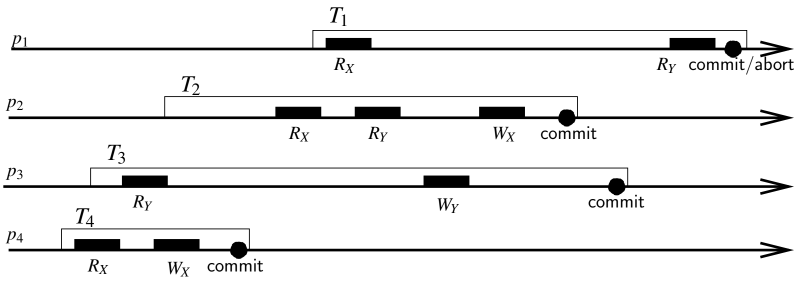

Figure 2.11: Example of fast reads that would violate opacity

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

2.5.3 Making read operations invisible at commit time

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

Similarly to fast read operations where line 11 of X.read$_T()$ does not need to be always executed, a read does not always need to be made visible during the try_to_commit$_T()$ operation. There are two locations in try_to_commit$_T()$ that modify shared memory with respect to objects $X \in lrs_T \setminus lws_T$. The first is on line 32 where the value of $cell(X).last\_read$ is updated for each object $X \in lrs_T$. The second is on line 19 where each object $X \in lrs_T$ is locked. Concerning efficiency and scalability, not only can locking and writing to shared memory be considered expensive operations, but is is desirable for reads to be completely invisible.

Discussion: When write to shared memory and lock can be avoided Assume some shared variable Y read by transaction T. First consider the write to $cell(Y).last\_read$. In the algorithm, the only time the $last\_read$ field of a cell is read is on line 21 of the try_to_commit$_T()$ operation. Since $last\_read$ is only read for objects in lws_T, and T locks all objects in lws_T, T is guaranteed to be accessing the most recent cell. This means that, during a $\mathsf{try\_to\_commit}_T()$ operation on line 32, if $lcell(Y).origin$ is not the latest cell for Y, then the value of *last_read* written will never be accessed and there is no reason to write the value.

So the write on line 32 is not necessary when $lcell(Y).origin$ is not the latest cell, what about the lock associated with Y? When is it not necessary to lock objects in lrs_T? This turns out to not be necessary in the same case. If line 32 is not executed for some variable $Y \in lrs_T$, then the only other place a cell of Y is accessed in the $\mathsf{try\_to\_commit}_T()$ operation is on line 20. Here the only shared memory accessed is $(lcell(Y).origin\downarrow).end$. From the construction of the algorithm $(lcell(Y).origin\downarrow).end$ will be updated at most once, therefore if the loop iteration for Y on line 11 of $\mathsf{X.read}_T()$ (where X and Y may or may not be the same variable) had been executed previously with $(lcell(Y).origin\downarrow).end \neq +\infty$ (meaning $(lcell(Y).origin\downarrow).end$ had been updated previously), then the loop iteration for Y does not need to be executed again which results in Y not needing to be locked.

An implementation It follows from the previous discussion that a read operation of a shared variable Y can be made invisible at commit time if $(lcell(Y).origin\downarrow).end$ had not equal to $+\infty$ during the loop iteration for Y on line 11 for any execution of $\mathsf{X.read}_T()$. Implementing this in the algorithm becomes easy, line 11 must be replaced by the following.

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

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

Additional Benefits It is also worth noting that this improvement can also improve the performance of the read operations performed by the STM. This is because when the one of the new lines added is executed an object is removed from lrs_T, and the cost of the read operation depends on the size of lrs_T. In the best case this can cause the cost of a read operation to be $O(1)$ instead of $O(|lrs_T|)$ (note that this works concurrently along with fast reads as described in Appendix 2.5.2).

2.6 Conclusion

To conclude this chapter we will return to the idea that the main pourpose of transactional memory is to make concurrent programming easier. Overall the contribution of this chapter is a study of the interaction between several properties and consistency conditions of transactional memory. So then what does this have to do with the ease-of-use of transactional memory?

When designing an STM protocol it is necessary that we design it to satisfy some consistency criteron. This is important because these consitency criterion define the semantics of a transaction to the programmer. As long as the choosen consistency criterion is satisfied then we can start considering other secondary, but still important things such as performance. This is where different properties such as read invisiblity and permissiveness come in, a protocol that chooses or not to implement such properties might impact the performace of that protocol, but does not change the semantics of a transaction. By doing this we are putting the ease-of-use for the programmer as a first class requirement.

The first contribution of this chapter shows that permissive and read invisibility are incompatible with opacity, we then show that we can choose a weaker consistency criterion (virtual world consistency) to design a protocol that satisfies permissiveness and read invisibility. Most importatnly, even though we have weakend the consistency criteron, the programmer will see no difference in the semaintics of a transaction. If virtual world consistency changed the way a programmer had to think about transactions versus opacity then it would not be considered an appropriate consistency criteron for transactional memory.

The second contribution of this chapter introduces a relistic algorithm that satisfies virtual world consistency, permissiveness, and read invisibility. It then shows some ways to optimize the speed of the read operations of the protocol by trading-off some permissiveness.

An important distinction of this work from most is that instead of looking at properties independtly, it considers multiple properties at once. More specifally it looks at the compaability of properties, how realistic protocols can be desinged to satisfy them, and how trade-offs in these proterties can be made for efficiency. Hopefully these contributions have convinced the reader that studing such combinations is important for understanding transactional memory and will help inspire similar research on the many combinations left unexamined. Importantly this suggests that weakening the transactional model in the interest of improving performance might not be necessary, as much of what is possible under the current model has yet to be exaimed.

As a final note on ease-of use, it should be said that the research direction suggested in this chapter is also similar in a way to that of the majority of published work on transactional memory in that it concerns itself with designing an efficient protcol without modifying the idea of a transaction. Each of the protocols presented in these various works have no impact on how the programmer understands the transaction which, in a way, puts ease-of-use first. The following chapters will take a different direction, examing some of what we consider shortcomings of the commonly used transactional model and how to improve it.

Chapter 3

Universal Constructions and Transactional Memory

3.1 Introduction

The previous chapter focused on a few contributions to an area of transactional memory research that takes a fixed view of the semantics of a transaction for the programmer and studies what can be done in an STM protocol without changing the semantics. This type of STM research puts first the ease-of-use for the programmer using the STM protocol before considering secondary intrestes (usually performance). In most cases this ease-of-use is ensured by having the protocol satisfy opacity. Opacity is used because generally it is considered to ensure the ammount of safety a programmer would expect from an atomic block without having to worry about inconsistencies of aborted transactions (Note that in some cases virtual world consistency is used as it does not change how the programmer views a transaction). This chapter takes a slightly differnt appproach to STM research as it suggest that transactional memory might be more usable to a programmer if it satisfied more then just opacity.

Opacity and other consistency criteron are generally considered to be safety properties. Informally this is because they ensure a protocol that implements them will act in a way that a user would expect and not produce any weird behavoir. For example opacity prevents any transaction from executing on invalid states of memory, preventing things such as divide by zero exceptions in correct code. What such consistency criteron do not consider (among other things) is how often transactions commit. For example a protocol could satisfy opacity by just aborting every transaction before it performs any action, but of course this protocol would be useless. In order to avoid this, certain STM protocols satisfy liveness (or progress) properties These properties, not limited to transactional memory, ensure the operations of a process will make some sort of progress sometimes depending on the ammount of contention in the system. Let us now look at some of these properties.

3.1.1 Progress properties

This section will give an overview of the most common progress properties defined for concurrent algorithms. They are arranged into two categories, blocking and non-blocking.

3.1.1.1 Blocking properties

The most common way to write concurrent programs is by using locks, generally lock based programs satisfy blocking progress properties. A property is blocking when a thread's progress can be blocked because it is waiting for another thread to perform some actaion. For example thread A might want to accquire lock l, but thread B currently owns lock l so then thread A waits for thread B to release lock l. In this case thread A is blocked by thread B. The three most common blocking properties are (from weakest to strongest) *deadlock freedom*, *livelock freedom*, and *starvation freedom*. They are described breifly in the following paragraphs.

Deadlock freedom Deadlock freedom is the weakest blocking property, it prevents the implementing protocol form entering a state of deadlock. Deadlock occurs when at least two threads are preventing eachother from progressing due to eachother holding a lock (or resourse) that the other wants to accquire. A simple example of deadlock would be the following: thread A owns lock $l1$ and wants to accquire lock $l2$, concurrently thread B owns lock $l2$ and wants to accquire lock $l1$. In this case thread A and B will be stuck infinately, wating to accquire the lock that the other already owns creating a state of deadlock. It is generally considered that every correct concurrent protocol should at least satisfy deadlock freedom. A common way to avoid deadlock freedom is by ensuring threads accquire locks in a fixed global order.

Livelock freedom Stronger then deadlock freedom, livelock freedom prevents a state of livelock in which threads can never progress due to their progess depending on a shared state that is created by another thread. Consider the follwing simple example: in order to progress a thread must own locks $l1$ and $l2$ concurrently. Thread A runs a protocol that first acquires $l1$ and then $l2$, while thread B runs a protocol that first acquires $l2$ then $l1$. In order to avoid deadlock when a thread noticies that another thread owns a lock it wants, it relases all the locks it owns and starts the locking protocol over. Consider the events of thread A and B happen in the following order: $A.acquire(l1) = success$, $B.acquire(l2) = success$, $A.is\_owned(l2) = true$, $B.is\_owned(l1) = true$, $A.release(l1)$, $B.release(l2)$.

[NOTE!!!!: NEED TO DEFINE THIS HISTORY STRUCTURE]

If such an order of events is continually reapeted then livelock is observed. It should be noted that by definition livelock freedom also ensure deadlock freedom.

Starvation freedom The strongest blocking property, starvation freedom ensure that no threads starve. A thread is starved when it requires access to some shared resourses in a certain state to progress, but it is never able to such gain access to them due to one or more concurrent "greedy" threads that consistantly own the needed resources. Starvation freedom prevents both deadlock and livelock from happening.

3.1.1.2 Non-blocking properties

There are also several non-blocking progress properties of concurrent programming. Unlike the blocking properties these ensure that a thread's progress is never blocked due to it waiting for another thread. Inherantly this means that it is not possible protocols that use standard locks to be non-blocking. Instead of using locks, non-blocking protocols generally use atomic operations such as test-and-set or compare-and-swap (in a non-blocking fashion) when synchronization between threads is necessary. Generally programming using these operations in this way is considered to be much more difficut then programming using locks.

The problem with blocking If non-blocking algorithms are more difficult to program then why not just use locks? When concerning scalibility, non-blocking algorithms have two main advantages over their blocking counterparts.

The first most obvious advantage is by definition, in a non-blocking algorithm a thread will never be blocked waiting for another thread. Normally waiting might seem to be necessary part of synchronization, in our everyday when working together with someone on a project we will wait for a teammate to finish their task before starting ours. But then consider a massive project that involves hundreds of people across the world in multiple organizations, one person on this project might have an approching deadline and he might not want to wait on someone across the world that he has never met before in order to start his task. Similar situations can exist in largely parallel computer systems, threads might be spread across multiple processors or machines, some might be sleeping, some might execute slower then others, the data connection between some processors might be slower then others. In such cases the ammount of time a thread might have to wait could be unknown and harmful to scalibility.

The second advantage is when faults are considered. If a thread is waiting for a lock that is owned by another thread that has crashed then without fault detection and recovery this thread will be waiting forever. Non-blocking algorithms on the other hand by definition do not have to worry about this. Given that fault detection and recovery is a difficult problem especially in massively parrallel systems this is an obvious advantage of non-blocking algorithms.

Non-blocking The first non-blocking property is *obstruction-freedom*. A protocol that is obstruction-free ensures that any thread will eventually make progress as long as it is able to run by itself for long enough. This property ensure that no thread is ever stuck wating for another thread, but only guarantees progress in the absence of contention.

Lock-freedom For certain tasks progress might be required even in the face of contention, in such cases non-blocking is not strong enough. *Lock-freedom* ensures that at any time there is at least one thread who will enventually make progress. Even though some threads may starve, in a lock-free algorithm we at least know that the system as a whole is making progress.

Wait-freedom An even stronger progress property *wait-freedom* ensures that all threads eventually make progress. This is a nice property to ensure as each thread in the system is only dependent on itself and not other threads for making progress.

3.1.2 Universal Constructions for concurrent objects

Given the increased progress guarenteed by wait-free protocols they are desireable over lock based protocols. Unfortunately wait-free protocols are known to be extreemly difficult to write and understand.

Two years before the concept of transactional memory was introduced, the notion of a universal construction for concurrent objects (or concurrent data structures) was introduced by Herlihy [41].

Like transactional memory, a universal construction's main concern is with making concurrent programming easier. A universal construction takes any sequential implementation of an object or data structure and makes its operations concurrent, wait-free, and linearizable. The concurrent objects suited to such constructions are the objects that are defined by a sequential specification on total operations (i.e., operations that, when executed alone, always return a result). For example data structures are the typical example of the use of a universal construction.

A brief introduction to a universal construction protocol Upon first inspection it might appear to be a nearly impossible task to design a construction that can automatically turn the operations of a sequential object into a concurrent one with such a strong progress guarantee as wait-freedom. Fortunately even though the fine details of the universal construction proposed in [41] might be intracate, the key design concepts are quite clear.

The first concept has to deal with correctness. How to ensure that the sequentail code is exectued safely when there can be multiple threads concurrently performing operations on the object? This is ensured simply by each thread operating on a local copy of the object. Before a thread starts executing the original sequential code, it makes a copy of the object in its local memory and the operation is performed on that object. Once the sequential operation is complete it must be then made visible so other threads can be aware of the modificaiton. In order to acheive this there is a single global operation pointer. An operation completes by performing a compare and swap on this pointer changing it to point to a descriptor of its operation. The value swapped out must be the same as it was when the operation started, if not the operation discards its modifications and starts over with a new up to date local copy. Unfortunately this means that best case performance will be no better then a single thread, but in certain cases this might be an acceptable trade-off for wait-free progress.

The second key concept has to deal with liveness. As described in the previous paragraph an operation completes by modifying a global pointer with a compare-and-swap operation, but this operation can fail due to a concurrent modifcation to the global pointer by some other thread. Now according to the progress guaratee of wait-freedom, every operation by every thread must eventually complete successfully without blocking, meaning the compare-and-swap must not fail infinately many times. The key concept used to ensure this is helping. Since a failed compare-and-swap can only be caused by a different thread succeeding with its compare-and-swap, why not have this successful thread help the thread that failed? Simply put helping here means that several threads will all execute the operation of a thread whos operation has failed in order to ensure that that operation eventually succeeds. When helping it is important to ensure that each operation is not performed several times.

Alternative universal constructions Since the original, several universal constructions have been proposed (e.g., [25, 26, 34]) focusing on ensuring different properties or increased efficiency. Interstingly many of the key design concepts from the original universal construction such as helping are also included in these designs.

One intresting example related to the work in this thesis is a universal construction for wait-free *transaction friendly* concurrent objects presented in [30]. The words "transaction friendly" means here that a process that has invoked an operation on an object can abort it during its execution. Hence, a 'transaction friendly" concurrent object is a kind of abortable object. It is important to notice that this abortion notion is different from the notion of transaction abortion. In the first case, the abort of an operation is a programming level notion that the construction has to implement. Differently, in the second case, a transaction abort is due the implementation itself. More precisely, transaction abortion is then a system level mechanism used to prevent global inconsistency when the system allows concurrent transactions to be executed optimistically (differently, albeit very inefficient, using a single global lock for all transactions would allow any transaction to executed without being aborted).

(A main issue solved in [30] lies in ensuring that, without violating wait-freedom, an operation op issued by a process p_i is not committed in the back of p_i by another process p_j which is helping it to execute that operation.)

3.1.3 Transactional memory and universal constructions

The previous sections fist discussed progress properties for concurrent code in general followed by a discussion on universal constructions which can be used to create a concurrent wait-free construction of any sequential object. The following sections will discuss how progress properties can be considered in transactional memory as well as the relation between univeral constructions and transactional memory.

3.1.4 Progress properties and transactional memory

NOTE: That most STM don't ensure progress just for speed's sake.

The previously mentioned blocking and non-blocking progress properties were defined for concurrent code in general and do not concern the specifics of transactional memory. The key difference to consider between transactional memory traditional concurrent code is that transactions can abort and restart. Does an aborted transaction entitle progress? If we are just considering that code being executed entitles progress then yes, but when considering the ease-of-use of transactional memory it is more intersting to consider that only committed transactions create progress. The question of what to do with aborted transactions is an important one, the following section first looks at how the previously mentioned progress properties can be applied to transactional memory followed by a breif overview of how progress is approached in transactional memory.

How blocking and non-blocking progress properties relate to transactional memory Aborted transactions are not mentioned specifally in any of the blocking or non-blocking progress properties. Without considering aborted transactions it might not be very interesting to have a transactional memory protocol that satisfies one of them as the protocol could still just abort every transaction, being completely useless to a programmer using the protocol.

We can then simply extend these properties by adding additional requirements for the committing of transactions. For example we might want a lock-free transactional memory protocol to ensure that at least one of the live transactions in the system will eventually commit. A more detailed analysis of non-blocking progress properties and transactional memory has been performed in []. In this work they define the non-blocking properties *solo-progress* as a equivalent to the obstruction-freedom property for transactional memory and *local-progress* as a equivalent of wait-freedome for transactional memory. Informally a protocal that satisfies solo-progress must ensure that every process they executes for long enough must make progress (where progress requires eventually committing some live transaction) while a protocol satisfying local-progress must ensure that "every process that keeps executing a transaction (say keeps retrying it in case it aborts) eventually commits it." Additionaly in [] they examine how these properties can be appiled in faulty and faut-free systems with or without parisitic transactions (a parisitic transaction is one which is continually executed, but never tries to commit). They show that local-progess is impossible in a faulty system where each transaction is fixed to a certain process. An extended discussion on the possible/impossible liveness properties of a STM system is presented by the same authors in [40] where it 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.

3.1.5 Previous approaches

In order to ensure levels of progress and cope with aborted transactions, several solutions have been proposed witch each taking differnt approches to progress. A whole range of solutions has been proposed. Some do not directly confront the problem of progress, focusing mainly on the performance of the protcols, while others offer "best effort semantics" (which means that there is no provable strong guarantee) and others offer provable guarantess of progress.

Programmer's task The least complex solution simply leaves the management of aborted transactions to the application programmer (similarly to exception handling encountered in some systems). In such systems the programmer has the choice to have the protocol execute a chosen set of code when a transaction is aborted one or several times. This can be a powerful option for an experience programmer who knows the details of an transactional memory implemntation, but this thesis takes the view that the promary goal of transactional memory is ease-of-use and having programmers have to manage aborted transactions themselves goes against this goal.

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) [42] and much research has been done on the topic since then.

In some cases the contention manager's goal is to improve performance while others ensure (best effort or provable) progress guarantees or a combination. An associated theory is described in [38]. Failure detector-based contention managers (and corresponding lower bounds) are described in [39]. 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 [52].

An overview of different contention managers and their performance is presented in [38]. 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 [50], 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. In these blocking solutions, a transaction's eventual commit depends on both on the process that issued this transaction as well as the process that issued the transaction with the oldest timestamp.

Unfortunately even though such contention managers exist that ensure all transactions commit, most STM implementations do not use them in the interest of performance and as a result provide less strong progress guarentees. As a solution to avoid these performance problems while still eventually providing strong progress, some modern STM's (e.g., for example TinySTM [36] or SwissSTM [33]) 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 Another approach to dealing with aborts consists in designing schedulers that decide when and how transactions are executed in the system base on certain properties. One approach is to design schedules that perform particularly well in appropriate workloads, for example the case of read-dominated workloads is deeply investigated in [28].

Another interesting approach is called *steal-on-abort* [27]. 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 $T2$. Interestingly in order to help a transaction commit, this scheduler allows a transaction to be executed and committed by a processor different the one it originated from. 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.

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). This priority given to the running irrevocable transaction allows it to guarantee to succed, but does so at the cost of preventing other transactions from progressing until it finishes. STM protocols supporting ireovokable transactions are proposed and discussed in [51] and [54]. Irrevocable transactions suited to deadline-aware scheduling are presented in [47].

Robust STMs Ensuring progress even when bad behavior (such as process crash) can occur has been investigated in several papers. As an example, [53] 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 nonterminating transactions.

Obstruction-Freedom, Lock-Freedom There have been several proposals for non-blocking STM protocols, some of them are obstruction-free (e.g., [42, 23]), while others are lock-free (in the sense there is no deadlock) [8].

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).

Unfortunately, none of them provides the property that every issued transaction is committed. As described in the previous section, in order for the described universal constructions ensure that each operation is performed succesfully, threads must help other threads by executing eachother's operations. In previously proposed non-blocking STM, helping only occurs with transactions that are in the process of committing. With only this type of help, some transaction can be aborted indefinitely without violating safety. Consider for example a thread $T1$ transaction $t1$ and a seperate thread $T2$ that executes an infinate sequence of the same transaction $t2$. Now simply consider a history as follows that is repeated infiantely, $t2$ starts executing, $t1$ starts executing, $t2$ commits, $t1$ noticies that it conflicts with $t2$ so it must abort. In such a history $t1$ will always abort before it reaches the commitial phase so if blocking is not allowed, helping only in the commital phase will not ensure the commitial of every transaction.

3.1.6 Ensuring transaction completion

As seen, there are many ways to cope with aborted transactions and to ensure progress in transactional memory. Unfortunately none of these solutions quite realize to goal of ease-of-use for the programmer is still concerned with the idea of abort/commit. In some cases the programmer has to deal directly with aborted transactions in his code, in others a programmer can prioritize certain transactions so they will not abort, others allow transactions to be blocked, while other allow transactions to be aborted infinately. Absent from these solutions is the case where all transactions are guaranteed to commit where the progress of a transaction does not rely on the other processes then the one that issued the transaction. Such a solution would prioritize ease-of-use as such a protocol would hide the concept of aborted transactions from the programmer.

More precisely we want a non-blocking STM protocol which ensures that every transaction issued by a process is eventually committed wheres its progress only depends on the issuing process. Then, 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 [13], 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 remainder of this chapter focuses on the design of a such protocol that. (As a note it should be said that the presentation of previous approches that deal with aborted transactions is not entirely fair as they are primarily efficiency-oriented while our construction is more theory-oriented.)

3.1.7 A short comparison with object-oriented universal construction

The first point to notice is that an STM that hides the concept of commit/abort from the programmer has objectives similar to that of a universal consturction described earier in this chapter. A universal construction allows a programmer to turn a sequential object into a concurrent one where each operation is linearizable and completes successfully, while the STM protocol we want here is one that allows a programmer to place transactions in his code each of which are executed successfully and are leaniable. Although similar, the key difference between these lies in the difference between an operation and a transaction.

There is an important and fundamental difference between an operation on a concurrent object (e.g., a shared queue) and a transaction performed by an STM protocol. **[NOTE!!!!: Clean up this]** Albeit the operations on a queue can have many different implementations, their semantics is defined once for all. Differently, each transaction is a specific atomic procedure whose code can be seen as being any dynamically defined code. What is significant here is that any transaction is able to read and write to any location in shared memory while an operation in a universal construction is fixed to a predefined set, which is often a single instance of a data structure. Simply put, a programmer might want to use a universal construction when he has an object with predefined self-contained operations that he wants to use concurrently (such as a data structure) while transactions might be more suitable when the programmer wants to perform general atomic operations within his code.

To briefly examine how these differences can effect the implementation of a protocol we will look at the universal construction proposed by Herlihy in [41] (denoted H_UC in the following). Interestingly H_UC could be used for STM programs simply by piecing together all the shared objects objects into a single concurrent object *TO*, and considering all the transactions as operations on this object *TO*. This brute force approach is not conceptually satisfying. When considering lock-based mechanisms, it is like using a single lock on the single "big" object *TO*, instead of a lock per object. Moreover, as it requires each operation on an object to make a copy of this object (before accessing it), H_UC would force each operation to copy the whole shared memory, even it it works on a very small subset of its content. This is not the case in the construction proposed in this chapter, where only the specific locations accesed by a transaction needs to be copied. The space granularities required by H_UC (when applied to STM) and the proposed construction do not belong to the same magnitude order. Traditionally STM protocols commonly use a read and write set with the pourpose of tracking the locations the STM has read so far as well as those that will be modified upon commit, validating these sets in order to ensure correctness. In the case of the STM protocol presented in this chapter these read and write sets have the additional benifit of saving us from making copies of the entire shared memory.

Even given the differences, the proposed STM protocol in this chapter borrows key ideas from previous universal constructions such as helping and using a shared global pointer that is modififed using a compare-and-swap in order to ensure progress and correctness.

Another interesting feature of the proposed construction (which is specifically designed for STM programs) is the systematic use of speculative execution. Even if efficiency is not a first class requirement addressed in our work, the notion of a speculative execution can be a basis for future work on universal construction that will focus on efficiency. As discussed in section 3.1.2, a traditional universal construction can expect best case performance to be equal to that of a sequental implementation. While in the case of the STM's speculative execution any number of transactions are able to execute concurrently (and successfully if no conflict is found) by performing validations on their read sets. In some ways, the computation cost of performing validation can be seen as a trade off in order to allow for higher concurrency.

3.2 A universal construction for transaction based programs

The following secations present a new STM construction that, in order to hide the notion of abort/commit from the programmer, ensures every transaction issued by a process is necessarily committed and each process makes progress. More specifcally it ensures linearizable X-ability. X-ability, or exactly-once ability, was originally defined by Frølund and Guerraoui in [37] as a correctness condition for replicated services such as primary-backup. In this model there are actions, such as transactions, that cause some side effect. For a service to satisfy X-ability, every invoked action and its side effect must be observed as if it had happened exactly once. In order to ensure this, actions might be executed multiple times by the underlying system. Given this requirement, X-ability concerns both correctness and liveness and can complement concurrency correctness conditions.

To our knowledge, this is the first STM system we know of to combine these concepts in a realistic protocol. Given the similarities between this STM based construction and universal constructions as well as the differences between transactions and operations on objects we define this protocol as a "universal construction for transaction based programs".

3.3 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.

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 where each processor is in charge of a subset of processes. The multiprocess program, defined by the programmer, is made up of n processes where each process is a separate thread of execution. 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 for the transaction to 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 [41]). 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.

3.3.1 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 the description of 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) each invocation of 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, any transaction invocation 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 only accesses a single t-object once. Hence encapsulating 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., [48]). 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).

3.3.2 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 $X.\mathsf{Compare\&Swap}(old, new)$:
$atomic\{$ **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.

3.4 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 [41]).

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.

In our universal construction, the STM controls entirely the transactions; this is different from what is usually assumed in STMs [40].

3.4.1 Control variables shared by the processors

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

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 the entry $OWNED\_BY[x]$ contains the set of 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 the t-objects of the 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 $\langle \mathrm{X}, v\rangle$ 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 or the non-transactional code that follows T in the code of p_i.

Helping mechanism: the array $LAST\_CMT[1..m, 1..n]$ This array is such that $LAST\_CMT[x, i]$ contains the sequence number of process p_i's last committed transaction as known by processor P_x. $LAST\_CMT[x, i]$ is written only by P_x. Its 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 for the last transaction issued by p_i to 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 in the transaction list after the transaction pointed to by pt.

3.4.2 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 $\langle X, v \rangle$ 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 a fixed local pointer variable *current* towards *FIRST*) the descriptor list until encountering the first pair $\langle X, v \rangle$, 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).

3.4.3 Behavior of a processor: initialization

Initially a processor P_x executes the non-transactional code (if any) of each process p_i it owns until p_i's first transaction and then initializes accordingly the atomic register $STATE[i]$. Next P_x invokes $\mathsf{select}(OWNED\_BY[x])$ that returns the identity of a process it owns, this value is then assigned to P_x's local variable *my_next_proc*. P_x also initializes local variables whose role will be explained later. This is described in Figure 3.1.

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, $\mathsf{select}(\emptyset) = \perp$.

for each $i \in OWNED\_BY[x]$ **do**
    execute p_i until the beginning of its first transaction;
    $STATE[i] \leftarrow \langle 1, p_i\text{'s current local state}, +\infty, FIRST \rangle$
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 3.1: Initialization for processor P_x ($1 \leq x \leq m$)

3.4.4 Behavior of a processor: main body

The behavior of a processor P_x is described in Figure 3.2. This 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 01-12) 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 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 07-12). 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 $(\downarrow current).next = \perp$ 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 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 13-18) 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 14-18). The set of concurrent t-objects read by p_i is saved in P_x's local set lrs, and the pairs $\langle$Y,$v\rangle$ such that the transaction issued Y.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 19-33) Once 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, but only if certain conditions are satisfied. To that end, P_x enters a loop (lines 20-25). There are two reasons for not trying 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 22-23).
- The second is when the transaction is 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 24.

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 26). To do this it first creates a new descriptor $DESCR$, updates its fields with the data obtained from its speculative execution (line 28) and then tries to add it to the list. To perform the commit, P_x issues Compare&Swap$((\downarrow current).next, \perp, \uparrow DESCR)$. 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 29).

Finally, if the transaction has been committed P_x updates $LAST\_CMT[x,i]$ (line 33).

Use the ownership notion to ensure the progress of each process (Lines 34-47) 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 for p_i's transaction to be eventually committed. To that end, it assigns (if not yet done) the next date value to $STATE[i].help\_date$ (lines 36-39). Then, P_x proceeds to the next loop iteration. (Let us observe that, in that case, my_next_proc is not modified.)

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

Figure 3.2: Algorithm for processor P_x ($1 \leq x \leq m$)

- Given that P_x is responsible for p_i's progress, if *committed* is equal to *true* then P_x executes the non-transactional code (if any) that appears after the transaction (line 40). Next, if p_i has terminated (finished its execution), i is suppressed from $OWNED\_BY[x]$ (line 42). Otherwise, P_x updates $STATE[i]$ in order for it to contain the information required to execute the next transaction of p_i (line 43). 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 45).

3.4.5 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 so that no processor either (a) permanently helps only processes owned by other processors or (b) loops forever in an internal while loop (lines 06-12 or 20-25). 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.

The procedure select_next_process() This operation is described in Figure 3.3. 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 for a "too long" period, requires help from the other processors.

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*) either 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 119), 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 109-114). In that case, P_x also resets $k1\_counter$ to 0 (line 115).

```
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)
(109)                 then let j = my_next_proc;
(110)                      if (state[j].help_date = +∞)
(111)                        then helpdate ← Fetch&Incr(CLOCK);
(112)                             STATE[j] ←
(113)                               ⟨state[j].tr_sn, state[j].local_state, helpdate, state[j].last_ptr⟩
(114)                      end if;
(115)                      k1_counter ← 0
(116)               end if
(117)        end if
(118) end if;
(119) return(i).
```

Figure 3.3: The procedure select_next_process()

Let us remark that the procedure select_next_process() implements a kind of aging mechanism, which is similar the one used by some schedulers to prevent process starvation.

```
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 3.4: Procedure prevent_endless_looping()

The procedure prevent_endless_looping() As indicated, the aim of this procedure, described in Figure 3.4, is to prevent a processor P_x from endless looping in an internal while loop (lines 05-09 or 18-22).

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 45 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 251), then P_x will require help for p_i once this period attains $K2$ (lines 253-256). 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).

3.5 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 3.1-3.4 generates an execution of *PROG*.

Lemma 12 *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 3.2) 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 Fetch&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 3.2) 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 33, 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. $\square_{Lemma\ 12}$

Lemma 13 *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 37-38, 111-112 or 255-256). 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 12 that there is a time after which all the processors that call select_next_process() obtains the process identity i. Let $\mathscr{P}$ 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 $(\downarrow \mathit{current}).\mathit{next} = \bot$ becomes eventually true, we conclude that at least one processor $P_y \in \mathscr{P}$ cannot be blocked forever in a while loop. Because the list is no longer changing, the predicate of line 26 then becomes satisfied at P_y. It follows that, when the processors of $\mathscr{P}$ execute line 29, 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. $\Box_{Lemma\ 13}$

Lemma 14 *No processor P_x loops forever in an internal while loop (lines 06-12 or 20-25).*

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. Because processor P_x loops forever we never have $((\downarrow \mathit{current}).\mathit{next} = \bot) \vee \mathit{committed}$, but each time it executes the loop body, P_x invokes prevent_endless_looping(i) (at line 07 or 21). The code of this procedure is described in Figure 3.4. As $i \in \mathit{OWNED\_BY}[i]$ and P_x invokes infinitely often prevent_endless_looping(i), it follows from lines 251-253 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 45) 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 253-256). It then follows from Lemma 13, 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 $(\downarrow \mathit{current}).\mathit{tid} = \langle i, i\_tr\_sn\rangle$ (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 operation select_next_process()). The proof follows from the fact that, due to Lemma 13, T is eventually committed. As previously (but now *current* is P_y's local variable), the predicate $(\downarrow \mathit{current}).\mathit{tid} = \langle i, i\_tr\_sn\rangle$ eventually becomes true and processor P_y sets *committed* to *true*. P_y then stops looping inside an internal while loop (line 08 or 23) which concludes the proof of the lemma. $\Box_{Lemma\ 14}$

Lemma 15 *Any invocation of a transaction T by a process is eventually committed.*

Proof Considering a processor P_x, let $i \in \mathit{OWNED\_BY}[x]$ 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 43 where it has updated $\mathit{STATE}[i]$ that now refers to that transaction. If P_x requires help for T, the result follows from Lemma 13. 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 14), it eventually executes lines 34-39 and consequently either commits T or requires help for T at line 38.

- select_next_process() never returns i. In that case, as P_x never loops forever in a while loop (Lemma 14), 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 107-116) and, due to Lemma 13, T is eventually committed.

Let us now observe that that, after T has been committed (by some processor), P_x executes lines 40-45 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. $\Box_{Lemma\ 15}$

Lemma 16 *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 29 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 06-29.

- We conclude from the last update of $STATE[i].last\_ptr = pt$ by P_y (line 43) 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 $(\downarrow current).next = \perp$. It then follows from lines 08 and 23 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. $\Box_{Lemma\ 16}$

Lemma 17 *Each invocation of a transaction T by a process is committed exactly once.*

Proof The proof follows directly from Lemma 15 and Lemma 16. $\Box_{Lemma\ 17}$

Lemma 18 *Each invocation of non-transactional code issued by a process is executed exactly once.*

Proof This lemma follows directly from lines 40-45: 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). $\Box_{Lemma\ 18}$

Lemma 19 *The simulation is starvation-free (no process is blocked forever by the processors).*

Proof This follows directly from Lemma 14, Lemma 17, Lemma 18 and the definition of the function select(). $\Box_{Lemma\ 19}$

Lemma 20 *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). As far as item (a) is concerned, the linearization point of a transaction invocation is defined as follows[1].

- 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 26, 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.

[1] 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).

- $P_x = P_y$ (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 16). After processor P_x has executed the non-transactional code that follows the invocation of T (if any, line 40), it updates $STATE[i].local\_state$ with the current value of i_local_state (if p_i had not yet terminated, line 43).
- $P_x \neq P_y$ (the processor that commits T and the owner of p_i are different processors). In that case, P_y has saved the new local state of p_i in $DESCR.local\_state$ (line 28) 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 23 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 43).

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).

- 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.

$\square_{Lemma\ 20}$

Lemma 21 *The simulation of a transaction-based n-process program by m processors (executing the algorithms described in Figures 3.1-3.4) is linearizable.*

Proof Let us first observe that, due to Lemma 20, 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 [13][2] and (b) TO is linearizable and every nt-object is linearizable, it follows that the execution of the multiprocess program is linearizable.

$\square_{Lemma\ 21}$

Theorem 5 *Let PROG be a transaction-based n-process program. Any simulation of PROG by m processors executing the algorithms described in Figures 3.1-3.4 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 19 and Lemma 21. The execution

[2] 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 [13] that linearizability is a local property.

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.

$\Box_{Theorem\ 5}$

3.6 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 22 *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 38 after the transaction invocation T aborts. It follows from line 103 of the operation 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 35 and 36 and the statements of line 45.

$\Box_{Lemma\ 22}$

Theorem 6 *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 38).

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.

- 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 22 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 16 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.

$\square_{Theorem\ 6}$

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 $O(m^2)$.

3.7 Conclusion

3.7.1 A short discussion

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 [51, 54]. 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. Furthermore due to the helping mechanism, once a process has asked for help with a transaction that transaction is guaranteed to commit as long as there exsits at least one live failure free process.

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 [29] (where there are simulators that execute processes) and Herlihy's universal construction to build wait-free objects [41] (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.

The previous chapter explored an area of transactional memory research that focuses on improving STM protocols without effecting how the user interacts with the STM, that is by ensuring some implementation level properties or by increasing performance. This chapter, while similiar to the previous chapter in suggesting propertes and showing how a protocol can implement them, takes a more visible approch that directly effect the interaction between the programmer and the STM. Abstractly, it examines how the semantics are defined between the programmer and the STM protocol and suggests they be simplified. Previous research has expected some level of interaction between the programmer and aborted transactions, while this chapter suggests the notion of commit/abort be completely abstracted away from the programmer level left to be soely an implemntation concern. This frees the programmer from having to consider if his transacion might not commit and to eiter try to prevent such a situation, or to come up with ways to deal with it when it does.

As a final motivation for thse simplifited semantics, let us consider how it compares to the consistency condition of opacity. Opacity differs from linearizability or serializibility in that it frees the programmer from having to worry about consistency issuses that could arrise in aborted transactions. This liveness suggessted in this chapter for STM protocols differes from previous suggestions in that it frees the programmer from having to worry that his transaction might not commit. In a way it can be considered the equivalent to opacity except opacity considers correctness (bad things happening in aborted transactions), while this chapter considers liveness (a transaction that is only aborted).

Without opacity the programmer has to come up with solutions in order to prevent things like such as invalid pointers, infinate loops, or divide by 0 errors from happening in aborted transactions. Without the liveness suggested in this chapter a programmer has to come up with solutions in the case that a transaction is not able to progress due to the actions of some other process in the system.

Further research is still needed on how the semantics of a transaction can be simplified. For example currently transactions are only able to contain reads and writes, while things that input and output are not allowed, and the interaction between transactions and other synchronization methods is undefined or prohibited.

Chapter 4

Ensuring Strong Isolation in STM Systems

4.1 Introduction

Simplified transaction semantics may not be enough. In the interest of ease-of-use, each chapter in this thesis has had a primary focus on transaction semantics. The first chapert suggested improving STM protocols without changing the semantics of a transaction, the second chapter suggested simplifying the semantics of a transaction, and this chapter will suggest expanding the semantics. It will look at how a programmer might use transactions within his code in order to suggest the expansion of transactional semantics again in the interest of ease-of-use.

Following the example of the previous chapter we will promote our change of semantics by contransting transactions to concurrent objects.

A discussion on semantics the of concurrent protocols is meaningless without first considering consistency conditions. By satisfying a consistency condition a concurrent protocol allows the user of that protocol to reason about how he can use it in his program in a concurrent setting. For example having a concurrent data structure that satisfies linearizability means that the operations of a data strcuture will have a global order based on the invocation and completion time of operations. This allows, for example, a programmer who is coding some user based service to use reasoning such as: if user A who is being serviced by thread T_A changes his status to online (adding him to the set of online users backed by a linearizable concurrent data structure) then a following query for the names of online users by user B serviced by a different thread T_B is guaranteed to contain user A. Without linearizability the programmer cannot necessarily use this reasoning, he might then have to consider that the query by thread T_B might or might not include A leaving the programmer to have to come up with a solution for this.

Likewise in transactional memory, a correctness condition helps the programmer reason about how he can use transactions in his programs. For example opacity ensures that all transactions have a global order and each transaction appears to have happened instantly at some point in time between its invocation and commital. This allows the programmer to create mult-process programs where the processes are synchronized using programmer defined atomic operations. The most common example of this is a banking application defined by transactions. A programmer writing such a bank application may want to define an operation that transfers a given ammount of money m from account A to another accout B. In order to do this he creates a transaction that first reads the balance on the source accout ($A.balance$), and if the account has enough money the transfer proceeds by first setting $A.balance$ to the value previously read minus m. Next the balance of account B is read, before setting its new balance to the value read plus m and the transaction is completed. Given that the programmer uses an STM protocol that ensures transactions are executed atomically, a user of this program will see only valid transfers completed, otherwise conditions might arrise where a blanace is concurrently modified by another process during a transaction, resulting in invalid balances where someone could loose or gain too much money.

As the above examples illustrate, it is necessary for a concurrent protocol to satisfy a clear and straightforward consistency condition in order for a programmer to reason about how to use it. Now we will suggest that due to the way transactions are used in a program an STM protocol must satisfy more then a correctness condition such as opacity that is only coconcerend with transactions. The reason for this is best seen by going back to the comparison of a transaction and an operation on a concurrent object.

For a concurrent object consider the common example of a tree data structure implementing the set abstraction. This abstraction might provide the following lineariable operations: $insert(K)$ which adds K to the set if it does not already exist, returning $true$ on success, $delete(K)$ which removes K, if it exists, from the set returning $true$ on success, and $contains(K)$ returning $true$ if K exists in the set, $false$ otherwise. Here there are two importnat things to point out, first is that these operations are contained to their data structre implementation, the status of shared memory outside of the data structure has no affect on these operations. Second is that these operations fully implemnt the desired abstraction (in this case a set), a user chooses to use this specific implementation because he needs the $insert$, $delete$, and $contains$ operations. If he wanted addtional functionality such as in-order iteration, then he would have to choose a different implemntation providing the appropriate abstraction. This might lead the programmer to consider data structure as a seperate entity from that of thit program where this entity is accessable through some fixed interface, he need not be aware of the implementation below the interface.

Transactions on the other hand are integreated into the users program rather then compromising some seperate entity. When a user places a trasaction in his code it is because he requires synchronization between the processes of his program, this synchronization does not have to be based on some predefined abstraction or only access some contained structure. The concept of the transaction allows the programmer to be free to perform any sort of operation, performing reads and write to any location in the shared memory. In this sense the transaction is a different type of mechanism then a concurrent object as transactions are integreated and contained within a user's program instead of being separate entities behind a fixed interface.

So we have concurrent objects whose operations follow some correctness condition allowing them to be accessed as something interesting to the programmer as a seperate entity. And transactions which appear as atomic blocks (thanks to the STM protocol satisfying a correctness condition) built into a users program. At a low level the difference between a transaction and an operation on a shared object might imply separate protocol design chocies. For example the previous chapter highlighted some of the implementation differences between a traditional universal construction and a universal construction for transaction based programs.

But, let us step backwards for a moment and look at a higher level. Before we consider implementation details we must consider how how the programmer interacts with a transaction is considered at an abstraction level. The previous discussion has expressed the view that transactions exsit as an integrated part of a users program. This happens when the programmer places a group of reads and writes inside a block of code bouded by the keywords *transaction_begin* and *transaction_end*. Inside this block of code reads and writes to shared memory are then treated by the STM protocol as calls to the operations *transaction_read* and *transaction_write*. The STM protocol then takes care of the difficult synchronization in order to ensure these transactions appear as if they happen atomically. At this point we have an important choice to make dealing with the semantics and correctness of a transaction when considering shared memory. Should the transactions appear to be atomic only with other transactions, or should concurrent non-transactional shared memory accesses also be considered? More precisely, should the shared memory that is accessed from within a transaction be only safely accessable from within transactions, or should it be safe to access memory inside and outside of transactions at any time, or should it be something inbetween?

Interestingly, this possibility of the *existence of two different paradigms* reveals two different interpretations of transactional memory: On one hand considering TM as an implementation of shared memory, and, on the other hand, considering TM as an additional way of achieving synchronization, to be used alongside with locks, fences, and other traditional methods.

When putting ease-of-use as the primary requirement of transactional memory as well considering the discussion on transactions vs operations on concurrent objects the choice of paradigm seems clear. If within a program there are two sets of shared memory, one set for accessing inside transactions, and another set for accessing outside of transactions, then we loose a bit of the integration concept for a bit more of the seperated object concept. One way to think of it is that the transactions represent at programmer defined interface to one large discrete shared object. Transactions are still defined and placed by the programmers, but they are limited in what memory they can access. On the other hand if the programmer can access shared memory both inside and outside of transactions while the system is still ensurubg the atomicity of the transactions then we have a more straightforward integrated system.

4.1.1 Transaction vs Non-transactional Code.

Let us now define precisely what we mean by trasactional vs, non-transactional accesses. In a concurrent environment, shared memory may be accessed by both transactions as well as non-transactional operations. In this work we consider that shared memory can be accessed through reads and writes either inside or outside of transactions. A read (respectively write) performed inside a transaction is consider a transactional read (transactional write) while a read (resp. write) performed outside a transaction is considered a non-transactional read (non-transactional write).

The next section will discuss previous solutions on how to deal with concurrent accesses to shared memory inside and outside of transactions and where we think they fall short on the ease-of-use objective.

4.1.2 Dealing with transactional and non-transactional memory accesses

Weak isolation STM protocols implementing *weak isolation* make no guarantee about concurrent transactional and non-transactional memory accesses. Transactions are considered to happen atomically only with respect to other transactions. Given this, it is possible for non-transactional operations to see intermediate results of transactions that are still live. Conversely, a transaction may see the results of non-transactional operations that happened during the transaction's execution. Nevertheless, these concurrecny issuses can be anticipated and used appropriately by the programmer, still resulting in correctly functioning applications. Ways of how a programmer might consider to use weak isolation in his code is presented in [111]. This requires the programmer to be conscious of eventual race conditions between transactional and non-transactional code that can change depending on the STM system used. Not only can this make transactional code extreemly difficult to write and understand, but a simple change in the STM protocol could render the program useless. Obviously this is not a good solution to concurrent transactional and non-transactional access when considering ease-of-use.

More commonly, an STM protocol that implemnts weak isolation expects users to avoid any concurrent interaction between the shared memory outside and inside of transactions. This is because the type of consistency guaranteed is usually undefined and depends on the implementation of the STM protocol used. A programmer already has to consider different types of memory such as thread local process local and shared memory so having him decide what memory should be transactional and what should not be just adds an additional burden to the programmer. Along with this, having a seperate section of memory for transactions is also not consistent with the concpet of transactions being integrated within a program.

Therefore, in order to keep consistent with the spirit of TM principles a system should prevent unexpected results from occurring in presence of race conditions. Furthermore, concurrency control should ideally be implicit and never be delegated to the programmer [**?**, 112]. It is for these reasons that when considering transactional and non-transactional accesses we want something stronger then weak isolation in order to keep things as simple as possible for the programmer.

It is important to note that most current high performance STM protocols implement weak isolation as providing stronger guarantees tend to have an impact on performance.

Specialized transactions In some cases a programmer using an STM that satisfies weak isolation can deal with concurrency issues between transactional and non-transactional accesses himself. Usually a programmer will porpously subject himself to these difficulties in the interest of performance. Examining more formally these performance over ease-of-use concepts, serveral STM protocols have been proposed allowing the existance of transactions that might not apper as being atomic. These protcols often focus on ways to give the programmer the power to weaken the semantics of certain transactions or even specific reads within transactions. DSTM [42] introduced the concept of *early-release*, giving a programmer the ability to remove locations from the read set of a transaction, in the hope of improving performance by lessening the likelyhood of a transactional abort. *Elastic* transactions [66] allow a programmer to mark transactions that satisfy a weaker consistency condition then opacity allowing for efficient implementations of search data structures. While this chapter focuses on the interaction between shared memory accesses inside and outside of transactions, we felt it was interesting to mention these solutions as in a way they allow a sort of non-transactional read to be preformed from within a transaction.

Privitization Privitization is an interesting solution problem to the splitting of memory between transactions and non-transactional accesses. As the name implies, privitization gives the programmer the ability to privitize sections memory to a specific process giving it exclusive access to memory that was previously shared. The programmer does this through the use of what is called a privitizing transaction within which he makes some memory unaccessable from other processes' transactions. Once a process has privitized some memory then is safe to access it outside of transaction within the privitizing thread. After privitizaion, memory can then be publicized to be accessed from within transactions again by using a publication transaction.

A typical example of privatization would be the manipulation of a shared linked list. The removal of a node n by a transaction T_i, for private use, through non-transactional code, by the process p that invoked T_i, constitutes privatization. Then, T_i is called privatizing transaction. Obviously after the transaction commits future transactions will no longer be observe n in the list, but unless the STM protocol is privitization safe p cannot safely access n non-transactionally. Normally this is due to the fact that there could be live transactions that started before T_i committed who have concurrently accessed n. These problems are closely examined in [118]. Here they show that prviding privitization safety is not as simple as just implemnting an STM protcol that provides opacity as incorrect results due to privatization can occur regardless of the update policy (redo log or undo log) that a transactional memory algorithm implements. As an example, consider the cases that result when given two processes p_1 and p_2 that privatize shared variable x:

- The TM implementation uses a redo log. Process p_1 privatizes x with privatizing transaction t_1 but stalls before committing t_1. p_2 executes during this stall and will not see the effects of t_1. p_2 proceeds to privatize x itself and to access it through non-transactional operations. The variable x will still be accessed through transaction t_1 when process p_1 resumes and the results of p_2's privatization will not be visible to it.
- The TM implementation uses an undo log. As above, p_2 privatizes x but p_1 stalls before performing validation before attempting to commit. In the meanwhile, p_2 executes, privatizes x and commits. Given that updates are done in-place, p_2 will observe the updates performed by t_1. However, when t_1 resumes and attempts to commit, its validation will fail and it will abort. p_2 will be left privately accessing data that are no longer valid.

[NOTE!!!!: Need to put in citations for these, update this paragraph] Several solutions have been proposed for the privatization problem such as using visible reads, conventional means of synchronization such as fences, sandboxing, partitioning by consensus [115], the use of lock-free reference counters [106] or by using private transactions [108], to name a few. Several protocols implementing privitization have been proposed, ranging from solutions where a programmer uses a special keyword to mark which transactions are privitizing, to others that ensure safety by having a privtizing transaction block until all other concurrent transactions have finished, to more complex non-blocking solutions.

A main advatage of privitizaion is in the case of performance, as when a thread privitizes some memory, it can then access it without overhead. While privitization is an interesting solution, we feel that is does not fit in our model of ease-of-use for two main reasons. Firstly the seperation between the memory safe to access by transactions and that safe to access outside of transactions is still separated, privitization/ publication just gives the user the abitiy to dynamically move sections of memory from one part to another. Secondly it adds an extra layer of complication to the transactional memory abstraction, if the programmer wants to takes advatage of privitization he has to decide when and what to privitize and publicize.

Solution from scott/spear [NOTE!!!!: Need to write and cite]

Strong isolation *Strong isolation*, in order to spare the programmer from the responsibilities raised by weak isolation or privitizaion, ensures that both the transactions and the non-transactional read and write operations are be implemented in a way that takes their co-existence into account. Strong isolation ensures that the non-transactional accesses to shared memory do not violate the atomicity of the transactions. As a result, the aforementioned scenarios described in the paragraphs about weak isolation and privitization where non-transactional operations violate transaction isolation, are not be allowed to happen. In fact strong isolation inherently also solves the privatization problem as it imposes synchronization between transactional and non-transactional code.

In order to better understand strong isolation guarantees, consider the follwoing simple intuitive extention that can be applied to an STM protocol in order to ensure strong isolation. In order to provide this extention, any non-transactional operation that accesses shared data is simply treated as a "mini-transaction", i.e., a transaction that contains a single read/write operation. In that case, transactions will have to be consistent (see Sect. 4.2) not only with respect to each other, but also with respect to the non-transactional operations. Obviously in this solution there is no seperate section of memory for transactions and their atomicity is preserved resulting in a simplier framework for the programmer then privitization or weak isolation. While this solution is simple, it is not necessarily optimal performance wise. As such, several more efficient approaches to implementing strong isolation have been proposed, in [116] a blocking implementation to strong isolation is described by placing barriers in transactions to ensure safety. In the intrest of improving performance, this paper also proposes ways of performing dynamic and static analysis in order to remove certain barriers while still ensuring safety. Further dynamic optimizations proposed in [114], but even considering the most efficient implementations, strong isolation has been suggested to be too costly [107]. Other work has considered strong isolation in hardware transactional memory [113].

When implementing strong isolation in order to ensure the safety of non transactional reads and writes (weather they are implemeted as mini-tranactions or by using memory barriers) they become more then just a single load or store operation when actually being executed on the processor. In order not to increase the complexity of using transactional memory we want the programmer to be able to perform non-transactional operations in his code just as he would a normal read and write. This means at some point after the programmer has written the code and before it is executed the additional operations must be added. For example in [116, 114] memory barriers are added before non-transactional reads and writes by the compiler. This can also be done dynamically at runtime.

Precisely Defining Strong Isolation. The distinction of isolation guarantees was originally made in [111], where reference was made to "weak atomicity" versus "strong atomicity". Interestingly this paper also presents examples of code that a programmer might write to run along with an STM protocol that provides some type of weak isolation which would not run using an STM protocol that provides strong isolation, further suggesting that a standard should be proposed so that the programmer is not stuck having to uderstand the intracacies provided by different STM protocols.

Further complicating things, there are different definitions in literature for strong isolation [111, 110, 109]. In this paper we consider strong isolation to be the following: (a) non-transactional operations are considered as "mini" transactions which contain only a single read or write operation, and (b) the consistency condition for transactions is opacity (or virtual world consistency). We choose this definition as it is simple and keeps with the spirit of atomic transactions without having to seperate the memory.

Using this definition, the properties referd to as *containment* and *non-interference* are satisfied. These properties were defined in [111] in order to describe the syncronization issues between concurrent transactional and non-transactional operations. Containment is illustrated in the left part of Fig. 4.1. There, under strong isolation, we have to assume that transaction T_1 happens atomically, i.e.,"all or nothing", also with respect to non-transactional operations. Then, while T_1 is alive, no non-transactional read, such as R_x, should be able to obtain the value written to x by T_1. Non-interference is illustrated in the right part of Fig. 4.1. Under strong isolation, non-transactional code should not interfere with operations that happen inside a transaction. Therefore, transaction T_1 should not be able to observe the effects of operations W_x and W_y, given that they happen concurrently with it, while no opacity-preserving serialization of T_1, W_x and W_y can be found. Non-interference violations can be caused, for example, by non-transactional operations that are such as to cause the ABA problem for a transaction that has read a shared variable x.

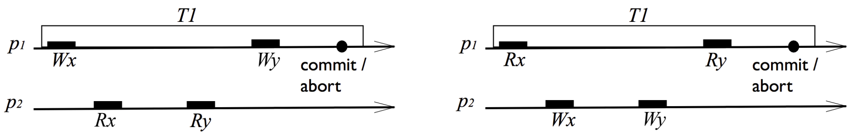

Figure 4.1: Left: *Containment* (operation R_x should not return the value written to x inside the transaction). Right: *Non-Interference* (wile it is still executing, transaction T_1 should not have access to the values that were written to x and y by process p_2).

4.1.3 Terminating Strong Isolation

Strong isolation allows transactions to not be restricted to use a fixed subset of memory while still ensuring their atomicity with repect to other transactions as well as non-transactional operations. This allows the programmer to preform reads ands write to the same memory he accesses in his transactions without having to worry about observing or creating inconsistencies. As indicated in the paragraph on strong isolation, one possible implementation is to simply convert all non-transactional reads and writes into mini-transactions. Let us also consider the solution to the problem of ensuring strong isolation by using locks or barriers: Each shared variable would then be associated with a lock and both transactions as well as non-transactional operations would have to access the lock before accessing the variable. Locks are already used in TM algorithms - such as TL2 itself - where it is however assumed that shared memory is only accessed through transactions. The use of locks in a TM algorithm entails blocking and may even lead a process to starvation. Previous research on STM protocols has taken the approach that these characteristics might acceptable. The reason for this is that the programmer accepts the fact that a transaction has a duration and that it may even fail: Given that every live transaction has the posibility to abort means that the failure to complete can be considered a part of the transaction concept.

Now if we have non-transactional operations that rely on the same mechanisms (either implemnted using locks or by mini-transactions) then they are sucseptable to the same posibilty of non-completion. However one must consider that a transaction does not have the same semantics as a simple read or write operation. While the concept of the memory transaction includes the possibility of failure, the concept of a simple read/write operation does not. When it comes to single read or write accesses to a shared variable, a non-transactional operation is normally understood as an event that happens atomically and completes. While executing, a read or write operation is not expected to be de-scheduled, blocked or aborted. Unfortunately strong isolation implemented with locks entails the blocking of non-transactional read and write operations and would not provide termination. In the previous chapter we argued that, when considering ease-of-use, every transaction should be committed and the possible non-termination of transactions is unacceptable. Even worse would be if a programmer had to consider that simple read and write operations to shared memory are not guaranteed to terminate. Such an approach to strong isolation would then be rather counter-intuitive for the programmer (as well as possibly detrimental for program efficiency).

For this reason we believe that an STM protocol implementing srong isolation should also consider the progress of non-transactional operations. In order to deal with this we suggest that a STM protcol should implemnt *terminating strong isolation*, which we simply define as strong isolation with the additional guarantee that the non-transactional read and write operations of a process are guaranteed to terminate no matter the actions of concurrent processes in the system. The rest of this chapter will focus on the design of an STM protocol that ensures terminating strong isolation.

4.2 Implementing terminating strong isolation

As previously described, the goal is to design a protocol in which non-transactional read and write operations never block or abort i.e. *terminating strong isolation*. Before diving into the design of a new protocol, let us consider how this relates to the protocol described in the previousl chapter. The previous chapter introduced an STM protocol in which every transaction is guaranteed to commit where the progress of a transaction only depends on its issuing process. This helps hide the notion of abort for the programmer, but does not prevent aborts from happeneing at all. In fact any transaction is allowed to abort, but only a finite number of times. When implementing terminating strong isolation on the other hand we want to compeltely avoid non-transactional operations from aborting. The most obvious reason for this is because we want non-transactional reads and write to behave the same as simple atomic reads and writes.

The second, less theoretically intersting, but just as important reason is performance. To the programmer the non-transactional operations as simple reads and writes, he should be able to assume that they are efficient. In fact, the primary concern of previous research on strong isolation is performance, it has even been suggested that

In the previous chapter we were more interested in showing that a concept was possible, while in this chapter performace is a just as important concern. Given this we cannot simply implement non-transactional operations as we did transactions in the previous chapter's protocol.

Still it is not necessary to design a completely new protocol, instead we choose to take the base design of an efficient state-of-the-art STM protocol that provides weak isolation and extend it to provide terminating strong isolation. The TL2 protocol was choosen for this as it is fairly straightforward and can be considered as one of the fastest STM implementations. In the redesigned TL2 protocol presented in this chapter read and write operations that appear inside a transaction follow the original TL2 algorithm rather closely (cheap read only transactions, commit-time locking, write-back), with the addition of non-transactional read and write operations that are to be used by the programmer, substituting conventional shared memory read and write operations in order to provide terminating strong isolation.

4.3 A Brief Presentation of TL2

TL2, aspects of which are used in this paper, has been introduced by Dice, Shalev and Shavit in 2006 [?]. The word-based version of the algorithm is used, where transactional reads and writes are to single memory words. Instead of presenting the detailed alorithm we will only present the main concepts of the algorithm that are used in the protocol presented in this chapter. The safety condition ensure by TL2 for transactions is opacity. It should be noted that the original version of TL2 takes no consideration into non-transactional memory accesses taking the view that transactions should be relegated to their own separate section of memory. Extended versions of TL2 that are privitizaion safe have also been exaimed [63].

Main Features of TL2. The shared variables that a transaction reads form its *read set*, while the variables it updates form the *write set*. Read operations in TL2 are *invisible*, meaning that when a transaction reads a shared variable, there is no indication of the read to other transactions. Write operations are *deferred*, meaning that TL2 does not perform the updates as soon as it "encounters" the shared variables that it has to write to. Instead, the updates it has to perform are logged into a local list (also called *redo log*) and are applied to the shared memory only once the transaction is certain to commit.

One of the key features of TL2 is that read-only transactions are considered efficient. This is because they do not need to maintain local copies of a read or write set and because if they reach the try_to_commit operation before aborting then they can commit immediately as they are guaranteed to be consistend. To control transaction synchronization, TL2 employs locks and logical dates.

Locks and Logical Date. A lock and a logical date is associated with each shared variable. TL2 implements logical time as an integer counter denoted *GVC*. When a transaction starts it reads the current value of *GVC* into local variable, *rv*. This value is used in order to ensure the transaction views a consistent state of memory.

When a transaction attempts to commit it first has to obtain the locks of all the variables in its write set, before it can update them. Furthermore, a transaction has to check the logical dates of the variables in its read set in order to ensure that the values it has read correspond to a consistent snapshot of shared memory. Its read set is valid if the logical date of every item in the set is less than the transaction's *rv* value. If, on the contrary, the logical date of a read set item is larger than the *rv* of the transaction, then a concurrent transaction has updated this item, invalidating the read. A transaction must abort if its read set is not valid. Once the read set it varified to be consistent the commit operation performs an increment-and-fetch on *GVC*, and stores the return value in local variable *wv* (which can be seen as a write version number or a version timestamp). Should the transaction commit, it will assign its *wv* as the new logical date of the shared variables in its write set. Finally the locks are realeased completing the commit operation.

A read operation must check both the lock and the logical time of the variable it is reading. If it is either locked, or the value of the logical time is greater then *rv* then the transaction must abort, otherwise the read is consistent and the value can be returned. Additionally, if the transaction is an update transaction then the variable is also added to the read set. Importantly, thanks to the use of the logical clocks the read operations take constant time and do not require validating the locations previously read in order to ensure opacity.

In order to implemnt the locks and logical time, they are stored in a shared array of words. Each shared memory word accessed by the STM protocol is mapped to a location in the lock array through a one-to-many hash function, resulting in one lock covering several shared memory locations. This partitions the memory into so-called stripes. For each memory word in the array the first bit acts as lock bit, indicating whether the lock is free or not. The rest of the bits form the logical time of the variables assoiated to that location by the hash function.

4.4 The Protocol

The following sections will describe the protocol.

4.4.1 Memory Set-up and Data Structures.

Memory Set-up. The underlying memory system is made up of atomic read/write registers. Moreover some of them can also be accessed by the the following two operations. The operation denoted $\mathsf{Fetch\&increment}()$ atomically adds one to the register and returns its previous value. The operation denoted $\mathsf{C\&S}()$ (for compare and swap) is a conditional write. $\mathsf{C\&S}(x,a,b)$ writes b into x iff $x=a$. In that case it returns *true*. Otherwise it returns *false*.

The proposed algorithm assumes that the variables are of types and values that can be stored in a memory word. This assumption aids in the clarity of the algorithm description but it is also justified by the fact that the algorithm extends TL2, an algorithm that is designed to be word-based.

As in TL2, the variable *GVC* acts as global clock which is incremented by update transactions. Apart from a global notion of "time", there exists also a local one; each process maintains a local variable denoted *time*, which is used in order to keep track of when, with respect to the *GVC*, a non-transactional operation or a transaction was last performed by the process. This variable is then used during non-transactional operations to ensure the (strict) serialization of operations is not violated.

As described in section 4.3, in TL2 a shared array of locks is maintained and each shared memory word is associated with a lock in this array by some function. Given this, a memory word directly contains the value of the variable that is stored in it. Instead, the algorithm presented here, uses a different memory set-up that does not require a lock array, but does require an extra level of indirection when loading and storing values in memory. Instead of storing the value of a variable directly to a memory word, each write operation on variable *var*, transactional or non-transactional, first creates an algorithm-specific structure that contains the new value of *var*, as well as necessary meta-data and second stores a pointer to this structure in the memory word. The memory set-up is illustrated in Fig. 4.2. Given the particular memory arrangement that the algorithm uses, pointers are used in order to load and store items from memory. [1]

T-record and NT-record. These algorithm-specific data structures are shared and can be of either two kinds, which will be referred to as T-records and NT-records. A T-record is created by a transactional write operation while an NT-record is created by a non-transactional write operation.

[1]The following notation is used. If pt is a pointer, $pt\downarrow$ is the object pointed to by pt. if aa is an object, $\uparrow aa$ is a pointer to aa. Hence $((\uparrow aa)\downarrow = aa$ and $\uparrow(pt\downarrow) = pt$.

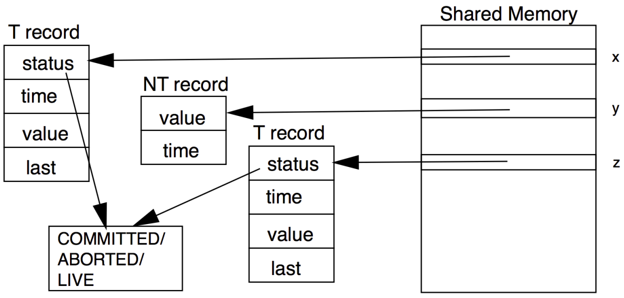

Figure 4.2: The memory set-up and the data structures that are used by the algorithm.

New T-records are created during the transactional write operations. Then during the commit operation the pointer stored at *addr* is updated to point to this new T-record. During NT-write operations new NT-records are created and the pointer at *addr* is updated to point to the records.

When a read operation - be it transactional or non-transactional - accesses a shared variable it cannot know beforehand what type of record it will find. Therefore, it can be seen in the algorithm listings, that whenever a record is accessed, the operation checks its type, i.e., it checks whether it is a T-record or an NT-record (for example, line 209 in Fig. 4.3 contains such a check. A T-record is "of type T", while an NT-record is "of type NT").

T-record. A T-record is a structure containing the following fields.

status This field indicates the state of the transaction that created the T-record. The state can either be LIVE, COMMITTED or ABORTED. The state is initially set to LIVE and is not set to COMMITTED until during the commit operation when all locations of the transaction's write set have been set to point to the transaction's T-records and the transaction has validated its read set. Since a transaction can write to multiple locations, the *status* field does not directly store the state, instead it contains a pointer to a memory location containing the state for the transaction. Therefore the *status* field of each T-record created by the same transaction will point to the same location. This ensures that any change to the transaction's state is immediately recognized at each record.

time The *time* field of a T-record contains the value of the *GVC* at the moment the record was inserted to memory. This is similar to the logical dates of TL2.

value This field contains the value that is meant to be written to the chosen memory location.

last During the commit operation, locations are updated to point to the committing transaction's T-records, overwriting the previous value that was stored in this location. Failed validation or concurrent non-transactional operations may cause this transaction to abort after it updates some memory locations, but before it fully commits. Due to this, the previous value of the location needs to be available for future reads. Instead of rolling back old memory values, the *last* field of a T-record is used, storing the previous value of this location.

NT-record. An NT-record is a structure containing the following fields.

value This field contains the value that is meant to be written to the chosen memory location.

time As in the case of T-records, the *time* field of NT-records also stores the value of the *GVC* when the write took place.

Due to this different memory structure a shared lock array is no longer needed, instead of locking each location in the write set during the commit operation, this algorithm performs a compare and swap directly on each memory location changing the address to point to one of its T-records. After a successful compare and swap and before the transactions status has been set to COMMITTED or ABORTED, the transaction effectively owns the lock on this location. Like in TL2, any concurrent transaction that reads the location and sees that it is locked ($status =$ LIVE) will abort itself.

Transactional Read and Write Sets. Like TL2, read only transactions do not use read sets while update transactions do. The read set is made up of a set of tuples for each location read, $\langle addr, value \rangle$ where *addr* is the address of the location read and *value* is the value. The write set is also made up of tuples for each location written by the transaction, $\langle addr, item \rangle$ where *addr* is the location to be written and *item* is a T-record for this location.

4.4.1.1 Discussion.

One advantage of the TL2 algorithm is in its memory layout. This is because reads and writes happen directly to memory (without indirection) and the main amount of additional memory that is used is in the lock array. Unfortunately this algorithm breaks that and requires an additional level of indirection as well as additional memory per location. While garbage collection will be required for old T- and NT-records, here we assume automatic garbage collection such as that provided in Java, but additional solutions will be explored in future work. These additional requirements can be an acceptable trade-off given that they are only needed for memory that will be shared between transactions. In the appendix of this paper we present two variations of the algorithm that trade off different memory schemes for different costs to the transactional and non-transactional operations.

4.4.2 Description of the Algorithm.

The main goal of the algorithm is to provide strong isolation in such a way that the non-transactional operations are never blocked or aborted. In order to achieve this, the algorithm delegates most of its concurrency control and consistency checks to the transactional code. Non-transactional operations access and modify memory locations without waiting for concurrent transactions and it is mainly up to transactions accessing the same location to deal with ensuring safe concurrency. As a result, this algorithm gives high priority to non-transactional code.

4.4.3 Non-transactional Operations.

Algorithm-specific read and write operations shown in Fig. 4.3 must be used when a shared variable is accessed accessed outside of a transaction. As described in section 4.1.2 this can be done by hand by the programmer, but more appropriately a programmer will write normal shared reads and writes which will then be automatically converted to non-transactional read (resp. write) operations by the compiler or dynamically at runtime.

```
operation non_transactional_read(addr) is
(208)   tmp ← (↓ addr);
(209)   if ( tmp is of type T ∧(↓ tmp.status) ≠ COMMITTED )
(210)      then if (tmp.time ≤ time ∧ (↓ tmp.status) = LIVE)
(211)               then C&S(tmp.status, LIVE, ABORTED) end if;
(212)            if ((↓ tmp.status) ≠ COMMITTED)
(213)               then value ← tmp.last
(214)               else value ← tmp.value
(215)            end if;
(216)      else value ← tmp.value
(217)   end if;
(218)   time ← max(time, tmp.time)
(219)   if (time = ∞) then time = GCV end if;
(220)   return (value)
end operation.

operation non_transactional_write(addr, value) is
(221)   allocate new variable next_write of type NT;
(222)   next_write ← (addr, value, ∞);
(223)   addr ← (↑ next_write)
(224)   time ← GVC;
(225)   next_write.time ← time;
end operation.
```

Figure 4.3: Non-transactional operations for reading and writing a variable.

Non-transactional Read. The operation non_transactional_read() is used to read, when not in a transaction, the value stored at *addr*. The operation first dereferences the pointer stored at *addr* (line 208). If the item is a T-record that was created by a transaction which has not yet committed then the *value* field cannot be immediately be read as the transaction might still abort. Also if the current process has read (or written to) a value that is more recent then the transaction (meaning the process's *time* field is greater or equal to the T-records *time*, line 210) then the transaction must be directed to abort (line 211) so that opacity and strong isolation (containment specifically) is not violated. From a T-record with a transaction that is not committed, the value from the *last* field is stored to a local variable (line 213) and will be returned on operation completion. Otherwise the *value* field of the T- or NT-record is used (line 214).

Next the process local variable *time* is advanced to the maximal value among its current value and the logical date of the T- or NT-record whose value was read. Finally if *time* was set to ∞ on line 218 (meaning the T- or NT-record had yet to set its *time*), then it is updated to the *GCV* on line 219. The updated *time* value is used to prevent consistency violations. Once these book-keeping operations are finished, the local variable *value* is returned (line 220).

Non-transactional Write. The operation non_transactional_write() is used to write to a shared variable *var* by non-transactional code. The operation takes as input the address of the shared variable as well as the value to be written to it. This operation creates a new NT-record (line 221), fills in its fields (line 222) and changes the pointer stored in *addr* so that it references the new record it has created (line 223). Unlike update transactions, non-transactional writes do not increment the global clock variable *GCV*. Instead they just read *GCV* and set the NT-record's time value as well as the process local *time* to the value read (line 224 and 225). Since the *GCV* is not incremented, several NT-records might have the same *time* value as some transaction. When such a situation is recognized where a live transaction has the same time value as an NT-record the transaction must be aborted (if recognized during an NT-read operation, line 211) or perform read set validation (if during a transactional read operation, line 230 of Fig. 4.4).

This is done in order to prevent consistency violations caused by the NT-writes not updating the *GCV*.

4.4.4 Transactional Read and Write Operations.

The transactional operations for performing reads and writes are presented in Fig. 4.4.

Transactional Read. The operation transactional_read() takes *addr* as input. It starts by checking whether the desired variable already exists in the transaction's write set, in which case the value stored there will be returned (line 226). If the variable is not contained in the write set, the pointer in *addr* is dereferenced (line 227) and set to *tmp*. Once this is detected to be a T- or NT-record some checks are then performed in order to ensure correctness.

In the case that *tmp* is a T-record the operation must check to see if the status of the transaction for this record is still LIVE and if it is the current transaction is aborted (line 236). This is similar to a transaction in TL2 aborting itself when a locked location is found. Next the T-record's *time* field is checked, and (similar to TL2) if it greater then the process's local *rv* value the transaction must abort (line 239) in order to prevent consistency violations. If this succeeds without aborting then the local variable *value* is set depending on the stats of the transaction that created the T-record (line 236-237).

In case *tmp* is an NT-record (line 228), the operation checks whether the value of the *time* field is greater or equal to the process local *rv* value. If it is, then this write has possibly occurred after the start of this transaction and there are several possibilities. In the case of an update transaction validation must be preformed, ensuring that none of the values it has read have been updated (line 230). In the case of a read only transaction, the transaction is aborted and restarted as an update transaction (line 231). It is restarted as an update transaction so that it has a read set that it can validate in case this situation occurs again. Finally local variable *value* is set to be the value of the *value* field of the *tmp* (line 233).

It should be noted that the reason why the checks are performed differently for NT-records and T-records is because the NT-write operations do not update the global clock value while update transaction do. This means that the checks must be more conservative in order to ensure correctness. If performing per value validation or restarting the transaction as an update transaction is found to be too expensive, a third possibility would be to just increment the global clock, then restart the transaction as normal.

Finally to finish the read operation, the $\langle addr, value\rangle$ is added to the read set if the transaction is an update transaction (line 241), and the value of the local variable *value* is returned.

Transactional Write. The transactional_write() operation takes *addr* as input value, as well as the value to be written to *var*. As TL2, the algorithm performs commit-time updates of the variables it writes to. For this reason, the transactional write operation simply creates a T-record and fills in some of its fields (lines 244 - 245) and adds it to the write set. However, in the case that a T-record corresponding to *addr* was already present in the write set, the *value* field of the corresponding T-record is simply updated (line 246).

Begin and End of a Transaction The operations that begin and end a transaction are begin_transaction() and try_to_commit(), presented in Fig. 4.5. Local variables necessary for transaction execution are initialized by begin_transaction(). This includes *rv* which is set

```
operation transactional_read(addr) is
  if addr ∈ ws then return (item.value from addr in ws) end if;
  tmp ← (↓ addr);
  if (tmp is of type NT)
    then if (tmp.time >= rv)
           then if this is an update transaction then validate_by_value()
                  else abort() and restart as an update transaction end if;
           end if;
           value ← tmp.value;
    else
           if ((status ← (↓ tmp.status)) ≠ COMMITTED )
             then if (status = LIVE) then abort() else value ← tmp.last end if;
             else value ← tmp.value
           end if;
           if (tmp.time > rv) then abort() end if;
  end if;
  if this is an update transaction then add ⟨addr, value⟩ to rs end if;
  return (value)
end operation.

operation transactional_write(addr, value) is
  if addr ∉ ws
    then allocate a new variable item of type T;
           item ← (value, (↑ status), ∞); ws ← ws ∪ ⟨addr, item⟩;
    else  set item.value with addr in ws to value
  end if;
end operation.
```

Figure 4.4: Transactional operations for reading and writing a variable.

to *GCV* and, like in TL2, is used during transactional reads to ensure correctness, as well as *status* which is set to LIVE and the read and write sets which are initialized as empty sets. (lines 248-250).

After performing all required read and write operations, a transaction tries to commit, using the operation try_to_commit(). Similar to TL2, a try_to_commit() operation starts by trivially committing if the transaction was a read-only one (line 251) while an update transaction must announce to concurrent operations what locations it will be updating (the items in the write set). However, the algorithm differs here from TL2, given that it is faced with concurrent non-transactional operations that do not rely on locks and never block. This implies that even after acquiring the locks for all items in its write set, a transaction could be "outrun" by a non-transactional operation that writes to one of those items causing the transaction to be required to abort in order to ensure correctness. As described previously, while TL2 locks items in its write set using a lock array, this algorithm compare and swaps pointers directly to the T-records in its write set (lines 252-260) while keeping a reference to the previous value. The previous value is stored in the T-record before the compare and swap is performed (lines 255-256) with a failed compare and swap resulting in the abort of the transaction. If while performing these compare and swaps the transaction notices that another LIVE transaction is updating this memory, it aborts itself (line 255). By using these T-records instead of locks concurrent operations have access to necessary metadata used to ensure correctness.

The operation then advances the *GVC*, taking the new value of the clock as the logical time for this transaction (line 261). Following this, the read set of the transaction is validated for correctness (line 261). Once validation has been performed the operation must ensure that non of its writes have been concurrently overwritten by non-transactional operations (lines 262-265) if so then the transaction must abort in order to (line 264) to ensure consistency. During this

```
operation begin_transaction() is
(248)  determine whether transaction is update transaction based on compiler/user input
(249)  rv ← GVC; Allocate new variable status;
(250)  status ←LIVE;  ws ← ∅; rs ← ∅
end operation.

operation try_to_commit() is
(251)  if (ws = ∅) then return (COMMITTED) end if;
(252)  for each (⟨addr, item⟩ ∈ ws) do
(253)     tmp ← (↓ addr);
(254)     if (tmp is of type T ∧ (status ← (↓ tmp.status)) ≠ COMMITTED )
(255)            then if (status = LIVE) then abort() else item.last ← tmp.last end if;
(256)            else item.last ← tmp.value
(257)     end if;
(258)     item.time ← tmp.time;
(259)     if (¬C&S(addr, tmp, item)) then abort() end if;
(260)  end for;
(261)  time ← fetch&increment(GVC); validate_by_value();
(262)  for each (⟨addr, item⟩ ∈ ws) do
(263)     item.time ← time;
(264)     if (item ≠ (↓ addr)) then abort() end if;
(265)  end for;
(266)  if C&S(status, LIVE, COMMITTED)
(267)     then return (COMMITTED)
(268)     else abort()
(269)  end if;
end operation.
```

Figure 4.5: Transaction begin/commit.

check the transaction updates the *time* value of its T-records to the transactions logical time (line 263) similar to the way TL2 stores time values in the lock array so that future operations will know the serialization of this transaction's updates.

Finally the transaction can mark its updates as valid by changing its *status* variable from LIVE to COMMITTED (line 266). This is done using a compare and swap as there could be a concurrent non-transactional operations trying to abort the transaction. If this succeeds then the transaction has successfully committed, otherwise it must abort and restart.

Transactional Helping Operations. Apart from the basic operations for starting, committing, reading and writing, a transaction makes use of helper operations to perform aborts and validate the read set. Pseudo-code for this kind of helper operations is given in Fig. 4.6.

Operation validate_by_value() is an operation that performs validation of the read set of a transaction. Validation fails if any location in *rs* is currently being updated by another transaction (line 274) or has had its changed since it was first read by the transaction (line 278) otherwise it succeeds. The transaction is immediately aborted if validation fails (lines 274, 278). Before the validation is performed the local variable *rv* is updated to be the current value of *GVC* (line 270). This is done because if validation succeeds then transaction is valid at this time with a larger clock value possibly preventing future validations and aborts.

When a transaction is aborted in the present algorithm, the status of the current transaction is set to ABORTED (line 280) and it is immediately restarted as a new transaction.

4.5 Proof of correctness

[NOTE!!!!: Need to do this]

```
operation validate_by_value() is
    rv ← GVC;
    for each ⟨addr, value⟩ in rs do
        tmp ← (↓ addr);
        if (tmp is of type T ∧tmp.status ≠ COMMITTED)
                then if (tmp.status = LIVE ∧item ∉ ws) then abort() end if;
                    new_value ← tmp.last;
                else new_value ← tmp.value
        end if;
        if new_value ≠ value then abort() end if;
    end for;
end operation.

operation abort() is
    status ← ABORTED;
    the transaction is aborted and restarted
end operation.
```

Figure 4.6: Transactional helper operations.

4.6 Conclusion

This paper has presented an algorithm that achieves non-blocking strong isolation "on top of" a TM algorithm based on logical dates and locks, namely TL2. In the case of a conflict between a transactional and a non-transactional operation, this algorithm gives priority to the non-transactional operation, with the reasoning that while an eventual abort or restart is part of the specification of a transaction, this is not the case for a single shared read or write operation. Due to this priority mechanism, the proposed algorithm is particularly appropriate for environments in which processes do not rely heavily on the use of especially large transactions along with non-transactional write operations. In such environments, terminating strong isolation is provided for transactions, while conventional read and write operations execute with a small additional overhead.

4.7 Version of algorithm that does not use NT-records

This algorithm also provides wait-free NT read and write operations. The difference is that NT-records are not used. Instead NT values are read and written directly from memory. By doing this, memory allocations are not needed in NT writes and NT reads have one less level of indirection.

The cost of this is more frequent validations required in transactions when conflicts with NT writes occur. This algorithm is shown in Figs. 4.7-4.9.

4.8 Version of algorithm with non-blocking NT-reads and blocking NT-writes

This algorithm allows wait-free NT read operations. The only change that is needed to the base TL2 algorithm is that when an item is locked it points to the write-set of the transaction, and that each transaction has a marker that is initialized as *LIVE* and is set to *COMMITTED* just before the transaction starts performing write backs during the commit phase. The NT-read operation

```
operation non_transactional_read(addr) is
    tmp ← (↓ addr);
    if ( tmp is of type T )
       then if (tmp.status = LIVE)
                then C&S(tmp.status, LIVE, ABORTED)
             end if;
             if (tmp.status = ABORTED)
                then value ← tmp.last
                else value ← tmp.value
             end if;
       else value ← tmp
    end if;
    return (value)
end operation.

operation non_transactional_write(addr, value) is
    addr ← (↑ unMark(value)) end operation.
```

Figure 4.7: Non-transactional operations for reading and writing a variable.

is shown in Fig. 4.10.

operation transactional_read($addr$) **is**
  if $addr \in ws$ **then** return ($item.value$ from $addr$ in ws) **end if**;
  $tmp \leftarrow (\downarrow addr)$;
  if (tmp is of type T)
    then if ($status =$ LIVE) **then** abort() **end if**;
      if ($tmp.time > rv$) **then** abort() **end if**;
      if ($status =$ COMMITTED)
        then $value \leftarrow tmp.val$
        else $value \leftarrow tmp.last$
      end if;
    else
  % Do validation to prevent abort due to a non-transactional write
      $rv \leftarrow$ validate_by_value();
      $value \leftarrow tmp$;
  end if;
  if this is an update transaction **then** add $value$ to rs **end if**;
  return ($value$)
end operation.

operation transactional_write($addr, value$) **is**
  if $addr \notin ws$
    then allocate a new variable $item$ of type T;
      $item \leftarrow (addr, value, status, \infty)$; $ws \leftarrow ws \cup item$
    else set $item.value$ with $addr$ in ws to $value$
  end if
end operation.

Figure 4.8: Transactional operations for reading and writing a variable.

```
operation try_to_commit() is
    if (ws = ∅) then return (COMMITTED) end if;
    for each (item ∈ ws) do
        tmp ← (↓ addr);
        if (tmp is of type T)
                then if ((status ← tmp.status) = COMMITTED)
                    then item.last ← tmp.value
                    else if (status = ABORTED) then item.last ← tmp.last
                    else abort()
                  end if;
                else item.last ← tmp
        end if;
        if (¬C&S(item.addr, tmp, item)) then abort() end if;
    end for;
    time ← fetch&increment(GVC);
    validate_by_value();
    % Ensure the writes haven't been overwritten by non-transactional writes
    for each (item ∈ ws) do
        if (item ≠ (↓ item.addr)) then abort() end if
        item.time ← time;
    end for;
    if (C&S(status, LIVE, COMMITTED))
        then return (COMMITTED)
        else abort()
    end if;
end operation.
```

Figure 4.9: Transaction commit.

```
operation non_transactional_read(addr) is
    lock ← load_lock(addr);
    value ← (↓ addr);
    if ( lock is locked ∧tmp.status = COMMITTED ∧addr ∈ lock.ws)
        then value ← item.value from addr in lock.ws
    end if;
    return (value)
end operation.

operation non_transactional_write(addr, value) is
    Perform a transactional begin/write/commit operation
end operation.
```

Figure 4.10: Non-transactional operations for reading and writing a variable.

Conclusion

Bibliography

[1] Attiya H. and Hillel E., Single-version STM Can be Multi-version Permissive. *Proc. 12th Int'l Conference on Distributed Computing and Networking (ICDCN'11)*, Springer-Verlag, LNCS #6522, pp. 83-94, 2011.

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

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

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

[5] Crain T., Imbs D. and Raynal M., Read invisibility, virtual world consistency and permissiveness are compatible. *Tech Report #1958*, IRISA, Univ. de Rennes 1, France, November 2010.

[6] 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.

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

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

[9] 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.

[10] 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.

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

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

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

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

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

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

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

[19] Virendra J. Marathe, Michael F. Spear, Christopher Heriot, Athul Acharya, David Eisenstat, William N. Scherer Iii, and Michael L. Scott. Lowering the overhead of nonblocking software transactional memory. In *Dept. of Computer Science, Univ. of Rochester*, 2006.

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

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

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

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

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

[25] Afek Y., Dauber D. and Touitou D., Wai-free made Fast. *Proc. 7th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, ACM Press, pp. 538-547, 1995.

[26] Anderson J. and Moir M., Universal Constructions for Large Objects. *IEEE Transactions on Parallel and Distributed Systems*, 10(12):1317-1332, 1999.

[27] 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.

[28] 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.

[29] 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.

[30] Chuong Ph., Ellen F. and Ramachandran V., A Universal Construction for Wait-free Transaction Friendly Data Structures. *Proc. 22th Int'l ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'10)*, ACM Press, pp. 335-344, 2010.

[31] Crain T., Imbs D. and Raynal M., Towards a universal construction for transaction-based multiprocess programs. *Proceedings of the 13th International Conference on Distributed Computing and Networking (ICDCN 2012)*, Springer-Verlag LNCS, 2012.

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

[33] 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.

[34] Fatourou P. and Kallimanis N., The Red-blue Adaptive Universal Construction. *Proc. 22nd Int'l Symposium on Distributed Computing (DISC '09)*, Springer-Verlag, LNCS#5805, pp. 127-141, 2009.

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

[36] 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.

[37] Frølund S. and Guerraoui R., X-Ability: a Theory of Replication. *Distributed Computing*, 14(4):231-249, 2001.

[38] 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.

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

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

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

[42] 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.

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

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

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

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

[47] 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.

[48] 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.

!!!

[49] 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.

[50] 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.

[51] 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.

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

[53] 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.

[54] 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.

[55] G. Adelson-Velskii and E. M. Landis. An algorithm for the organization of information. In *Proc. of the USSR Academy of Sciences*, volume 146, pages 263–266, 1962.

[56] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *Proc. of the 14th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2009.

[57] Lucia Ballard. Conflict avoidance: Data structures in transactional memory, May 2006. Undergraduate thesis, Brown University.

[58] Rudolf Bayer. Symmetric binary b-trees: Data structure and maintenance algorithms. *Acta Informatica 1*, 1(4):290–306, 1972.

[59] Luc Bougé, Joaquim Gabarro, Xavier Messeguer, and Nicolas Schabanel. Height-relaxed AVL rebalancing: A unified, fine-grained approach to concurrent dictionaries, 1998. Research Report 1998-18, ENS Lyon.

[60] Nathan G. Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. A practical concurrent binary search tree. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[61] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *Proc. of The IEEE Int'l Symp. on Workload Characterization*, 2008.

[62] Christopher Cole and Maurice Herlihy. Snapshots and software transactional memory. *Sci. Comput. Program.*, 58(3):310–324, 2005.

[63] Luke Dalessandro, Michael Spear, and Michael L. Scott. NOrec: streamlining STM by abolishing ownership records. In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[64] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, and E. F. M. Steffens. On-the-fly garbage collection: an exercise in cooperation. *Commun. ACM*, 21(11):966–975, 1978.

[65] Aleksandar Dragojevic, Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Why STM can be more than a research toy. *Commun. ACM*, 54(4):70–77, 2011.

[66] Pascal Felber, Vincent Gramoli, and Rachid Guerraoui. Elastic transactions. In *Proc. of the 23rd Int'l Symp. on Distributed Computing*, 2009.

[67] Vincent Gramoli and Rachid Guerraoui. Democratizing transactional programming. In *Proc. of the ACM/IFIP/USENIX 12th Int'l Middleware Conference*, 2011.

[68] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th Annual Symp. on Foundations of Computer Science*, 1978.

[69] Tim Harris, Simon Marlow, Simon Peyton-Jones, and Maurice Herlihy. Composable memory transactions. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2005.

[70] Maurice Herlihy and Eric Koskinen. Transactional boosting: A methodology for highly-concurrent transactional objects. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2008.

[71] Intel Corporation. Intel transactional memory compiler and runtime application binary interface, May 2009.

[72] J. L. W. Kessels. On-the-fly optimization of data structures. *Comm. ACM*, 26:895–901, 1983.

[73] Udi Manbar and Richard E. Ladner. Concurrency control in a dynamic search structure. *ACM Trans. Database Syst.*, 9(3):439–455, 1984.

[74] C. Mohan. Commit-LSN: a novel and simple method for reducing locking and latching in transaction processing systems. In *Proc. of the 16th Int'l Conference on Very Large Data Bases*, 1990.

[75] J. Eliot B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, 2006.

[76] Yang Ni, Vijay Menon, Ali-Reza Abd-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proc. of the 12th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2007.

[77] O. Nurmi and E. Soisalon-Soininen. Uncoupling updating and rebalancing in chromatic binary search trees. In *Proc. of the 10th ACM Symp. on Principles of Database Systems*, 1991.

[78] O. Nurmi, E. Soisalon-Soininen, and D. Wood. Concurrency control in database structures with relaxed balance. In *Proc. of the 6th ACM Symp. on Principles of Database Systems*, 1987.

[79] Victor Pankratius and Ali-Reza Adl-Tabatabai. A study of transactional memory vs. locks in practice. In *Proc. of the 23rd ACM Symp. on Parallelism in Algorithms and Architectures*, 2011.

[80] Christopher J. Rossbach, Owen S. Hofmann, and Emmett Witchel. Is transactional programming actually easier? In *Proc. of the 15th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2010.

[81] Nir Shavit. Data structures in the multicore age. *Commun. ACM*, 54(3):76–84, 2011.

[82] Richard M. Yoo, Yang Ni, Adam Welc, Bratin Saha, Ali-Reza Adl-Tabatabai, and Hsien-Hsin S. Lee. Kicking the tires of software transactional memory: why the going gets tough. In *Proc. of the 20th ACM Symp. on Parallelism in Algorithms and Architectures*, 2008.

[83] S. Borkar. Thousand core chips: a technology perspective. In *DAC*, pages 746–749, 2007.

[84] T. Crain, V. Gramoli, and M. Raynal. A speculation-friendly binary search tree. In *Proc. of the 17th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, 2012.

[85] C. Dwork, M. Herlihy, and O. Waarts. Contention in shared memory algorithms. *J. ACM*, 44:779–805, November 1997.

[86] F. Ellen, P. Fatourou, E. Ruppert, and F. van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 131–140, New York, NY, USA, 2010. ACM.

[87] M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *Proceedings of the Second Symposium on Operating System Design and Implementation*, pages 123–136, 1996.

[88] M. Fomitchev and E. Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, PODC '04, pages 50–59, New York, NY, USA, 2004. ACM.

[89] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University, September 2003.

[90] M. Frigo, C. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the 40th Annual Symposium on Foundations of Computer Science*, pages 285 –297, 1999.

[91] T. Harris. A pragmatic implementation of non-blocking linked-lists. In *DISC*, pages 300–314, 2001.

[92] M. Herlihy, Y. Lev, V. Luchangco, and N. Shavit. A simple optimistic skiplist algorithm. In *Proceedings of the 14th international conference on Structural information and communication complexity*, SIROCCO'07, pages 124–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[93] G. Korland, N. Shavit, and P. Felber. Deuce: Noninvasive software transactional memory. *Transactions on HiPEAC*, 5(2), 2010.

[94] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe. The 48-core SCC processor: the programmer's view. In *SC*, pages 1–11, 2010.

[95] M. M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *SPAA*, pages 73–82, 2002.

[96] W. Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM*, 33, June 1990.

[97] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In *DISC*, 2006.

[98] H. Sutter. Choose concurrency-friendly data structures. *Dr. Dobb's Journal*, June 2008.

[99] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Computing Systems*, 10:99–127, 1976. 10.1007/BF01683268.

[100] D. Wentzlaff, P. Griffin, H. Hoffmann, L. Bao, B. Edwards, C. Ramey, M. Mattina, C.-C. Miao, J. Brown, and A. Agarwal. On-chip interconnection architecture of the tile processor. *IEEE Micro*, 27(5):15–31, 2007.

[101] D. L. Detlefs, P. A. Martin, M. Moir, and G. L. Steele, Jr. Lock-free reference counting. In *PODC*, pages 190–199, 2001.

[102] D. Lea. Jsr-166 specification request group.

[103] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *SAC*, pages 1438–1445, 2004.

[104] G. Taubenfeld. Contention-sensitive data structures and algorithms. In *DISC*, pages 157–171, 2009.

[105] J. D. Valois. *Lock-free data structures*. PhD thesis, Rensselaer Polytechnic Institute, 1996.

[106] Afek, Y., Avni, H., Dice, D., Shavit, N.: Efficient lock free privatization. In: *Proc. 14th Int'l conference on Principles of Distributed Systems (OPODIS'10)*, pp. 333–347, Springer-Verlag, LNCS #6490 (2010)

[107] Dalessandro, L., Scott, M.: Strong Isolation is a Weak Idea. In: *Proc. Workshop on transactional memory (TRANSACT'09)* (2009)

[108] Dice, D., Matveev, A., Shavit, N.: Implicit privatization using private transactions. In: *Proc. Workshop on transactional memory (TRANSACT'10)* (2010)

[109] Harris, T., Larus, J., Rajwar, R.: Transactional Memory, *2nd edition, Synthesis Lectures on Computer Architecture*, Morgan & Claypool Publishers (2006)

[110] Maessen, J.-W., Arvind, M.: Store Atomicity for Transactional Memory. *Electronic Notes on Theoretical Computer Science*, 174(9):117–137 (2007).

[111] Martin, M., Blundell, C., Lewis, E.: Subtleties of Transactional Memory Atomicity Semantics. *IEEE Computer Architecture Letters*, 5(2): (2006)

[112] Matveev, A., Shavit, N.: Towards a Fully Pessimistic STM Model. In: *Proc. Workshop on transactional memory (TRANSACT'12)* (2012)

[113] Minh, C., Trautmann, M., Chung, J., McDonald, A., Bronson, N., Casper, J., Kozyrakis, C., Olukotun, K.: An effective hybrid transactional memory system with strong isolation guarantees. In: *SIGARCH Comput. Archit. News*, 35(2):69–80 (2007)

[114] Schneider, F., Menon, V., Shpeisman, T., Adl-Tabatabai, A.: Dynamic optimization for efficient strong atomicity. In: *ACM SIGPLAN Noticers*, 43(10):181–194 (2008)

[115] Scott, M.L., Spear, M.F., Dalessandro, L., Marathe, V.J.: Delaunay Triangulation with Transactions and Barriers. In: *Proc. 10th IEEE Int'l Symposium on Workload Characterization (IISWC '07)*, IEEE Computer Society, pp. 107–113 (2007)

[116] Shpeisman, T., Menon, V., Adl-Tabatabai, A.R., Balensiefer, S., Grossman, D., Hudson, R.L., Moore, K.F., Saha, B.: Enforcing isolation and ordering in STM. In: *ACM SIGPLAN Noticers*, 42(6):78–88 (2007)

[117] Spear M.F., Dalessandro L., Marathe V.J., Scott, M.L.: Ordering-Based Semantics for Software Transactional Memory. In: *Proc 12th Int'l Conf. on Principles of Distributed Systems (OPODIS '08)*, Springer-Verlag LNCS #5401, pp. 275–294 (2008)

[118] Spear, M.F., Marathe, V.J., Dalessandro, L., Scott, M.L.: Privatization techniques for software transactional memory. In: *Proc. 26th annual ACM symposium on Principles of Distributed Computing (PODC '07)*, . ACM press, pp. 338–339, (2007)

[119] Michael F. Spear, Virendra J. Marathe, Luke Dalessandro, and Michael L. Scott. Privatization techniques for software transactional memory. In *Proceedings of the 26th PODC ACM Symposium on Principles of Distributed Computing*. Aug 2007.

[120] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. Compiler and runtime support for efficient software transactional memory. In *PLDI '06: Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation*, pages 26–37, New York, NY, USA, 2006. ACM.

[121] Lorenzo Alvisi. Lock-free serializable transactions. Technical report, 2005.

[122] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir. Transactional contention management as a non-clairvoyant scheduling problem. In *PODC '06: Proceedings of the twenty-fifth annual ACM symposium on Principles of distributed computing*, pages 308–315, New York, NY, USA, 2006. ACM.

[123] Hagit Attiya, Rachid Guerraoui, Danny Hendler, and Petr Kuznetsov. The complexity of obstruction-free implementations. *J. ACM*, 56(4):1–33, 2009.

[124] Hagit Attiya, Eshcar Hillel, and Alessia Milani. Inherent limitations on disjoint-access parallel implementations of transactional memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 69–78, New York, NY, USA, 2009. ACM.

[125] Robert Ennals and Robert Ennals. Efficient software transactional memory. Technical report, 2005.

[126] Robert Ennals and Robert Ennals. Software transactional memory should not be obstruction-free. Technical report, 2006.

[127] Keir Fraser and Tim Harris. Concurrent programming without locks. *ACM Trans. Comput. Syst.*, 25, May 2007.

[128] Vincent Gramoli, Derin Harmanci, and Pascal Felber. On the Input Acceptance of Transactional Memory. *Parallel Processing Letters*, 2009.

[129] R. Guerraoui, M. Herlihy, and B. Pochon. Polymorphic Contention Management. In *Proceedings of the 19th International Symposium on Distributed Computing (DISC'05)*, volume 3724 of *Lecture Notes in Computer Science*, pages 303–323, 2005.

[130] Damien Imbs and Michel Raynal. Software transactional memories: An approach for multicore programming. In *PaCT '09: Proceedings of the 10th International Conference on Parallel Computing Technologies*, pages 26–40, Berlin, Heidelberg, 2009. Springer-Verlag.

[131] Simon P. Jones. *Beautiful Concurrency*. O'Reilly Media, Inc., 2007.

[132] Idit Keidar and Dmitri Perelman. On avoiding spare aborts in transactional memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 59–68, New York, NY, USA, 2009. ACM.

[133] Yossi Lev, Victor Luchangco, Virendra Marathe, Mark Moir, Dan Nussbaum, and Marek Olszewski. Anatomy of a scalable software transactional memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*, feb 2009.

[134] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles Muller. Scheduling support for transactional memory contention management. In *PPoPP '10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 79–90, New York, NY, USA, 2010. ACM.

[135] Virendra J. Marathe, William N. Scherer Iii, and Michael L. Scott. Adaptive software transactional memory. In *In Proc. of the 19th Intl. Symp. on Distributed Computing*, pages 354–368, 2005.

[136] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197, New York, NY, USA, 2006. ACM.

[137] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[138] Michael L. Scott. Sequential specification of transactional memory semantics. In *ACM SIGPLAN Workshop on Transactional Computing*. Jun 2006. Held in conjunction with PLDI 2006.

[139] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. A comprehensive strategy for contention management in software transactional memory. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 141–150, New York, NY, USA, 2009. ACM.

List of Publications

International Conferences Articles

Technical Reports