N^o d'ordre : ANNÉE 2012

UNIVERSITÉ DE RENNES 1 ueb

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 :

Improving the Ease of Use of the Software Transactional Memory Abstraction

Abstract

[NOTE!!!!: Update abstract] Writing concurrent programs is well know difficult task and because if this there exists many abstractions designed with the intetion to make concurrent programming easier. Traditionally locks have been used to write concurrent programs, unfortunately it is difficult to write correct concurrent programs using locks that perform efficiently. In addition to this, locks present other problems such as scalability issues. Transactional memory has been proposed as a possible promising solution to these difficulties of writing concurrent programs. Using transactions can be viewed as a different methodology for concurrent programming, allowing the programmer the ability to declare what sections of his code should be executed atomically, without having to worry about synchronization details. Unfortunately transactional memory is know to suffer from performance and ease of use problems. This thesis specifically focuses on the ease of use problem by discussing how previous research has coped with these problems so far and proposes some ways in which we can address some unanswered problems.

Even though the idea of a transaction is clear, how the programmer should interact with the abstraction is still not clearly defined. The first chapter presents an introduction to this problem while following four chapters look at different approces working towards making transactional memory easier to use, organized as follows:

1. The second chapter presents an STM protocol that satisfies two properties that can be considered good for efficiency.

2. The following chapter introduces an STM protocol that guarantees transactions commit exactly once with the goal of hiding the concept of aborts to the programmer.

3. The fourth chapter discusses an STM protocol that allows safe concurrent access to the same memory both inside and outside of transactions.

4. The fifth chapter discusses adding efficient libraries specifically designed for use by programmers within STM.

The thesis finishes with a chapter detailing the conclusion and future work.

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. 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.

1.1 Lock-based concurrent programming

It is well known that the design of a concurrent program is not an easy task. Given this, base synchronization objects have been defined to help the programmer solve concurrency and process cooperation issues. A major milestone in this area, which was introduced more than forty years ago was the concept of *mutual exclusion* [32] that has given rise to the notion of a *lock* object. A lock object provides the programmer 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. 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] giving the programmer additional operations. The type of synchronization admitted by locks is often referred to as *pessimistic*, as each access to some location x blocks further accesses to x until the location is released. Unsurprisingly given that a person's thought process happens sequentially and that this concept of mutual exclusion is a straightforward way to conceive of synchronization/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. In a bad case all threads except the one owning the lock could be blocked waiting for the lock to be released, resulting in no better then sequential performance. 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. Using locks in such a way is often referred to as "course-grained".

Of course in order to improve the performance of course-grained locking the first solution that comes to mind would be to simply use a finer grain. 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 process progressing. In order to avoid deadlock locks are often acquired 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 same lock (sometimes called *priority inversion*. Further problems can occur if a thread crashes or is stalled while holding a lock.

Another important drawback associated with locks lie in the fact that lock-based operation cannot be easily reused [69, 67]. Consider the queue example, a programmer using a lock based queue might want to create a new operation that inserts two items in the queue atomically. If the queue is implemented using fine grained locking then creating this operation would likely require the programmer to not only have extensive knowledge of the current queue implementation, but to also have to make modifications to the entire implementation. If a single lock is used for the entire queue object then this operation can be implemented easily, but any performance gain due to concurrency is lost. This process of reusing operations is often referred to as composition.

1.2 Alternatives to locks

Given that programming using locks is no simple task we must ask the question: how to ease the job of the programmer to write concurrent applications? This is neiher a simple question to answer nor a question that has one correct answer. Due to this much research has been done from providing the programmer with access to additional efficient low level hardware operations, to providing high level abstractions, to completely changing the model of programming. We will very briefly mention some popular examples before introducing the solution that is exaimned in this thesis (namely software transactional memory (STM)).

Non-blocking concurrent programming Non-blocking algorithms [87] have been introduced as an alternative to using locks in order to avoid some of the scalability and progress problems of locks. These algorithms are implemented using powerful system level synchronization operations such as *compare_and_swap*. There have been many efficient and scalable non-blocking data structures proposed [95, 103, 105, 88, 89]. Unfortunately such algorithms are known to be extremely difficult to implement and understand. Simply implementing a concurrent non-blocking version of a sequential data-structure has been enough to publish papers in top research publications, showing that non-blocking algorithms can be very difficult to get correct. As a result, non-blocking operations can help overcome some of the specific problems of locks but make no considerations about the difficulties of writing correct concurrent programs.

Libraries A different partial solution to making concurrent programming easier consists of providing the programmer with an appropriate library where 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 his particular synchronization issues, even simply composing several operations of the provided implementation to create a new operation is usually not possible using these libraries.

[NOTE!!!!: Should mention more things here???]

1.3 The Software Transactional Memory approach

The concept of *Software Transactional Memory* (STM) is a possible answer to the challenge of concurrent programming. Before describing the details, let us first consider a "spirit/design philosophy" that has helped give rise to STM systems: the notion of *abstraction level*. More precisely, the aim of an increased abstraction level is to allow the programmer to focus and concentrate only on the problem 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. In this manner STM can be seen as a new concept that takes up this challenge when considering synchronization issues.

The way transactional memory abstracts away the complexity associated with concurrent programming is 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 synchronization 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 middle-ware approach that provides the programmer 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 The programmer then must 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. Therefore the programmer is not concerned by synchronization except when 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 using low level synchronization operations such as `compare&swap` or even other abstractions such as locks (note that all these details are hidden from the programmer as he only has access to the interface to the STM which allows him to define atomic computation units).

Another important advantage of using transactional memory over locks it that a transactional program can be directly reused by another programmer within his own code. Hence a programmer composing operations from a transactional library into another transaction is guaranteed to obtain new deadlock-free operations that execute atomically. Further promoting the ease of use of transactions, several studies [79, 80] have been performed that find that (under the parameters

of their studies) users can create concurrent programs easier when using transactional memory instead of locks.

1.4 Details

The notion of transactional memory was first proposed nearly twenty years ago by Herlihy and Moss as an abstraction to be implemented in hardware and be used in order to easily implement lock-free concurrent data structures [12]. It has since been first implemented in software by Shavit and Touitou [23] and, partially thanks to the multi-core revolution, has recently gained great momentum as a promising alternative to locks in concurrent programming [7, 10, 46, 49]. Especially in the research community transactional memory has been a very hot topic with hundreds of papers being published, this section will give a very brief overview of some of that research in order to help explain STM.

1.4.1 Programming

As an abstraction designed to make concurrent programming easier, transactional memory needs a simple and precise interface for programmers to use. In order to actually define transactions in code, the most common approach is to surround the code by some keywords that indicate the beginning and end of a transaction. For example the programmer might just enclose his transaction using the *atomic* keyword:

$$atomic\{\ldots\}$$

The code within this block will then be treated as a transaction and appear to be executed atomically. In an ideal world this would be all that a programmer would have to know before starting to use transactional memory, but unfortunately as will be shown in this thesis, it is much more complex than this and there are many other things that the programmer must consider.

1.4.2 Memory access

Within a transaction a programmer will perform reads and writes to memory. Certain of these reads and writes will be to shared memory that is also visible to other transactions being executed by other processes in the system. In this case these memory accesses must be synchronized by the STM protocol. Each of these reads perform a transactional_read operation with each writes performing a transactional_write operation. These operations are defined by the STM protocol and usually require executing several lines of code. In addition to shared accesses, some memory accesses within a transaction might be to local memory (memory that is only accessible by the current process), as such this memory does not need to be synchronized (although it needs to be reinitialized when a transaction is restarted after an abort [aborts are described in the following section]) and can therefore be performed more efficiently then shared accesses.

Deciding which accesses should be treated as shared and which others should be treated as local can either be done by the programmer or done automatically by the system. Of course a solution that requires the programmer to define this makes using an STM more complicated, but on the other hand determining them automatically can lead to more accesses being declared as shared then necessary causing increased overhead during execution. Doing this efficiently is no easy task, [65] discusses some of the difficulties with STM code complication. and in [120]

an efficient STM is designed with optimizations for their JIT complier to lower the overhead of shared memory accesses.

A further complication with shared memory accesses is defining what happens when memory is accessed both inside and outside of transactions, i.e. transactionally and non-transactionally. There have been several proposals detailing different ways to deal with this such as *strong* and *weak isolation* [116] and *privatization* [119]. Chapter 4 looks at the problem of memory access in more detail.

1.4.3 Abort/Commit

For an STM protocol, 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 referred to as *optimistic synchronization*), but unfortunately, 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 in order to ensure safety there is an *abort/commit* notion associated with transactions. 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). From a programming point of view, an aborted transaction has no effect. Usually it is then up to the process that issued an aborted transaction to re-issue it or not; a transaction that is restarted is considered a new transaction. Unfortunately this does not mean that the programmer can be ignorant to the concept of transaction aborts. Most STM systems require the programmer to at least be aware of aborts, if not responsible to take some action to certain cases of aborts. The problem of aborted transactions and how a programmer must deal with them is discussed in detail in chapter 3.

1.4.4 Correctness

Without a proper consistency or correctness criterion the programmer is lost, these criterion must ensure that transactions execute atomically and prevents the programmer from having to worry about inconsistencies that might occur otherwise. STM consistency criterion define when a transaction can be committed or when it must abort. Interestingly, choosing a consistency criterion for memory transactions is different than from database transactions and even different then other concurrent objects, for example a common consistency criterion for database transactions is *serializability* [20]. Serializability guarantees that every committed transaction happened atomically at some point in time, as if they were executed sequentially. The real time order of execution does not need to be ensured, a transaction that took place long after a previous transaction committed can be ordered before the other, as long as the ordering creates a valid sequential execution. *Lineraizability* [13], the most common criterion used for concurrent objects, is stronger than serializabilty by adding the condition that transactions must be ordered according on their occurrence in real time. Notice how these conditions only relate to committed transactions, in a database system a transaction can run without causing harm in the system even if it accesses an *inconsistent set* of data. In transactional memory this is not always the case. An inconsistent view of the memory or set of data is one that could not have been created

by previous committed transactions, meaning the transaction must abort. **[NOTE!!!!: Should include this example??]** Consider the following example. There are two transactions A, and B, and three shared variables x, y, and z all initialized to 0. Transaction A performs the following operations, first it reads x, then it reads y, then it sets z to the value of $\frac{y}{x}$, then it tries to commit. Transaction B writes the value 1 to x, writes the value 1 to y, then tries to commit. It is obvious that any transaction reading the consistent values of x and y will read them as either both 0 or both 1. Now consider the following execution pattern, where transaction B commits successfully:

$$start_B, start_A, read_{A_X}, write_{B_X}, write_{B_Y}, commit_B, read_{A_Y}, commit_A$$

Now transaction A will access an inconsistent state of the data, reading x as 0 (because x was read before B committed) and y as 1 (because y was read after B committed). Since A now has an inconsistent view of the memory, when it performs the division $\frac{y}{x}$ a divide by 0 exception is created possibly resulting in undesirable behavior such as crashing the program. Divide by zero exceptions are not the only possible undesirable behaviors that can be caused by reading inconsistent memory values, some other possible problems include infinite loops, and accessing invalid pointers. In order to deal with this many STM implementations abort a transaction before it can access an inconsistent view of the memory, but it depends on the consistency criterion. Chapter 2 looks at the problem of correctness in transactional memory in more detail.

1.4.5 Nesting

One of the suggested benefits of writing code using transactions is that they be composable, which could be a huge benefit over locks in terms of code simplicity and re-usability. Still it is not obvious how implement composability correctly and efficiently in transactional memory.

The term *nesting* is used to describe the execution of transactions within other transactions. Different way of implementing nesting have been studied with varying properties. The simplest way to implement nesting is called *flattening* [], **[NOTE!!!!: Where is this citation from??]** in this model a nested transaction is combined together with its parent, so it and its parent execute as if it were a single transaction. This is nice because it is simple and it is composable, but it creates larger and larger transactions, limiting performance.

A slightly more complex model *closed nesting* [120] allows a transaction C to run as a separate transaction within its parent P, but when C commits, its changes are only visible to P and not visible to the rest of the system until P itself commits. Running C as a separate transaction allows it to abort itself without aborting P, hopefully increasing performance over the *flattening* model. By not committing C's changes to shared memory until P commits, it prevents there from being consistency issues or roll backs of shared memory in the case that P aborts after C commits.

A more complex notation of nesting is *open nesting* which allows for nested transactions to write to shared memory upon their commit and not wait until their parent commits. The main advantage of open nesting is performance, like closed nesting it has the advantage that if a nested transactions conflicts with another transaction while it is running and must abort, then it does not have to abort the parent transaction. In addition to this open nesting has the advantage that the memory locations accessed by a nested transaction need not be shared with the parent transaction when detecting conflicts with concurrent transactions. **[NOTE!!!!: remove example??]** For example consider a parent transaction P that accesses memory location x and

a nested transaction N that access memory location y, and a separate transaction S that also accesses y. Now consider the execution where P starts, then N starts and commits, then S starts and commits, and finally P completes executing. In open nesting, P can still commit, because even though N accesses the same memory as S, N has already committed to shared memory before S started, so there is no conflict. In closed nesting and flattening, P might have to abort because N has only committed within P and not to shared memory, so there is a conflict between P and S. [75] gives a comprehensive overview of open nesting. Allowing a transaction to commit to shared memory from within a transaction obviously violates the idea that everything within a transaction is executed atomically, possibly increasing performance, but also making programming transactions more difficult. Given this a new consistency model has to be considered, one such model, *abstract* serializability, is described in [76].

1.4.6 Implementation

This next section will look at how to to actually implement these things, and some of the difficulties that go along with this. Importantly, even though there are many ways to implement software transactional memory, there is no clear "best" method, in [137] they perform benchmarks on many of these options and come conclusions as to which choices are preferred, but it is not clear that their decisions are conclusive.

Read and write sets In a system where multiple transactions are executed optimistically in order to ensure that a transaction execute on a valid state of memory and satisfies a consistency criterion it must perform some sort of validation. Traditionally this process of validation requires keeping what is called a *read set*. A read set contains the set of all memory locations read so far by a transaction as well as some additional information (depending on the implementation) on these locations such as the values read. This read set is then validated based on some event such as when a new read is performed or when a transaction tries to commit, ensuring that the locations read so far are still valid.

In addition to a read set, a *write set* is also maintained by a transactions, keeping track of all the locations written so far by the transaction. Write sets are necessary due to the fact that a transaction might abort.

Write Buffering vs Undo locking When performing a write within a transaction, to the rest of the system the value written must not known until the transaction is guaranteed to commit, i.e. an aborted transaction must not effect the state of memory. Traditionally there have been two ways of keeping track of writes before they have committed in memory.

In an implementation that uses *undo locking* [], **[NOTE!!!!: Citation?]** the transaction performs its write directly to the shared memory, and then in case of an abort, the transaction must rollback the state of the memory to where it was before the transaction performed its first write. In order to prevent other transactions from writing to the shared memory concurrently or reading a value that could be rolled back, a transaction will use some sort of locking mechanism of each piece of memory before it writes to it. This is usually implemented as *visible writers*, which are described later in this section.

In an implementation that uses *write buffering* [] **[NOTE!!!!: Citation?]** when a transaction wants to perform a write to a shared memory location first it will make a local copy of that variable only visible to the transaction. Any subsequent writes this transaction does will be

preformed on this local copy, and if the transaction commits successfully then the value of the local copy will then be written into the shared memory.

There are some advantages and disadvantages to both solutions [137], and depend on how the rest of the protocol is implemented. Often one type performs better then the other depending on the workload.

1.4.7 Conflict Detection

The definition of a conflict in transactional memory is straightforward, two transactions conflict if they are run concurrently and they both access the same shared memory location with at least one being a write. Once a conflict happens, the system must deal with it in some way in order to ensure safety and progress, this can done directly by the STM or a contention manager [42] can be called. Many different solutions for detecting and dealing with conflicts have been proposed and depend on the STM protocol.

Visibility How conflict detection is performed depends on how reads and writes are implemented. Transactional reads and writes can either be *visible* or *invisible* [16]. When a transaction performs an invisible read or write it does not perform any modification to shared meta data, so no other transactions are aware that it has performed the read or write. In a visible implementation, the transaction writes some information to the shared meta data (usually adding its identity to the list of transactions that have read that memory location), allowing other transactions to be aware that this read or write has occurred.

Invisible reads have the advantage of not having to write to shared data, which can become a point of contention at shared memory locations that are accessed frequently. This problem of contention can be especially worry-some for read dominated workloads because contention is being introduced when there are no conflicts and can limit scalability. The principle disadvantage of invisible reads is that *validation* is required each time a new location is read or before committal in order to ensure an inconsistent view of the shared memory is not observed. With visible reads, when a location that has been read gets overwritten by a writing transaction a contention manager is called, so validation is not needed.

[NOTE!!!!: Should clean up this section]

Eager vs Lazy There are two basic concepts for conflict detection, *eager* and *lazy* [42]. An eager scheme will detect conflicts as soon as they happen, while a lazy scheme will detect conflicts at commit time. Write/write, read/write, and write/read conflict detection can be eager or lazy. Depending on which combination of these is chosen makes an impact on many different parts of a TM.

Eager An implementation can detect read/write, write/read, and write/write conflicts eagerly. Eager read/write conflict detection requires that the implementation use visible reads, and eager write/write conflict detection requires that the implementation use visible writes, while eager write/read conflict detection can use either visible reads or visible writes. **[NOTE!!!!: Not sure about write/read]**

If visible writes are used than read/write conflicts are detected sooner than if invisible writes are used. With visible writes the conflict is detected as soon as the writing transaction performs its write and acquires the lock for the memory location, while with invisible writes the conflict

is only detected when the writing transaction tries to commit. Read/write conflicts are detected because when a write occurs (or the writing transaction tries to commit) the writing transaction checks to see if there are any active readers in the list for this memory location, and if there are, then a read/write conflict has occurred and is dealt with according to the implementation and contention manager.

The visibility of writes and reads effect write/read conflicts. If a write is visible then from when a transaction acquires the write lock until it commits (or aborts), if a separate transaction performs a read at this location then a write/read conflict is detected and the conflict is dealt with. With invisible writes, write/read conflicts can still be detected eagerly, but require visible reads and are not detected until the writing transaction commits. When the writing transaction commits it will check if there are any readers for the memory location, and if there are then a write/read conflict has occurred and is dealt with.

When a write occurs the transaction checks to see if there is another transaction that owns the lock on this memory location, and if there it means a write/write conflict has occurred, and the conflict is dealt with according to the implementation.

Lazy Read/write, write/read, and write/write conflicts can also be detected lazily. This means they are not detected until commit time or when the read would cause the transaction to see an inconsistent view of the memory (depending on the consistency condition). They all use invisible reads and writes.

The lazy detection of a write/read or read/write conflict is done by the transaction that does the read. The conflict is not detected at the time of the read, but instead either when the transaction tries to perform its next read, or tries to commit. In order to be able to tell if a conflict has occurred the transaction will *validate* its read set at each read and at commit time (again depending on consistency criterion), which means to check if the combination of reads it has done so far are still consistent with respect to the chosen consistency criterion. Since the transaction doing the write has no idea the read has occurred it is usually unaffected by the read.

If write/write conflicts are detected lazily then they are found at the commit time of the transaction that commits last. The transaction that commits first has no idea there is a conflict so it is usually unaffected. When the second transaction commits it must make sure that by committing with the write/write conflict it will not violate the consistency criterion. Many implementations choose their serialization point as time of their commit operation and it is easy to see that in this case a write/write conflict by itself will not violate consistency and in this case it is not necessary to worry about these types of conflicts.

In order to make sure its operations are viewed a atomic when a writing transaction is committing in most implementation it will grab some sort of lock or set a flag for the memory locations it is going to write preventing other concurrently committing transactions from writing to these locations. When a concurrent transaction tries to commit, but notices that some other transaction is also committing to the same memory, a write/write conflict also occurs and must be handled by the TM implementation.

1.4.7.1 Dealing with conflicts

Once a conflict is detected the STM system must then choose a way to deal with it. Contention management was originally implemented in DSTM [42] as an out-of-bound mechanism that a transaction would call when it detected a conflict with another transaction, asking whether it should back off, abort, or tell the conflicting transaction to abort. Since DSTM was introduced

various contention managers have been proposed, each with varying levels of complexity, admitting increased overhead in the hopes of increasing liveness and producing better overall performance. Some of these include *passive* (or *suicide*), in which a transaction aborts itself whenever it detects a conflict, *timestamp* in which an older transaction aborts younger transactions and a younger transaction waits for older transactions to finish, *karma* which uses a heuristic to determine the amount of work a transaction has done, then aborts the one that has done the least amount of work, and *polka* which extends karma by adding a randomized exponential back-off mechanism.

Difficulties Mostly these contention managers have been proposed and designed by making certain assumptions about what applications transactional memory will be used for, then validating (or disproving) their assumptions by examining the contention manager's performance on a limited set of workloads. Part of the difficulty is that a contention management strategy that may work well for one STM implementation may not work at all for another, this is based on how the STM implements things such as visibility of reads, and eagerness of acquire for writes [33]. Given this, certain TM implementations such as LSA-STM have been tested using a wide array of different contention managers [22] but there is no definite answer as to what type of contention manager works best for which TM properties.

A workload can largely effect how well a contention manager performs, for example passive contention management is know to perform well on workloads with a regular access pattern [140], while polka [138] works well on small scale benchmarks [33]. The obvious problem with this is that there is no "best" contention manager that performs well in all reasonable situations, in [129] they come to this conclusion by running a set of the top performing contention managers on a range of benchmarks designed to simulate real world applications.

Liveness Unfortunately many contention managers do not actually prove any guarantee of progress, they often only suggest why they should work well, there are possibly workloads that can be generated that cause extremely poor performance and admit the live-lock or starvation of transactions. The contention manager *greedy* [38] is an exception to this, it is proven to prevent live-lock and starvation, yet in oder to ensure these properties it introduces a high amount of contention on some shared meta data, resulting in poor performance in workloads with a large amount of short transactions [33]. The liveness and progress of transactional memory is discussed in more detail in chapter 3.

1.5 Simplicity

Given that there are so many different interesting properties to consider when thinking about transactions, it is easy to loose sight of the what we are really looking for. It is important to remember that he primary goal of transactional memory is to make concurrent programming easier and more accessible, with performance following closely as a secondary goal. In reality unfortunately, neither of these goals have yet been realized. While the basic semantics of a transaction are widely agreed on (i.e. transaction atomicity with respect to other transactions satisfied by a consistency criterion such as opacity), 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 sight of 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 workloads [65], many people argue that its performance is not good enough to be an attractive alternative to locks, even considering all the difficulties that surround using locks.

Given the difficulties that currently exist for finding the correct semantics for transactional memory the goal of this thesis is to take a step towards fixed and easy to understand semantics while suggesting further ways to continue forward. The grand goal is that someday a programmer can write an efficient concurrent program with nearly as much ease as he would write a sequential one.

In reality we can not immediately realize or even know if this goal is possible yet for there are far too many questions left open when dealing with the semantics of transactional memory, to many even to introduce in this thesis or let alone solve. Instead, this thesis will look at what we believe are the main areas of transactional memory research that take the ease-of-use as a primary concern. In this way, each chapter will separately introduce a research area before looking closely at a specific open problem from that area and suggesting an STM protocol as a solution to the problem. Finally each chapter will mention some of the well know open questions from each area.

1.5.1 Plan

Traditionally research has taken the view that for a protocol to be considered a valid STM implementation then it should ensure transactions are atomic with respect to each other and all transactions (aborted transaction included) execute in a consistent state of memory. Once this is satisfied the research then focuses on improving the implementing protocol by increasing its performance or having it satisfy desirable properties (interestingly, this properties are usually related to improving performance). Following this model, the first chapter looks at the specific properties of *invisible reads* (a property important for the speed and scalability of an STM) and *permissiveness* (a property preventing a protocol from aborting transactions unnecessarily) showing that they are not compatible with the correctness condition of *opacity* before introducing a protocol that does satisfy these properties using the correctness condition of *virtual world consistency*.

When using this traditional view of transactional memory, issues that are not solved by ensuring a consistency criterion such as opacity are left up to the programmer. Some of these issues include things like how to deal with aborted transactions, how to deal with memory accessed inside and outside of transactions, how transactions are nested, how to deal with I/O, among others. The programmer then has to understand these issues and must then choose an appropriate STM protocol based on his needs. The second and the third chapters look at research that takes the approach that requiring the programmer to make these decisions is not appropriate for an easy to use concurrent programming abstraction. The second chapter suggests looking at ways to simplify the traditional STM semantics while the third chapter suggests expanding the traditional semantics to consider issues that are likely to come up when using STM in a realistic system (things that are ignored by consistency criterion such as opacity) and provide straightforward solutions to them.

Specifically, the second chapter looks 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. In detail it looks at the problem of aborted transactions and suggests that the programmer should not have to be aware of aborts at all, following this suggestion it introduces a protocol that ensures every transaction issued by a process commits no matter the concurrency pattern of other threads in the system.

The area of research looked at in the third chapter suggests expanding the basic semantics of transactions. Taking just the traditional definition of transactions from the first chapter only leaves a very limited view of how transactions fit into the big picture of concurrent computing. It only considers the interaction between transactions themselves, leaving the interaction between transactions and other entities in the system undefined. This is not optimal considering ease-of-use as if a programmer uses some of these mechanisms he would encounter undefined results or will have to be familiar with how different STM protocols specifically interact with each of these mechanisms. This chapter specifically looks at shared memory accesses, attempting to answer the question, "how should the system act when shared memory is accessed inside and outside of a transaction¿' To answer this we suggest and define a term called *terminating strong isolation* which ensures that a transaction's atomicity will not be violated by reads and writes performed outside of the transaction by other processes without holding back the progress of these reads and writes. A protocol protocol ensuring this property is presented.

The fourth chapter looks into the idea of providing efficient abstractions as libraries for programmers to reuse from within their transactions. Not only does this mean the programmer does not need to design such libraries himself, but it allows an expert programmer who knows well the synchronization pattern of STM to write high performance implementations designed specifically for transactions. It looks into designing efficient data structures implementing the map abstraction.

By solving small problems in each one of these areas hopefully this thesis will help move forward the ease of use of transactional memory and by mentioning some of the open problems in each of these areas will hopefully promote future strides towards ease of use.

1.6 STM computation model and base definitions

[NOTE!!!!: Need to ensure this stuff is true for all the protocols]

Before getting into the content we will introduce the model of STM that will be used throughout this thesis.

1.6.1 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).

1.6.2 Transactions and object operations

Transaction A transaction is a piece of code that is produced on-line by a sequential process (automaton), that is assumed to be executed atomically (commit) or not at all (abort). This means that (1) the transactions issued by a process are totally ordered, and (2) the designer of a transaction does not have to worry about the management of the base objects accessed by the transaction. Differently from a committed transaction, an aborted transaction has no effect on the shared objects. A transaction can read or 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 A transaction is started by calling the begin_transaction() operation. 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 important goal of the concept of STM is to make concurrent programming easier and more accessible 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, as will be seen in the later chapters in this thesis). 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 structures that can be used to provide a map abstraction, with examples such as a skip-list, binary tree, or hash table. Choosing 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 data structures, even when considering a single data structure 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 better or wore 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 (but they can still help him).

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. Beneath these atomic blocks lies the STM implementation which has many different aspects. Since the introduction of transactional memory in 1993 [12] 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. (Some of these properties are discussed in the introduction of this thesis in section 1.4)

In an ideal world there would exist a *"perfect"* STM algorithm that ensures all desirable properties without making any sacrifices. Unfortunately this algorithm has not yet been discovered (if it is even possible). In fact many of these desirable properties have been little more then introduced and many of their implications on how they affect STM algorithms or how they interact with each other has yet to be explored. Motivated by this, this chapter examines two desirable 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 additional 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 committed, i.e. the protocols 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

As viewed, the recurring 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 already have our syntax defined for the programmer ($atomic\{\ldots\}$) and a basic idea of what this means, "the code in the atomic block will appear as if it has been executed instantaneously", yet we need to precisely define what this means for an STM algorithm. At the heart of this we have consistency criterion. These criterion 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 consistency 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 constraint that the equivalent sequential 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 referred to as linearizability [13] 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 committed 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 extremely 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 $\widehat{S} = (S, \rightarrow_S)$ of a partial order $\widehat{PO} = (PO, \rightarrow_{PO})$ is a topological sort of this partial order, i.e.,

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

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

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

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

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

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

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

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

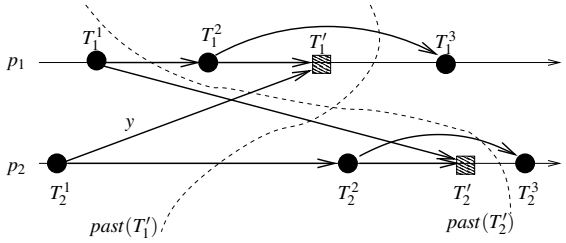


Figure 2.1: Examples of causal pasts

have committed, while the ones with a gray 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 implications of these properties on STM algorithms.

We start by proving that an STM protocol cannot implement invisible reads, opacity, and permissiveness at the same time. This is important because it shows that no matter how well we keep track of the interactions between transactions, an opaque STM protocol 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 unnecessarily. Importantly changing virtual world consistency 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.

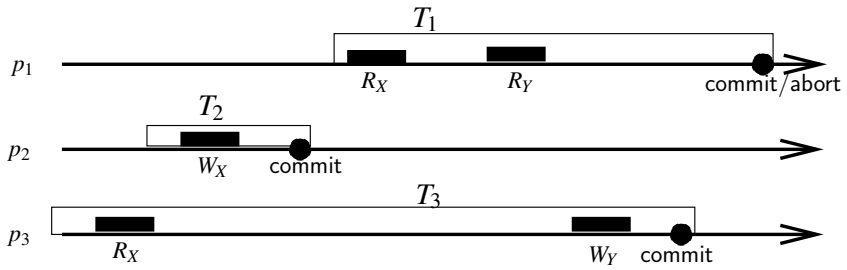


Figure 2.2: Invisible reads, opacity and permissiveness are incompatible

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. $T_C \rightarrow_S T_D \rightarrow_S T_B$ 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.

$\square_{Theorem\ 2}$

[NOTE!!!!: 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 "$X.\mathsf{read}_{T_1}(); Y.\mathsf{read}_{T_1}()$" of the aborted transaction T_1 can be ordered after T_2 (and T_3 does not appear in its causal past).

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 5.6) 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 $\langle v_{init}, 0, +\infty, 0, \bot, \bot \rangle$, where v_{init} is the initial value of X.

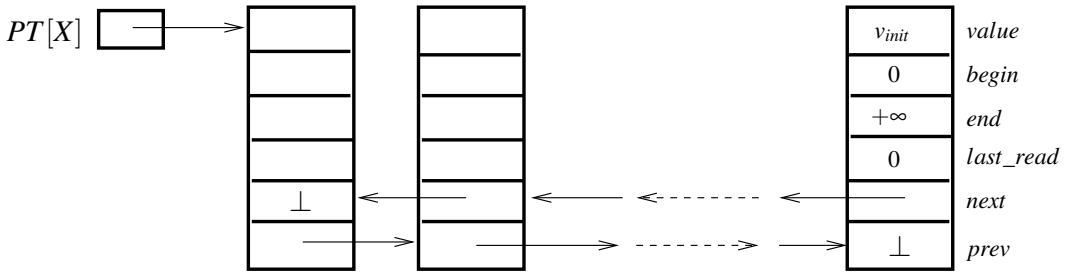


Figure 2.3: List implementing a transaction-level shared object X

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 begin_T():
(01) window_bottom_T ← last_commit_i; window_top_T ← +∞; lrs_T ← ∅; lws_T ← ∅.
  ==========================================================================================
operation X.read_T():
(02) if (∄ local cell associated with the R/W shared object X) then
(03)    allocate local space denoted lcell(X);
(04)    x_ptr ← PT[X];
(05)    lcell(X).value ← (x_ptr ↓).value;
(06)    lcell(X).begin ← (x_ptr ↓).begin;
(07)    lcell(X).origin ← x_ptr;
(08)    window_bottom_T ← max(window_bottom_T, lcell(X).begin);
(09)    lrs_T ← lrs_T ∪ X;
(10)    for each (Y ∈ lrs_T) do window_top_T ← min (window_top_T, (lcell(Y).origin ↓).end) end for;
(11)    if (window_bottom_T ≥ window_top_T) then return(abort) end if
(12) end if;
(13) return (lcell(X).value).
  ==========================================================================================
operation X.write_T(v):
(14) if (∄ local cell associated with the R/W shared object X) then allocate local space lcell(X) end if;
(15) lws_T ← lws_T ∪ X;
(16) lcell(X).value ← v;
(17) return(ok).
  ==========================================================================================
operation try_to_commit_T():
(18) lock all the objects in lrs_T ∪ lws_T;
(19) for each (Y ∈ lrs_T) do window_top_T ← min (window_top_T, (lcell(Y).origin ↓).end) end for;
(20) for each (Y ∈ lws_T) do window_bottom_T ← max((PT[Y] ↓).last_read, window_bottom_T) end for;
(21) if (window_bottom_T ≥ window_top_T) then release all locks and disallocate all local cells; return(abort) end if;
(22) commit_time_T ← select a (random/heuristic) time value ∈ ]window_bottom_T..window_top_T[;
(23) for each (X ∈ lws_T) do (PT[X] ↓).end ← commit_time_T end each;
(24) for each (X ∈ lws_T) do
(25)    allocate in shared memory a new cell for X denoted CELL(X);
(26)    CELL(X).value ← lcell(X).value; CELL(X).last_read ← commit_time_T;
(27)    CELL(X).begin ← commit_time_T; CELL(X).end ← +∞;
(28)    PT[X] ← ↑CELL(X)
(29) end for;
(30) for each (X ∈ lrs_T) do
(31)    (lcell(X).origin ↓).last_read ← max((lcell(X).origin ↓).last_read, commit_time_T)
(32) end for;
(33) release all locks and disallocate all local cells; last_commit_i ← commit_time_T;
(34) return(commit).
```

Figure 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 5.6. When p_i invokes this operation, it returns the value locally saved in $lcell(X).value$ if $lcell(X)$ exists (lines 02 and 13). If $lcell(X)$ has not yet been allocated, p_i does it (line 03) and updates its fields *value*, *begin* and *origin* with the corresponding values obtained from the shared memory (lines 04-07). Process p_i then updates $window\_bottom_T$ and $window\_top_T$. These updates are as follows.

- The algorithm defines the commit time of transaction T as a point of the time line such that T could have executed all its read and write operations instantaneously as

that time. Hence, T cannot be committed before a committed transaction T' that wrote the value of a shared object X read by T. According to the algorithm implementing the try_to_commit$_T()$ operation (see line 27), the commit point of such a transaction T' is the time value kept in $lcell(X).begin$. Hence, p_i updates $window\_bottom_T$ to $\max(window\_bottom_T, lcell(X).begin)$ (line 08). X is then added to $lrst_T$ (line 09).

- Then, p_i updates $window\_top_T$ (the top side of T's commit window, line 10). If there is a shared object Y already read by T (i.e., $Y \in lrs_T$) that has been written by some other transaction T'' (where T'' is a transaction that wrote Y after T read Y), then $window\_top_T$ has to be set to $commit\_time_{T''}$ if $commit\_time_{T''} < window\_top_T$. According to the algorithm implementing the try_to_commit$_T()$ operation, the commit point of such a transaction T'' is the date kept in $(lcell(Y).origin\downarrow).end$. Hence, for each $Y \in lrs_T$, p_i updates $window\_bottom_T$ to $\min\big(window\_top_T, (lcell(Y).origin\downarrow).end$ (line 10).

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

The X.write$_T(v)$ operation The algorithm implementing this operation is described at lines 14-17 of Figure 5.6. If there is no local cell associated with X, p_i allocates one (line 14) and adds X to lws_T (line 15). Then it locally writes v into $lcell(X).value$ (line 16) and return ok (line 17). Let us observe that no X.write$_T()$ operation can entail the abort of a transaction.

2.4.1.4 The try_to_commit$_T()$ operation

The algorithm implementing this operation is described in Figure 5.6 (lines 18-34). A process p_i that invokes try_to_commit$_T()$ first locks all transaction-level shared objects X that have been accessed by transaction T (line 18). The locking of shared objects is done in a canonical order in order to prevent deadlocks.

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

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

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

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

b. For each register X that has been read by T, p_i updates the field *last_read* to the maximum of its previous value and $commit\_time_T$ (lines 30-32). (Actually, this base version of the protocol remains correct when $X \in lrs_T$ is replaced by $X \in (lrs_T \setminus lws_T)$. (As this improvement is no longer valid in the final version of the 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 33) and returns the value *commit* (line 34).

On the random selection of commit points It is important to notice that, choosing randomly commit points (line 22, Figure 5.6), 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.write$_T()$" as the predicate "T is a committed transaction and the operation X.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. $\forall T1, T2 \in \mathscr{C}, \forall X : (T1 \xrightarrow{X}_{rf} T2) \Rightarrow (\nexists T3 : X.\text{write}_{T3}() \wedge T1 \rightarrow_S T3 \rightarrow_S T2)$.

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 20). Because $window\_bottom_{T'} > commit\_time_T$ (line 31), we have $T \rightarrow_S T'$. By transitivity, this holds for all the transactions issued by a process. $\square_{Lemma\ 2}$

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

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 $T1 \rightarrow_S T2$. $\square_{Lemma\ 3}$

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

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

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

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

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

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

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 28). Thus $T1 \xrightarrow{X}_{rf} T2 \Rightarrow T1 \xrightarrow{X}_{lock} T2$.

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

- Case $T3 \xrightarrow{X}_{lock} T2$. Let $lcell(X)$ be the local cell of $T2$ representing X. When $T2$ executes line 19), $T3$ has already updated the field *end* of the cell pointed by $lcell(X).origin$ with $commit\_time_{T3}$. $T2$ will then update $window\_top_{T2}$ at a smaller value than $commit\_time_{T3}$,contradicting the original assumption $T3 \rightarrow_S T2$.
- Case $T2 \xrightarrow{X}_{lock} T3$. When $T3$ executes line 20, $T2$ has already updated the field *last_read* of the cell pointed by $PT[X]$. $T3$ will then update $window\_bottom_{T3}$ at a value greater than $commit\_time_{T2}$, contradicting the original assumption $T3 \rightarrow_S T2$, which completes the proof of the lemma.

$\square_{Lemma\ 5}$

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 28), and thus has already updated the field *end* (line 23) of the cell pointed by $lcell(X).origin$ ($lcell(X)$ being the local cell of T representing X). T will then observe $window\_bottom_T \geq window\_top_T$ (line 11) and will not complete *last_op*, again a contradiction, which completes the proof of the lemma. $\square_{Lemma\ 9}$

2.4.3.1 VWC and read invisibility

Theorem 3 *The algorithm presented in Figure 5.6 satisfies virtual world consistency and implements invisible read operations*

Proof The proof that the algorithm presented in Figure 5.6 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 31-13 of Figure 5.6): 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 5.6. 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.

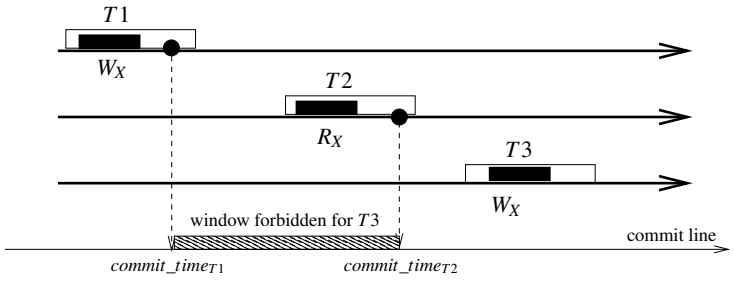


Figure 2.5: Commit intervals

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 5.6, 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 5.6. The lines with the same number are identical in both algorithms, while the number of the lines of Figure 5.6 that are modified are postfixed by a letter. The new/modified parts are the followings.

- Lines 20.A-20.I replace line 20 of Figure 5.6 that was computing the value of $window\_bottom_T$. These new lines compute instead the set of intervals that constitute $commit\_set_T$. To that end they suppress from the initial interval $]window\_bottom_T, window\_top_T[$, all the time intervals that would invalidate values read by committed transactions. This is done for each object $X \in lws_T$ (line 20.H; see section 2.4.4.2 for an example). If $commit\_set_T$ is empty, the transaction T is aborted (line 21.A).
- The commit time of a transaction T is now selected from the intervals in $commit\_set_T$ (line 22.A).
- Line 23 of Figure 5.6 was assigning, for each $X \in lws_T$, its value to $(PT[X]\downarrow).end$, namely the value $commit\_time_T$. This is now done by the new lines 23.A-23.E. Starting from $PT[X]$, these statements use the pointer *prev* to find the cell (let us denote it say CX) of the list implementing X whose field $CX.end$ has to be assigned the value $commit\_time_T$. Let us remember that $CX.end$ defines the end of the lifetime of the value kept in $CX.value$. This cell CX is the first cell (starting from $PT[X]$) such that $CX.begin < commit\_time_T$.
- Line 24 of Figure 5.6 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():
(18)   lock all the objects in lrs_T ∪ lws_T;
(19)   for each (Y ∈ lrs_T) do window_top_T ← min(window_top_T, (lcell(Y).origin ↓).end) end for;
(20.A) commit_set_T ← { ]window_bottom_T, window_top_T[ };
(20.B) for each (X ∈ lws_T) do
(20.C)   x_ptr ← PT[X]; x_forbid_T[X] ← ∅;
(20.D)   while ((x_ptr ↓).last_read > window_bottom_T) do
(20.E)       x_forbid_T[X] ← x_forbid_T[X] ∪ { [(x_ptr ↓).begin, (x_ptr ↓).last_read] };
(20.F)       x_ptr ← (x_ptr ↓).prev
(20.G)   end while
(20.H) end for;
(20.I) commit_set_T ← commit_set_T \ ⋃_{X∈lws_T} (x_forbid_T[X]);
(21.A) if (commit_set_T = ∅) then release all locks and disallocate all local cells; return(abort) end if;
(22.A) commit_time_T ← select a (random/heuristic) time value ∈ commit_set_T;
(23.A) for each (X ∈ lws_T) do
(23.B)   x_ptr ← PT[X];
(23.C)   while ((x_ptr ↓).begin > commit_time_T) do x_ptr ← (x_ptr ↓).prev end while;
(23.D)   (x_ptr ↓).end ← min((x_ptr ↓).end, commit_time_T)
(23.E) end for;
(24.A) for each (X ∈ lws_T) such that (commit_time_T > (PT[X] ↓).begin) do
(25)       allocate in shared memory a new cell for X denoted CELL(X);
(26)       CELL(X).value ← lcell(X).value; CELL(X).last_read ← commit_time_T;
(27)       CELL(X).begin ← commit_time_T; CELL(X).end ← +∞;
(28.A)     CELL(X).prev ← PT[X]; PT[X] ← ↑CELL(X)
(29.A) end for;
(30)   for each (X ∈ lrs_T) do
(31)     (lcell(X).origin ↓).last_read ← max((lcell(X).origin ↓).last_read, commit_time_T)
(32)   end for;
(33)   release all locks and disallocate all local cells; last_commit_i ← commit_time_T;
(34)   return(commit).
```

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

(line 28.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 20.H of Figure 2.6)

The subtraction operation on sets of intervals of real numbers $commit\_set_T \setminus x\_forbid_T[X]$ has the usual meaning, which is explained with an example in Figure 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 $commit\_time_T \setminus x\_forbid_T[X] = \{\ ]a'..b[, \ ]c..b'[, \ ]g..d'[\ \}$.

2.4.4.3 About the predicate of line 24.A of Figure 2.6

This section explains the meaning of the predicate used at line 24.A: $commit\_time_T > (PT[X]\downarrow).begin$. This predicate controls the physical write in a shared memory cell of the value v that T

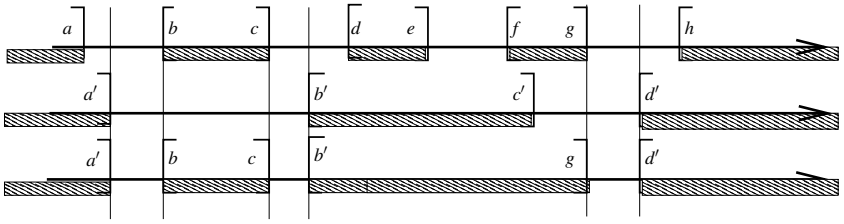


Figure 2.7: Subtraction on sets of intervals

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.

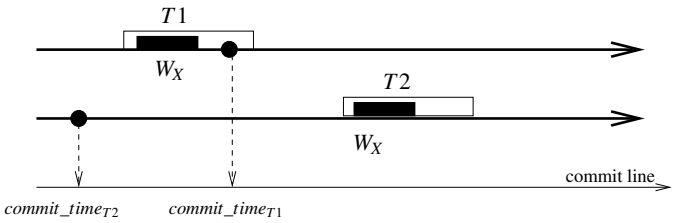


Figure 2.8: Predicate of line 24.A of Figure 2.6

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 $past(T) \backslash past(T')$ is not empty ($past(T)$ does not contain T'). Let $biggest\_ct_{T,T'}$ be the biggest value of the *commit_time* variables (chosen at line 22.A) of the transactions in the set $past(T) \cap past(T')$ if it is not empty, or 0 otherwise. Suppose that every transaction in $past(T) \backslash past(T')$ chooses the smallest value possible for its *commit_time* variable. These values cannot be constrained (for their lower bound) by a value bigger than $biggest\_ct_{T,T'}$, thus they can all be smaller than $biggest\_ct_{T,T'} + \varepsilon$ 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 10 and 19). Suppose now that $T1$ is T'. If there was such a transaction $T2$ in $past(T)$, then there would be no legal linear extension of $\to_{PO}$ 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 $T \to_S T'$.
□*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 10) resulting in a value for $window\_top_T$ that is at least $commit\_time_{T2}$. All the read operations of T will then succeed (line 11).

Let us now consider the case of the try_to_commit() operation. Because all read operations have succeeded, the set $commit\_set_T =]window\_bottom_T..window\_top_T[$ (line 20.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 20.B to 20.H) are the only ones removed from $commit\_set_T$ (line 20.I) and because there is a legal linear extension of $\rightarrow_{PO}$ which includes T, $commit\_set_T$ is not empty and the transaction can commit successfully, which ends the proof of the lemma. $\square_{Lemma\ 11}$

Theorem 4 *The algorithm presented in Figure 5.6, 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. $\square_{Theorem\ 4}$

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 23.

The second array, denoted $MIN\_READ[1..n]$, is made up of one-writer/one-reader atomic registers (let us recall that n is the total number of processes). $MIN\_READ[i]$ is written by p_i only and read by the background task BT only. It is initialized to $+\infty$ and reset to its initial value value when p_i terminates $\mathsf{try\_to\_commit}_T()$ (i.e., just before returning at line 21 or line 34 of Figure 5.6). 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 28.

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] ← PT[X] end for.

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

Figure 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 31 of $\mathsf{begin}_T()$ is replaced by the statement $window\_bottom_T \leftarrow \mathsf{System.get\_time}()$.
- The following statement is added just between line 19 and line 20 of $\mathsf{try\_to\_commit}_T()$ (Figure 5.6):

  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 or maybe should just remove it] 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 Conclusion

To conclude this chapter we will return to the idea that the main purpose 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 criterion. This is important because these consistency criterion define the semantics of a transaction to the programmer. As long as the chosen 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 invisibility and permissiveness come in, a protocol that chooses or not to implement such properties might impact the performance 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 importantly, even though we have weakened the consistency criterion, the programmer will see no difference in the semantics 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 criterion for transactional memory.

The second contribution of this chapter introduces a realistic 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 independently, it considers multiple properties at once. More specifically it looks at the compatibility of properties, how realistic protocols can be designed to satisfy them, and how trade-offs in these properties can be made for efficiency. Hopefully these contributions have convinced the reader that studying 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 examined.

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 protocol without modifying the idea of a transaction. Each of the protocols presented in these various works (DSTM, TL2, TinySTM, SwissSTM, RSTM, etc...) 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, examining 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 the 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 interests (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 provide the amount 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 different approach 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 criterion are sometimes also called 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 behavior. For example opacity prevents any transaction from executing on invalid states of memory, preventing things such as divide by zero exceptions in correct code. Even though consistency criterion remove the complexity of having to deal with undefined states of memory, they leave open other problems for the programmer to deal with that could make using STM more difficult. In this sense we suggest hiding some of these problems from the programmer by dealing with them in the STM implementation. One important thing such consistency criterion 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 amount of contention in the system. Let us now look at some of these properties.

3.1.1 Progress properties

[NOTE!!!!: Need citations for this section] 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 action. For example thread A might want to acquire 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 briefly 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 each other from progressing due to each other holding a lock (or resource) that the other wants to acquire. A simple example of deadlock would be the following: thread A owns lock $l1$ and wants to acquire lock $l2$, concurrently thread B owns lock $l2$ and wants to acquire lock $l1$. In this case thread A and B will be stuck infinitely, waiting to acquire 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 acquire locks in a fixed global order, but this can have negative implications on performance.

Livelock freedom Stronger then deadlock freedom, livelock freedom prevents a state of livelock where threads can never progress due to their progress depending on a shared state that is created by another thread. Consider the following 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 notices that another thread owns a lock it wants, it releases 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 repeated then livelock is observed. Even though the processes are not stalled they are not making progress. 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 resources 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 consistently 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. Inherently 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 difficult 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 scalability, 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 lives 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, if one person on this project might have an approaching 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 amount of time a thread might have to wait could be unknown and harmful to scalability.

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 (known to be a very difficult problem to solve [] **[NOTE!!!!: need citation]**) 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 parallel systems this is an obvious advantage of non-blocking algorithms.

obstruction-freedom 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 ensures that no thread is ever stuck waiting 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 eventually 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 guaranteed by wait-free protocols they are desirable over lock based protocols. Unfortunately wait-free protocols are known to be extremely 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 algorithms that can make 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 intricate, the key design concepts are quite clear.

The first concept has to deal with correctness. How to ensure that the sequential code is executed safely when there can be multiple threads concurrently performing operations on the object (i.e. satisfies linearizability)? 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 modification. In order to achieve 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 concurrent performance will be no better then a single thread, but in certain cases this might be an acceptable trade-off for having 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 modification to the global pointer by some other thread. Now according to the progress guarantee of wait-freedom, every operation by every thread must eventually complete successfully without blocking, meaning the compare-and-swap must not fail infinitely 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 who's 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 effi-

ciency. Interestingly many of the key design concepts from the original universal construction such as helping are also included in these designs.

One interesting 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).

3.2 Transactional memory, progress, and universal constructions

Now that we have discussed progress properties for concurrent code in general as well as universal constructions which can be used to create a concurrent wait-free construction of any sequential object we will now look into how this relates to transactional memory.

3.2.1 Progress properties and transactional memory

[NOTE!!!!: Say that 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 possibly yes, but when considering the ease-of-use of transactional memory it is more interesting to consider that only committed transactions create progress. Then 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 brief 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 specifically 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 done in [130]. 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-freedom for transactional memory. Informally a protocol 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." Additionally in [130] they examine how these properties can be applied in faulty and fault-free systems with or without parasitic transactions (a parasitic transaction is one which is continually executed, but never tries to commit). They show that local-progress 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 a general lock-free STM system is described.

3.2.2 Previous approaches

In order to ensure levels of progress and cope with aborted transactions, several solutions have been proposed with each taking different approaches to progress. A whole range of solutions have been proposed. Some do not directly confront the problem of progress, focusing mainly on the performance of the protocols, while others offer "best effort semantics" (which means that there is no provable strong guarantee) and others offer provable guarantees 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 specifically defined 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 implementation, but this also provides the programmer with additional problems to deal with.

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* 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 time-stamp.

Unfortunately even though such contention managers exist that ensure all transactions commit in a blocking fashion, most STM implementations do not use them in the interest of performance and as a result provide less strong progress guarantees. 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 succeed, but does so at the cost of preventing other transactions from progressing until it finishes. STM protocols supporting irrevocable 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 non-terminating 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 provide the property that every issued transaction is committed. As described in the previous section, in order for the described universal constructions to ensure that each operation is performed successfully threads must help other threads by executing each other'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 at most satisfying non-blocking global progress. Consider for example a thread $T1$ transaction $t1$ and a separate thread $T2$ that executes an infinite sequence of the same transaction $t2$. Now simply consider a history as follows that is repeated infinitely, $t2$ starts executing, $t1$ starts executing, $t2$ commits, $t1$ notices that it conflicts with $t2$ so it must abort. In such a history $t1$ will always abort before it reaches the committal phase so if blocking is not allowed, helping only in the committal phase will not ensure the committal of every transaction.

3.2.3 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 infinitely. 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, a better solution would be 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 then 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. (As a note it should be said that the presentation of previous approaches that

deal with aborted transactions are primarily efficiency-oriented while our construction is more theory-oriented.)

3.2.4 A short comparison with object-oriented universal construction

First it is important to notice that an STM that hides the concept of commit/abort from the programmer has objectives similar to that of a universal construction described earlier 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 linearizable. 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!!!!: update this section]** 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 and reusable 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 for STM, where only the specific locations accessed 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 purpose of tracking the locations the transaction has read so far as well as those that will be modified upon commit, validating these sets in order to ensure correctness.

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 modified using a compare-and-swap in order to ensure progress and correctness.

Another important feature of STM constructions is the systematic use of speculative execution. As discussed in section 3.1.2, a traditional universal construction can expect best case performance to be equal to that of a sequential 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. Still, efficiency is not a first class requirement addressed in our work but, the notion of a speculative execution can be a basis for future work on an STM protocol that will focus on efficiency and providing the same progress as a universal construction.

3.3 A universal construction for transaction based programs

The following sections 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 without depending on the state other processes. More specifically 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".

3.4 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.

3.4.1 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).

3.4.2 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.Compare&Swap(). This operation is a conditional write that returns a boolean value. Its behavior can be described by the following statement:
operation X.Compare&Swap(old, new):
$atomic\{$ **if** $X = old$ **then** $X \leftarrow new$; return($true$) **else** 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.Fetch&Increment(), that adds 1 to X and returns its new value.

3.5 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].

3.5.1 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 $\langle$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.

3.5.2 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).

3.5.3 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;
  $STATE[i] \leftarrow \langle 1, p_i$'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 ($1 \leq x \leq m$)

3.5.4 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 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 $(\downarrow current).next = \bot$ is never true. This happens when new committed transactions (different from P_x's transactions) are repeatedly and infinitely added to the list. The procedure 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($(\downarrow current).next, \bot, \uparrow DESCR$). It is easy to see that this invocation succeeds if and only if *current* points to the last descriptor of the list of committed transactions (line 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).

3.5.5 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 252), then P_x will require help for p_i once this period attains $K2$ (lines 254-257). 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).

3.6 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 256-257). 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 $(\downarrow current).next = \perp$ 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. $\square_{Lemma\ 13}$

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 $((\downarrow current).next = \perp) \vee committed$, 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 $i \in OWNED\_BY[i]$ and P_x invokes infinitely often prevent_endless_looping(i), it follows from lines 252-254 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 254-257). 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 $(\downarrow current).tid = \langle i, i\_tr\_sn\rangle$ (where i_tr_sn is T's sequence number among the transactions issued by p_i). When this occurs, P_x's local variable *committed* is set to *true* and P_x stops looping in an internal while loop.

Let us now consider the case of a processor $P_y \neq P_x$. Let us first notice that the only way for P_y to execute T is when T has requested help (line 101 of 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 $(\downarrow current).tid = \langle i, i\_tr\_sn\rangle$ eventually becomes true and processor P_y sets *committed* to *true*. P_y then stops looping inside an internal while loop (line 08 or 23) which concludes the proof of the lemma. $\square_{Lemma\ 14}$

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

Proof Considering a processor P_x, let $i \in OWNED\_BY[x]$ 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 $STATE[i]$ 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. $\square_{Lemma\ 15}$

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. $\square_{Lemma\ 16}$

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. $\square_{Lemma\ 17}$

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). $\square_{Lemma\ 18}$

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(). $\square_{Lemma\ 19}$

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 up-to-date (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.

□*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.

□*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.

□*Theorem* 5

3.7 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.

□*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 $O(m^2)$.

3.8 Conclusion

3.8.1 Additional notes

The aim of the universal construction for STM 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 exits 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.

3.8.2 A short discussion

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 similar to the previous chapter in suggesting properties and showing how a protocol can implement them, takes a more visible approach 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 solely an implementation concern. This frees the programmer from having to consider if

his transaction might not commit and to either try to prevent such a situation, or to come up with ways to deal with it when it does.

As a final motivation for these simplified 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 issues that could arise in aborted transactions. This liveness suggested in this chapter for STM protocols differs from previous liveness 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, infinite 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. An important problems in this area yet has a clear solution is the problem of what happens when a transaction is nested within another. How should these transactions be treated? Should a separate consistency or liveness condition be considered for these transactions? Given that composition is a strong argument for the use of transactions, coming up with a simple solution to nesting is vital for the future of STM.

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 takes a different look on transaction semantics. The first chapter 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 in the interest of ease-of-use.

Following the discussion of the previous chapter we will promote our change of semantics by contrasting 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 structure 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 committal. This allows the programmer to create multi-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 amount of money m from account A to another account B. In order to do this he creates a transaction that first reads the balance on the source account ($A.balance$), 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 arise where a balance 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 should satisfy more then just a correctness condition such as opacity. The reason for this is that opacity only considers the atomicity between transactions themselves, and to motivate this suggestion we will go 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 linearizable 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 important things to point out, first is that these operations are contained to their data structure implementation, the status of shared memory outside of the data structure has no affect on these operations. Second is that these operations fully implement 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 additional functionality such as in-order iteration, then he would have to choose a different implementation providing the appropriate abstraction. This might lead the programmer to consider data structure as a separate entity from that of the program where this entity is accessible through some fixed interface, he need not be aware of the implementation below the interface.

Transactions on the other hand are integrated into the users program rather then compromising some separate entity. When a user places a transaction 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 integrated and contained within a user's program instead of being separate entities behind a fixed interface.

So now we have concurrent objects whose operations follow some correctness condition allowing them to be accessed as something interesting to the programmer as a separate 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 choices. 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 exist as an in-

tegrated part of a users program. This happens when the programmer places a group of reads and writes inside a block of code bounded by the keywords *transaction_begin* and *transaction_end*. This atomic operation exists directly as a piece of his code preceded and followed by code also written by himself. 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 accessible from within transactions, or should it be safe to access memory inside and outside of transactions at any time, or should it be something in-between?

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 separated object concept. One way to think of it is that the transactions represent a 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 ensuring 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 transactional 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 concurrency issues 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 extremely 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 implements 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 separate section of memory for transactions is also not consistent with the concept 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 purposely subject himself to these difficulties in the interest of performance. Examining more formally these performance over ease-of-use concepts, several STM protocols have been proposed allowing the existence of transactions that might not appear as being atomic. These protocols 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 likely-hood 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.

Privatization Privatization is an interesting solution problem to the splitting of memory between transactions and non-transactional accesses. As the name implies, privatization gives the programmer the ability to privatize 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 privatizing transaction within which he makes some memory unaccessible from other processes' transactions. Once a process has privatized some memory then is safe to access it outside of transaction within the privatizing thread. After privatization, 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 privatization 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 providing privatization safety is not as simple as just implementing an STM protocol 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 privatization have been proposed, ranging from solutions where a programmer uses a special keyword to mark which transactions are privatizing, to others that ensure safety by having a privatizing transaction block until all other concurrent transactions have finished, to more complex non-blocking solutions.

A main advantage of privatization is in the case of performance, as when a thread privatizes some memory, it can then access it without overhead. While privatization is an interesting solution, we feel that is does not fit in our model of ease-of-use for two main reasons. Firstly the separation between the memory safe to access by transactions and that safe to access outside of transactions is still separated, privatization/ publication just gives the user the ability 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 advantage of privatization he has to decide when and what to privatize 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 privatization, 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 privatization 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 following simple intuitive extension that can be applied to an STM protocol in order to ensure strong isolation. In order to provide this extension, 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 separate section of memory for transactions and their atomicity is preserved resulting in a simpler framework for the programmer then privatization 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 interest 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 implemented as mini-transactions 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 understand the intricacies 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 separate the memory.

Using this definition, the properties referred to as *containment* and *non-interference* are sat-

isfied. These properties were defined in [111] in order to describe the synchronization 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.

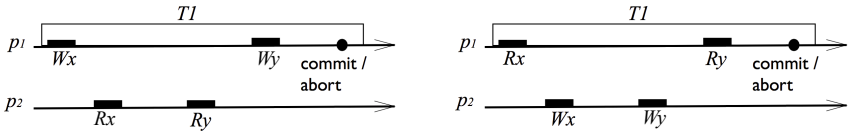


Figure 4.1: Left: *Containment* (operation R_x should not return the value written to x inside the transaction). Right: *Non-Interference* (wile it is still executing, transaction T_1 should not have access to the values that were written to x and y by process p_2).

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 respect 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 possibility 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 implemented using locks or by mini-transactions) then they are susceptible to the same possibility 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 strong isolation should also consider the progress of non-transactional operations. In order to deal with this we suggest that a STM protocol should implement *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 previous 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 happening 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 completely 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 interesting, 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 performance 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 chosen 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 algorithm 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 privatization safe have also been examined [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 consistent. 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 verified 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 released 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 implement 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 associated 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 Fetch&increment() atomically adds one to the register and returns its previous value. The operation denoted C&S() (for compare and swap) is a conditional write. 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 $\uparrow (pt\downarrow) = pt$.

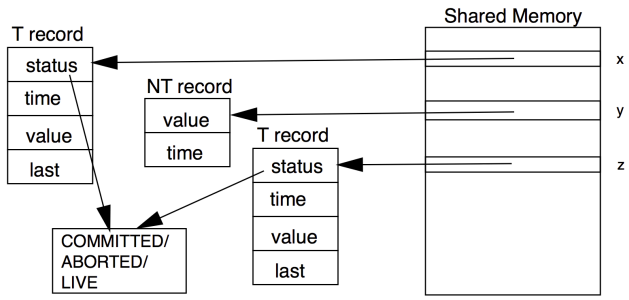


Figure 4.2: The memory set-up and the data structures that are used by the algorithm.

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.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. Then instead of waiting for the live transaction to finish committing it is directed to abort (line 211), ensuring that the operation's wait-free progress as well as 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 **??** 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 233). 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 236) 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 233-234).

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 **??**). In the case of a read only transaction, the transaction is aborted and restarted as an update transaction (line **??**). 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 230).

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 238), 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 245 - 246) 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 247).

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
(226)  if addr ∈ ws then return (item.value from addr in ws) end if;
(227)  tmp ← (↓ addr); validate ← false;
(228)  if (tmp is of type NT)
(229)    then if (tmp.time ≥ rv) then validate ← true end if
(230)         value ← tmp.value;
(231)    else
(232)         if ((status ← (↓ tmp.status)) ≠ COMMITTED )
(233)           then if (status = LIVE) then abort() else value ← tmp.last end if;
(234)           else value ← tmp.value
(235)         end if;
(236)         if (tmp.time > rv) then validate ← true end if;
(237)  end if;
(238)  if this is an update transaction then add ⟨addr, value⟩ to rs end if;
(239)  if (validate)
(240)    then if this is an update transaction then if (validate_by_value()) then abort() end if;
(241)           else abort() and restart as an update transaction end if;
(242)  end if;
(243)  return (value)
end operation.

operation transactional_write(addr, value) is
(244)  if addr ∉ ws
(245)    then allocate a new variable item of type T;
(246)         item ← (value, (↑ status), ∞); ws ← ws ∪ ⟨addr, item⟩;
(247)    else set item.value with addr in ws to value
(248)  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 249-251).

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 252) 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 253-262) 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 256-257) 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 256). 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 263). Following this, the read set of the transaction is validated for correctness (line 263). Once validation has been performed the operation must ensure that non of its writes have been concurrently overwritten by non-transactional operations (lines 264-267)

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

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

Figure 4.5: Transaction begin/commit.

if so then the transaction must abort in order to (line 266) to ensure consistency. During this check the transaction updates the *time* value of its T-records to the transactions logical time (line 265) 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 268). 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 (i.e. *tmp.status* = LIVE) (line 276) or has had its changed since it was first read by the transaction (line 280) otherwise it succeeds. The transaction is directed to abort if validation fails (lines 276, 280) by returning true, returning false otherwise. Note that in the case that a location is being updated (*tmp.status* = LIVE) and the location is in the write set of the current transaction then it does not abort, as this record belongs to the transaction currently being executed by the process performing the validation. Instead the *last* value of the record is validated. Before the validation is performed the local variable *rv* is updated to be the current value of *GVC* (line 272). 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.

```
operation validate_by_value() is
(272)  rv ← GVC;
(273)  for each ⟨addr, value⟩ in rs do
(274)    tmp ← (↓ addr);
(275)    if (tmp is of type T ∧tmp.status ≠ COMMITTED)
(276)          then if (tmp.status = LIVE ∧item ∉ ws) then return(true) end if;
(277)            new_value ← tmp.last;
(278)          else new_value ← tmp.value
(279)    end if;
(280)    if (new_value ≠ value) then return(true) end if;
(281)  end for;
(282)  return(false).
end operation.

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

Figure 4.6: Transactional helper operations.

When a transaction is aborted in the present algorithm, the status of the current transaction is set to ABORTED (line 283) and it is immediately restarted as a new transaction.

4.5 Proof of correctness

This section will prove the correctness of the algorithm by showing the linearizability of the transactions and non-transactional operations. Specifically we will show that every transaction (aborted or committed) as well as non-transactional operations have a unique linearization point, meaning that there is some instant in time during the execution of each operation where it appears to have taken place. Note that we do not use an STM specific consistency criterion such as opacity or virtual world consistency to prove correctness as traditionally they do not consider non-transactional operations.

Before detailing the linearization points of the operations we will introduce a few lemmas that will help with the proofs.

Lemma 23 *Neither the value field of a T or NT record nor the last field of a T record is changed once a pointer in shared memory is set to reference the record.*

Proof. In the case of an NT record, the *value* field is set just once on line 222 of non_transactional_write(), which is immediately after the record is allocated, and before any shared pointers are set. For a T record, allocation happens on line 245 of transactional_write(), then the *value* field is set on lines 246 and 247 of the same operation. The *last* field is set later on line 256 or 257 of the try_to_commit() operation, but the record is not referenced by a shared pointer until later in this operation by performing a compare and swap on line 261. □

Lemma 24 *The status field of a transaction record will change from LIVE to either COMMITTED or ABORTED exactly once.*

Proof. The operation begin_transaction() is called each time a new transaction start and each time a transaction is aborted. During this operation a new *status* variable is allocated and is

set to LIVE. The status variable is then accessible by the process that is executing the transaction or by other processes through T records created during the transaction (line 246 of transaction_write()) after they are added to shared memory during try_to_commit(). *status* can be modified in three places: On line 211 of non_transactional_read and on line 268 of try_to_commit, but in both these cases the value of *status* is changed from LIVE to either COMMITTED or ABORTED by a compare and swap. The third place is on line 283 of the abort where *status* is set to ABORTED by a normal write, but here *status* must either still be LIVE, or already ABORTED. The reason for this is that the only place *status* can be changed to COMMITTED is after a successful compare and swap during the try_to_commit() operation, but in this case the transaction is completed (lines 268–270). □

Lemma 25 *An aborted transaction makes no changes to the state of memory.*

Proof. For this proof we only need to consider transactions that abort after they have performed at least one successful compare and swap on line 261 of try_to_commit as transactions that have aborted before this have not made any modifications to shared memory. On line 261 the transaction compares and swaps each of its T records from its write set into shared memory. What is important here is that before the compare and swap the transaction updates the *last* field of its T record for *addr* using either the *value* or the *last* field of the current record at *addr*. The *value* field is used if the record is of type NT or comes from a committed transaction, otherwise the *last* field is used, this follows the same pattern as the read and validate operations which were shown to all use the same value from a record in lemma 25. Now the compare and swap on line 261 used to add the T record to memory ensures that the record referenced by *addr* has not changed since the *last* field of the new record was set. Now given that the transaction will abort (setting its *status* to aborted), any read or validation that dereferences this record will use the *last* field which is the same value that was in memory previously. □

Lemma 26 *For any T or NT record that is referenced by some shared address addr, any read operation (transactional or non-transactional) or validation (in* validate_by_value()*) to addr that dereferences this record will return (or validate) the same value.*

Proof. By lemma 23 we know that the *value* filed of T and NT records as well as the *last* field of T records will never change once referenced by shared memory, so to complete this proof we need to show that reads and validations are always performed using the same field (either *value* or *last*) of a record. There are two cases to consider, when the record dereferenced is of type NT and when the record is of type T.

NT Record In this case by looking at the code it is easy to see that both the transactional and non-transactional read as well as validation operations return/validate the value contained in the *value* field of the NT record dereferenced. In the case where a transactional_read() to this address causes a transactional abort, even though *value* is not returned, linearizability is not violated as aborted transactions appear to have not executed at all.

T Record This case is a bit more difficult as the value returned depends on the *status* of the transaction that created this record. First consider any non_transactional_read() that has dereferenced this object, on line 211 the compare and swap ensures that the status of the transaction

is no longer LIVE. Now by lemma 24 we know that *status* will only change from LIVE once. So if *status* is ABORTED then the *last* field is returned, otherwise if COMMITTED then the *value* field is returned. Next consider any transactional_read() that dereferences this object, in the same way as the non_transactional_read(), if *status* is ABORTED then the *last* field is returned and if COMMITTED then the *value* field is returned. Otherwise in the case that *status* is still LIVE, the transaction simply aborts (line 233). The validate_by_value() operation follows the same pattern, with the exception if the *status* is LIVE and the record being validate in the write set of the current transaction. In this case the process executing the validation is also the process who owns the record, so here the *last* field is used for validation as by lemma 30 we know this is the value previously in shared memory. but here the compare □

Lemma 27 *The* non_transactional_write *operation is linearizable.*

Proof. The linearization point of the non_transactional_write(addr, value) operation is line 223 where *addr* is set to point to next_write. There is no pre-condition needed for this operation as it acts as a simple write operation. The post-condition is that the value in memory at *addr* is *value*, in other words *next_write.value* is returned by read operations to *addr* (transactional and non-transactional) that are linearized after the write. For either type of read operation their linearization point is when *addr* is accessed dereferencing the record there (the linearization of these operations is proved in later lemmas), therefore any read operation linearized after this NT-write (and before another operation modifies *addr*) will dereference the pointer written by the NT-write. Using this fact along with lemma 25 completes the proof. □

Lemma 28 *The* non_transaction_read *operation is linearizable.*

Proof. The linearization point of the non_transaction_read(addr, value) is line 208 where the record pointed to by *addr* is dereferenced from memory. The pre and post conditions are that the value in memory at *addr* is *value*, or more specifically, any other read operation to *addr* (transactional and non-transactional) whose access to *addr* (line 208 of non_transaction_read and 226 of transaction_read) dereferences the same record as *tmp* returns the same value. Since the object dereferenced at the linearization point is pointed to by *addr* then the proof follows from lemma 25. □

Lemma 29 *The value validated by a successful* validate_by_value() *or returned by a successful* transactional_read() *operation to address addr is the same value that would be validated or returned by any read (non-transactional or transactional) that took place at the time of the most recent read of GCV by this process used to set rv for the transaction.*

Proof. By lemma 25 we know that for any record dereferenced the value read or validated will always be the same for that record, so the rest of this proof will consider that specific value when discussing a dereferenced record. There are two places where *rv* can be set for a transaction, either at the start of a transaction on line 250 of the begin_transaction() operation, or before validation on line 272 the validate_by_value() operation.

For the validate_by_value() case we have each location and its corresponding value read so far by the transaction contained in the read set (including the current read operation (line **??**)) before *rv* is set. In the validate_by_value() operation after *rv* is set, every location in the read

set is validated. This means that when *GCV* is read and *rv* is set all of the values are the same as they were when first read.

Let us now consider the case where the most recent value for *rv* was set in begin_transaction(). We will now show that the record at the location currently being accessed by the transactional_read() has the same value as when *GCV* was read in begin_transaction().

NT record If the object dereferenced on line 226 of transactional_read() is an NT record then we must have $tmp.time < rv$ as otherwise a validation would have been called, updating *rv* (lines **??-??**). Now on line 223 of the non_transactional_write() operation *addr* is set to reference the NT record, following this *GCV* is read whose value is then used to update the record's *time* field. As previously stated this value must be smaller then *rv*, and given that the value of *rv* was set using *GCV*, which is a synchronized growing only counter, *addr* must reference this NT record before *rv* was set, and has not changed up to the point where the record was dereferenced by the read.

T record If the object dereferenced on line 226 of transactional_read() is a T record then we must have $tmp.time \leq rv$ and $tmp.status =$ COMMITTED or $tmp.status =$ ABORTED, otherwise the current read operation would have directed the transaction to perform a validation (lines **??-??**). Given this, we just need to show that the T record dereferenced was in memory before *GCV* was incremented to the value read in begin_transaction (i.e. before $GCV = rv$). First consider the case where $tmp.status =$ COMMITTED, given that *tmp* is a T record, it was created during a try_to_commit operation. During this operation the compare and swap on line 261 ensures that *tmp* is referenced by *addr* before the process increases *GCV* by performing an increment and fetch (lines **??-??**). Then on line 265, the value returned from the increment and fetch is used to set the *time* field of each T record created by the transaction. Given this and that we know $tmp.time \leq rv$, *tmp* must be referenced by *addr* before $GCV = rv$. Now consider the case where $tmp.status =$ ABORTED, the compare and swap on line 261 of try_to_commit and lemma 30 ensures that the value in *tmp.last* was the previous value at *addr* as well as that *tmp* is referenced by *addr* before the increment and fetch. Following this (just like in the of $tmp.status =$ COMMITTED) we have the value returned from the increment and fetch used to set the *time* field of each T record created by the transaction. Therefore *tmp* must have been referenced by *addr* before $GCV = rv$.

Finally, given that begin_transaction() is called only once per transaction we have all previous read operations still being consistent. □

Lemma 30 *A successful* transactional_read() *operation (i.e. one that does not cause the transaction to abort) in linearizable, with a linearization point that includes the current read as well all previous reads of the transaction.*

Proof. For the transaction_read operation the linearization point is the most recent time that *rv* was set for the transaction. This could have been The pre and post conditions are that the values returned by this and all the previous reads of the transaction are the values in memory, or more specifically, (using lemma 25) at the linearization point for each *addr* read (including the current one) the location points to the record that contains the same value that was read. The proof of this lemma follows directly from lemma 28. □

Lemma 31 *A committed transaction is linearizable.*

Proof. For the precondition of the committed transaction we have the requirement that for each of the items in the transaction's read set that the value contained in the record currently referenced by the item's address is equal to the value previously read by the transaction. The post condition is for each of the items in the transaction's write set that the value input to the transactional write is set to the *value* field of the record referenced by the corresponding address in shared memory. Here the linearization point is line 263, just after the all of the transactions T records have had their corresponding address in shared memory set to point to them, and before the call to validate_by_value(). The precondition is ensured by the subsequent call to validate_by_value, as if any location in the read set has been changed since it was first read then the transaction would abort. The post condition is ensured by the for loop on lines 265-267 as well as the compare and swap on line 268 changing the transaction's *status* from LIVE to COMMITTED. The for loop goes through each T record in the transaction's write set, checking if a write by some other process has changed the record referenced by *addr* so that it no longer points to this transaction's record and aborting the transaction if there is. Finally the compare and swap setting *status* to COMMITTED ensures that any future read (transactional or non-transactional) that dereferences one of the transaction's records returns the value written by the transaction. Lemma 25 ensures that any read or validation to any one of these records returns the same value. It is interesting to note that at the linearization point the transaction is not guaranteed to commit, but lemmas 30 and 25 ensure that in the case the transaction does abort, it still appears to have not happened at all. This is done in order to ensure the progress and correctness of non-transactional operations that might need to abort a transaction who is in the process of committing. □

Theorem 7 *The algorithm provides linearizability for both committed and aborted transactions and non-transactional reads and writes.*

Proof. This follows directly from lemmas 26, 27, 29, and 31. □

(285) **then if** $(tmp.time \leq time \wedge (\downarrow tmp.status) = \text{LIVE})$

Figure 4.7: Modified line of the NT-read operation.

4.6 From linearizability to serializability

In algorithm 4.3 whenever a NT-read operation encounters a transaction in the process of committing it immediately attempts to abort the transaction. Interestingly this can be prevented in the case that the thread performing the NT-read has not observed a write that is serialized after the transaction by serializing the current NT-read before the transaction. Algorithm 4.7 describes the changes to the NT-read operation, line 210 of the original algorithm is modified to become line 285 in algorithm 4.7. It is important to note that this change means that the NT-reads will no longer be linearizable, instead they become serializable. The reason for this is that the NT-read operation could have started after the linearization point of a transaction, but before the transaction the transaction's status has been changed from LIVE to COMMITTED. Then if the NT-read

returns the value contained in *tmp.last* its real time occurrence was after the linearization point of the transaction, but it is still serialized before the transaction.

4.7 Conclusion

How should STM deal with mechanisms in the system that are not transactions? This chapter has suggested that the semantics of a transaction should be expanded in order that the programmer know there is fixed way that these different systems will interact. In this sense it has looked at the problem of accesses to the same shared memory being performed inside and outside of a transaction and suggested termination strong isolation to be used to deal with this.

Following this suggestion this chapter has presented an algorithm that achieves terminating 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. This allows the programmer to know his transaction will execute atomically even in the presence of non-transactional memory access, while ensuring the non-transactional access are never blocked from progressing.

The interaction between transactional and non-transactional accesses to shared memory is just one among many possible interactions that a programmer may face when using transactions. For example he might want to use locks or atomic hardware operations along side his transactions, he might want to use I/O within his transactions. Each of these and many more things deserve an investigation on possible models in which they are considered alongside the semantics of transactions.

Chapter 5

Libraries for STM

[NOTE!!!!: Should include this?? What to include here??]

As mentioned in the introduction a partial solution to the difficulty of concurrent programming is to provide concurrent libraries of popular abstractions such as data structures. Generally these are finely tuned high performance libraries providing the programmer with useful operations. The implementations of such libraries are often very complex and difficult to understand, but are provided with a simple to use interface for anyone to use. By doing this experts can create algorithms providing strong guarantees and good performance that can then be widely reused.

In a perfect world transactional memory would not need such libraries, as one could just directly place a sequential implementation of an abstraction directly in transactions and it would perform as well and provide the same guarantees as a library provided by an expert. Unfortunately this is not completely true. Due to the general nature of the synchronization required by STM, a sequential data structure implementation directly placed in transactions will never perform as well as a highly tuned concurrent (lock based or non-blocking) implementation of a data structure done by an expert given that the expert can consider the specific synchronization needed to implement that data structure safely.

This gives us the opportunity to design libraries designed specifically for use inside of transactions that take into account the synchronization considerations of STM. By doing this we can provide a programmer using transactions with abstractions that perform closer to highly optimized non-transactional ones.

A further advantage of providing efficient libraries in STM is given by the reusable nature of code implemented using transactions, a programmer can use the operations of the abstractions along with his own code to create new efficient atomic operations. For example a programmer might use a map abstraction that provides an insert and a delete operation to create an atomic move operation by simply combing the insert and delete in a transaction.

By providing a programmer with a large selection of transaction based implementations of commonly used abstractions his life becomes much easier. Also if these libraries provide good performance it gives an additional reason to use STM as an attractive alternative to concurrent code using locks or even highly tuned non-transactional abstractions. Furthermore STM has the advantage over these other types of concurrent abstractions that the programmer can reuse the libraries in his transactions to create new atomic operations specific to his needs.

This chapter will focus on designing efficient data structures in transactional memory providing the map abstraction.

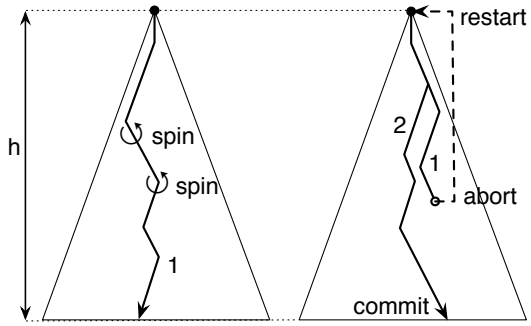


Figure 5.1: A balanced search tree whose complexity, in terms of the amount of accessed elements, is **(left)** proportional to h in a pessimistic execution and **(right)** proportional to the number of restarts in an optimistic execution

5.1 Introduction

The multicore era is changing the way we write concurrent programs. In such context, concurrent data structures are becoming a bottleneck building block of a wide variety of concurrent applications. Generally, they rely on invariants [81] which prevent them from scaling with multiple cores: a tree must typically remain sufficiently balanced at any point of the concurrent execution.

Yet it is unclear how one can adapt a data structure to access it efficiently through transactions. As a drawback of the simplicity of using transactions, the existing transactional programs spanning from low level libraries to topmost applications directly derive from sequential or pessimistically synchronized programs. The impacts of optimistic synchronization on the execution is simply ignored.

To illustrate the difference between optimistic and pessimistic synchronizations consider the example of Figure 5.1 depicting their step complexity when traversing a tree of height h from its root to a leaf node. On the left, steps are executed pessimistically, potentially spinning before being able to acquire a lock, on the path converging towards the leaf node. On the right, steps are executed optimistically and some of them may abort and restart, depending on concurrent thread steps. The pessimistic execution of each thread is guaranteed to execute $O(h)$ steps, yet the optimistic one may need to execute $\Omega(hr)$ steps, where r is the number of restarts. Note that r depends on the probability of conflicts with concurrent transactions that depends, in turn, on the transaction length and h. Although it is clear that a transaction must be aborted before violating the abstraction implemented by this tree, e.g., inserting k successfully in a set where k already exists, it is unclear whether a transaction must be aborted before slightly unbalancing the tree implementation to strictly preserve the balance invariant.

We introduce a *speculation-friendly* tree as a tree that transiently breaks its balance structural invariant without hampering the abstraction consistency in order to speed up transaction-based accesses. Here are our contributions.

- We propose a speculation-friendly binary search tree data structure implementing an associative array and a set abstractions and decoupling the operations that modify the abstraction (we call these *abstract transactions*) from operations that modify the tree structure

itself but not the abstraction (we call these *structural transactions*). An abstract transaction either inserts or deletes an element from the abstraction and in certain cases the insertion might also modify the tree structure. Some structural transactions rebalance the tree by executing a distributed rotation mechanism: each of these transactions executes a local rotation involving only a constant number of neighboring nodes. Some other structural transactions unlink and free a node that was logically deleted by a former abstract transaction.

- We prove the correctness (i.e., linearizability) of our tree and we compare its performance against existing transaction-based versions of an AVL tree and a red-black tree, widely used to evaluate transactions [6, 42, 61, 70, 36, 82, 65]. The speculation-friendly tree improves by up to 1.6× the performance of the AVL tree on the micro-benchmark and by up to 3.5× the performance of the built-in red-black tree on a travel reservation application, already well-engineered for transactions. Finally, our speculation-friendly tree performs similarly to a non-rotating tree but remains robust in face of non-uniform workloads.

- We illustrate (i) the portability of our speculation-friendly tree by evaluating it on two different Transactional Memories (TMs), TinySTM [36] and $\mathscr{E}$-STM [66] and with different configuration settings, hence outlining that our performance benefit is independent from the transactional algorithm it uses; and (ii) its reusability by composing straightforwardly the remove and insert into a new move operation. In addition, we compare the benefit of relaxing data structures into speculation-friendly ones against the benefit of only relaxing transactions, by evaluating elastic transactions. It shows that, for a particular data structure, refactoring its algorithm is preferable to refactoring the underlying transaction algorithm.

The paper is organized as follows. In Section 5.2 we describe the problem related to the use of transactions in existing balanced trees. In Section 5.3 we present our speculation-friendly binary search tree. In Section 5.5 we evaluate our tree experimentally and illustrate its portability and reusability. In Section 5.6 we describe the related work and Section 5.7 concludes.

5.2 The Problem with Balanced Trees

In this section, we focus our attention on the structural invariant of existing tree libraries, namely the *balance*, and enlighten the impact of their restructuring, namely the *rebalancing*, on contention.

Trees provide logarithmic access time complexity given that they are balanced, meaning that among all downward paths from the root to a leaf, the length of the shortest path is not far apart the length of the longest path. Upon tree update, if their difference exceeds a given threshold, the structural invariant is broken and a rebalancing is triggered to restructure accordingly. This threshold depends on the considered algorithm: AVL trees [55] do not tolerate the longest length to exceed the shortest by 2 whereas red-black trees [58] tolerate the longest to be twice the shortest, thus restructuring less frequently. Yet in both cases the restructuring is triggered immediately when the threshold is reached to hide the imbalance from further operations.

Generally, one takes an existing tree algorithm and encapsulates all its accesses within transactions to obtain a concurrent tree whose accesses are guaranteed atomic (i.e., linearizable),

however, the obtained concurrent transactions likely *conflict* (i.e., one accesses the same location another is modifying), resulting in the need to abort one of these transactions which leads to a significant waste of efforts. This is in part due to the fact that encapsulating an *update* operation (i.e., an insert or a remove operation) into a transaction boils down to encapsulating four phases in the same transaction:

1. the modification of the abstraction,
2. the corresponding structural adaptation,
3. a check to detect whether the threshold is reached and
4. the potential rebalancing.

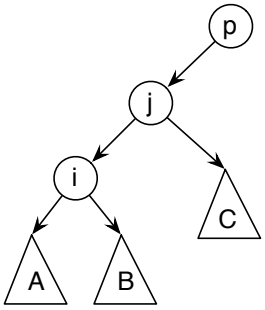
A transaction-based red-black tree An example is the transaction-based binary search tree developed by Oracle Labs (formerly Sun Microsystems) and other researchers to extensively evaluate transactional memories [6, 42, 61, 70, 36, 82, 65] . This library relies on the classical red-black tree algorithm that bounds the step complexity of pessimistic insert/delete/contains. It has been slightly optimized for transactions by removing sentinel nodes to reduce false-conflicts, and we are aware of two benchmark-suite distributions that integrate it, STAMP [61] and synchrobench[1].

Each of its update transactions encapsulate all the four phases given above even though phase (1) could be decoupled from phases (3) and (4) if transient violations of the balance invariant were tolerated. Such a decoupling is appealing given that phase (4) is subject to conflicts. In fact, the algorithm balances the tree by executing rotations starting from the position where a node is inserted or deleted and possibly going all the way up to the root. As depicted in Figure 5.2(a) and (b), a rotation consists of replacing the node where the rotation occurs by the child and adding this replaced node to one of its subtrees. A node cannot be accessed concurrently by an abstract transaction and a rotation, otherwise the abstract transaction might miss the node it targets while being rotated downward. Similarly, rotations cannot access common nodes as one rotation may unbalance the others.

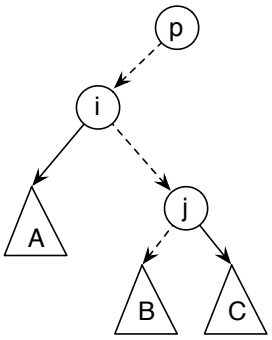
Moreover, the red-black tree does not allow any abstract transaction to access a node that is concurrently being deleted from the abstraction because phases (1) and (2) are tightly coupled within the same transaction. If this was allowed the abstract transaction might end up on the node that is no longer part of the tree. Fortunately, if the modification operation is a deletion then phase (1) can be decoupled from the structural modification of phase (2) by marking the targeted node as logically deleted in phase (1) effectively removing it from the set abstraction prior to unlinking it physically in phase (2). This improvement is important as it lets a concurrent abstract transaction travel through the node concurrently being logically deleted in phase (1) without conflicting. Making things worse, without decoupling these four phases, having to abort within phase (4) would typically require the three previous phases to restart as well. Finally without decoupling only contains operations are guaranteed not to conflict with each other. With decoupling, insert/delete/contains do not conflict with each other unless they terminate on the same node as described in Section 5.3.

To conclude, for the transactions to preserve the atomicity and invariants of such a tree algorithm, they typically have to keep track of a large *read set* and *write set*, i.e., the sets of accessed

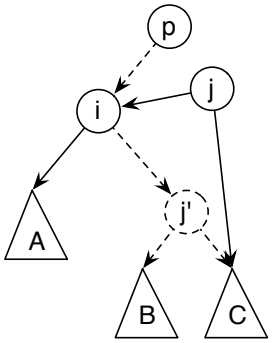
[1] http://lpd.epfl.ch/gramoli/php/synchrobench.php



(a) Initial tree



(b) Result of a classical right rotation



(c) Result of the new right rotation

Figure 5.2: The classical rotation modifies node j in the tree and forces a concurrent traversal at this node to backtrack; the new rotation left j unmodified, adds j' and postpones the physical deletion of j

memory locations that are protected by a transaction. Possessing large read/write sets increases the probability of conflicts and thus reduces concurrency. This is especially problematic in trees because the distribution of nodes in the read/write set is skewed so that the probability of the node being in the set is much higher for nodes near the root and the root is guaranteed to be in the read set.

Illustration To briefly illustrate the effect of tightly coupling update operations on the step complexity of classical transactional balanced trees we have counted the maximal number of reads necessary to complete typical insert/remove/contains operations. Note that this number includes the reads executed by the transaction each time it aborts in addition to the read set size of the transaction obtained at commit time.

Update	0%	10%	20%	30%	40%	50%
AVL tree	29	415	711	1008	1981	2081
Oracle red-black tree	31	573	965	1108	1484	1545
Speculation-friendly tree	29	75	123	120	144	180

Table 5.1: Maximum number of transactional reads per operation on three 2^{12}-sized balanced search trees as the update ratio increases

We evaluated the aforementioned red-black tree, an AVL tree, and our speculation-friendly tree on a 48-core machine using the same transactional memory (TM) algorithm[2]. The expectation of the tree sizes is fixed to 2^{12} during the experiments by performing an insert and a remove with the same probability. Table 5.1 depicts the maximum number of transactional reads per operation observed among 48 concurrent threads as we increase the update ratio, i.e., the proportion of insert/remove operations over contains operations.

For all three trees, the transactional read complexity of an operation increases with the update ratio due to the additional aborted efforts induced by the contention. Althought the red-black and the AVL trees objective is to keep the complexity of pessimistic accesses $O(\log_2 n)$ (proportional to 12 in this case), where n is the tree size, the read complexity of optimistic accesses grows significantly (14× more at 10% update than at 0%, where there are no aborts) as the contention increases. As described in the sequel, the speculation-friendly tree succeeds in limiting the step complexity raise (2.6× more at 10% update) of data structure accesses when compared against the transactional versions of state-of-the-art tree algorithms. An optimization further reducing the number of transactional reads between 2 (for 10% updates) and 18 (for 50% updates) is presented in Section 5.3.3.

5.3 The Speculation-Friendly Binary Search Tree

We introduce the speculation-friendly binary search tree by describing its implementation of an associative array abstraction, mapping a key to a value. In short, the tree speeds up the access transactions by decoupling two conflict-prone operations: the node deletion and the tree rotation. Although these two techniques have been used for decades in the context of data management [64, 74], our algorithm novelty lies in applying their combination to reduce transaction

[2]TinySTM-CTL, i.e., with lazy acquirement [36].

```
State of node n
(01) node a record with fields:
(02)    k ∈ ℕ, the node key
(03)    v ∈ ℕ, the node value
(04)    ℓ, r ∈ ℕ, left/right child pointers, initially ⊥
(05)    left-h, right-h ∈ ℕ, local height of left/right child, initially 0
(06)    local-h ∈ ℕ, expected local height, initially 1
(07)    del ∈ {true, false}, indicate whether logically deleted, initially false
State of process p
(08) root, shared pointer to root
operation find(k)_p
(09) next ← root
(10) while (true) do
(11)    curr ← next; val ← curr.k
(12)    if (val = k) then break() end if
(13)    if (val > k) then next ← read(curr.r)
(14)          else next ← read(curr.ℓ) end if
(15)    if (next = ⊥) then break() end if
(16) end while
(17) return(curr).
operation contains(k)_p
(18) begin_transaction()
(19) result ← true; curr ← find(k)
(20) if (curr.k ≠ k) then result ← false
(21)    else if (read(curr.del)) then result ← false end if
(22) end if
(23) try_to_commit()
(24) return(result).
operation insert(k, v)_p
(25) begin_transaction()
(26) result ← true; curr ← find(k)
(27) if (curr.k = k) then
(28)          if (read(curr.del)) then write(curr.del, false)
(29)             else result ← false end if
(30)    else allocate a new node
(31)          new.k ← k; new.v ← v
(32)          if (curr.k > k) then write(curr.r, new)
(33)             else write(curr.ℓ, new) end if
(34) end if
(35) try_to_commit()
(36) return(result).
```

Figure 5.3: Algorithm for the operations of the protocol

aborts. We first depict, in Algorithm **??**, the pseudocode that looks like sequential code encapsulated within transactions before presenting, in Algorithm **??**, more complex optimizations.

5.3.1 Decoupling the tree rotation

The motivation for rotation decoupling stems from two separate observations: (i) a rotation is tied to the modification that triggers it, hence the process modifying the tree is also responsible for ensuring that its modification does not break the balance invariant and (ii) a rotation affects different parts of the tree, hence an isolated conflict can abort the rotation performed at multiple

```
operation right_rotate(parent, left-child)_p
(01) begin_transaction()
(02) if (left-child) then n ← read(parent.ℓ)
(03)    else n ← read(parent.r) end if
(04) if (n = ⊥) then return(false) end if
(05) ℓ ← read(n.ℓ)
(06) if (ℓ = ⊥) then return(false) end if
(07) ℓr ← read(ℓ.r); write(n.ℓ, ℓr); write(ℓ.r, n)
(08) if (left-child) then write(parent.ℓ, ℓ)
(09)    else write(parent.r, ℓ) end if
(10) update-balance-values()
(11) try_to_commit()
(12) return(true).
operation delete(k)_p
(13) begin_transaction()
(14) result ← true
(15) curr ← find(k)
(16) if (curr.k ≠ k) then result ← false
(17)    else
(18)          if (read(curr.del)) then result ← false
(19)             else write(curr.del, true) end if
(20) end if
(21) try_to_commit()
(22) return(result).
operation remove(parent, left-child)_p
(23) begin_transaction()
(24) if (left-child) then n ← read(parent.ℓ)
(25)    else n ← read(parent.r) end if
(26) if (n = ⊥ ∨ ¬read(n.del)) then return(false) end if
(27) if ((child ← read(n.ℓ)) ≠ ⊥)
(28)    then if (read(n.r) ≠ ⊥) then return(false) end if
(29)    else child ← read(n.r) end if
(30) end if
(31) if(left-child) then write(parent.ℓ, child)
(32)    else write(parent.r, child) end if
(33) update-balance-values()
(34) try_to_commit()
(35) return(true).
```

Figure 5.4: Algorithm for the operations of the protocol

nodes. In response to these two issues we introduce a dedicated rotator thread to complete the modifying transactions faster and we distribute the rotation in multiple (node-)local transactions. Note that our rotating thread is similar to the collector thread proposed by Dijkstra et al. [64] to garbage collect stale nodes.

This decoupling allows the read set of the insert/delete operations to only contain the path from the root to the node(s) being modified and the write set to only contain the nodes that need to be modified in order to ensure the abstraction modification (i.e., the nodes at the bottom of the search path), thus reducing conflicts significantly. Let us consider a specific example. If rotations are performed within the insert/delete operations then each rotation increases the read and write set sizes. Take an insert operation that triggers a right rotation such as the one depicted in Figures 5.2(a)-5.2(b). Before the rotation the read set for the nodes p, j, i is $\{p.\ell, j.r\}$, where ℓ

and *r* represent the left and right pointers, and the write set is ∅. Now with the rotation the read set becomes $\{p.\ell, i.r, j.\ell, j.r\}$ and the write set becomes $\{p.\ell, i.r, j.\ell\}$ as denoted in the figure by dashed arrows. Due to $p.\ell$ being modified, any concurrent transaction that traverses any part of this section of the tree (including all nodes *i*, *j*, and subtrees *A*, *B*, *C*, *D*) will have a read/write conflict with this transaction. In the worst case an insert/delete operation triggers rotations all the way up to the root resulting in conflicts with all concurrent transactions.

Rotation As previously described, rotations are not required to ensure the atomicity of the insert/delete/contains operations so it is not necessary to perform rotations in the same transaction as the insert or delete. Instead we dedicate a separate thread that continuously checks for unbalances and rotates accordingly within its own node-local transactions.

More specifically, neither do the insert/delete operations comprise any rotation, nor do the rotations execute on a large block of nodes. Hence, local rotations that occur near the root can still cause a large amount of conflicts, but rotations performed further down the tree are less subject to conflict. If local rotations are performed in a single transaction block then even the rotations that occur further down the tree will be part of a likely conflicting transaction, so instead each local rotation is performed as a single transaction. Keeping the insert/delete/contains and rotate/remove operations as small as possible allows more operations to execute at the same time without conflicts, increasing concurrency.

Performing local rotations rather than global ones has other benefits. If rotations are performed as blocks then, due to concurrent insert/delete operations, not all of the rotations may still be valid once the transaction commits. Each concurrent insert/delete operation might require a certain set of rotations to balance the tree, but because the operations are happening concurrently the appropriate rotations to balance the tree are constantly changing and since each operation only has a partial view of the tree it might not know what the appropriate rotations are. With local rotations, each time a rotation is performed it uses the most up-to-date local information avoiding repeating rotations at the same location.

The actual code for the rotation is straightforward. Each rotation is performed just as it would be performed in a sequential binary tree (see Figure 5.2(a)-5.2(b)), but within a transaction.

Deciding when to perform a rotation is done based on local balance information omitted from the pseudocode. This technique was introduced in [59] and works as follows. *left-h* (resp. *right-h*) is a node-local variable to keep track of the estimated height of the left (resp. right) subtree. *local-h* (also a node-local variable) is always 1 larger than the maximum value of *left-h* and *right-h*. If the difference between *left-h* and *right-h* is greater than 1 then a rotation is triggered. After the rotation these values are updated as indicated by a dedicated function (line 5.2). Since these values are local to the node the estimated heights of the subtrees might not always be accurate. The propagate operation (described in the next paragraph) is used to update the estimated heights. Using the propagate operation and local rotations, the tree is guaranteed to be eventually perfectly balanced as in [59, 60].

Propagation The rotating thread executes continuously a depth-first traversal to propagate the balance information. Although it might propagate an outdated height information due to concurrency, the tree gets eventually balanced. The only requirement is that a node knows when it has an empty subtree (i.e., when $node.\ell$ is $\perp$, *node.left-h* must be 0). This requirement is guaranteed since a new node is always added to the tree with *left-h* and *right-h* set to 0 and these values are updated when a node is removed or a rotation takes place. Each propagate operation

is performed as a sequence of distributed transactions each acting on a single node. Such a transaction first travels to the left and right child nodes, checking their *local-h* values and using these values to update *left-h*, *right-h*, and *local-h* of the parent node. As no abstract transactions access these three values, they never conflict with propagate operations (unless the transactional memory used is inherently prone to false-sharing).

Limitations Unfortunately, spreading rotations and modifications into distinct transactions still does not allow insert/delete/contains operations that are being performed on separate keys to execute concurrently. Consider a delete operation that deletes a node at the root. In order to remove this node a successor is taken from the bottom of the tree so that it becomes the new root. This now creates a point of contention at the root and where the successor was removed. Every concurrent transaction that accesses the tree will have a read/write conflict with this transaction. Below we discuss how to address this issue.

5.3.2 Decoupling the node deletion

The speculation-friendly binary search tree exploits logical deletion to further reduce the amount of transaction conflicts. This two-phase deletion technique has been previously used for memory management like in [74], for example, to reduce locking in database indexes. Each node has a *deleted* flag, initialized to false when the node is inserted into the tree. First, the delete phase consists of removing the given key k from the abstraction—it logically deletes a node by setting a *deleted* flag to true (line 5.2). Second, the remove phase physically deletes the node from the tree to prevent it from growing too large. Each of these are performed as a separate transaction and the rotating thread is also responsible for garbage collecting nodes (cf. Section 5.3.4).

The deletion decoupling reduces conflicts by two means. First, it spreads out the two deletion phases in two separate transactions, hence reducing the size of the delete transaction. Second, deleting logically node i simply consists in setting the *deleted* flag to true (line 5.2), thus avoiding conflicts with concurrent abstract transactions that have traversed i.

Find The find operation is a helper function called implicitly by other functions within a transaction, thus it is never called explicitly by the application programmer. This operation looks for a given key k by parsing the tree similarly to a sequential code. At each node it goes right if the key of the node is larger than k (line 5.2), otherwise it goes left (line 5.2). Starting from the root it continues until it either finds a node with k (line 5.2) or until it reaches a leaf (line 5.2) returning the node (line 5.2). Notice that if it reaches a leaf, it has performed a transactional read on the child pointer of this leaf (lines 5.2–5.2), ensuring that some other concurrent transaction will not insert a node with key k.

Contains The contains operation first executes the find starting from the root, this returns a node (line 5.2). If the key of the node returned is equal to the key being searched for, then it performs a transactional read of the *deleted* flag (line 5.2). If the flag is false the operation returns true, otherwise it returns false. If the key of the returned node is not equal to the key being searched for then a node with the key being searched for is not in the tree and false is returned (lines 5.2 and 5.2).

Insertion The insert(k, v) operation uses the find procedure that returns a node (line 5.2). If a node is found with the same *key* as the one being searched for then the *deleted* flag is checked using a transactional read (line 5.2). If the flag is false then the tree already contains k and false is returned (lines 5.2 and 5.2). If the flag is true then the flag is updated to false (line 5.2) and true is returned. Otherwise if the *key* of the node returned is not equal to k then a new node is allocated and added as the appropriate child of the node returned by the find operation (lines 5.2-5.2). Notice that only in this final case does the operation modify the structure of the tree.

Logical deletion The delete uses also the find procedure in order to locate the node to be deleted (line 5.2). A transactional read is then performed on the *deleted* flag (line 5.2). If *deleted* is true then the operation returns false (lines 5.2 and 5.2), if *deleted* is false it is set to true (line 5.2) and the operation returns true. If the find procedure does not return a node with the same *key* as the one being searched for then false is returned (line 5.2 and 5.2). Notice that this operation never modifies the tree structure.

Consequently, the insert/delete/contains operations can only conflict with each other in two cases.

1. Two insert/delete/contains operations are being performed concurrently on some key k and a node with key k exists in the tree. Here (if at least one of the operations is an insert or delete) there will be a read/write conflict on the node's *deleted* flag. Note that there will be no conflict with any other concurrent operation that is being done on a different key.

2. An insert that is being performed for some key k where no node with key k exists in the tree. Here the insert operation will add a new node to the tree, and will have a read/write conflict with any operation that had read the pointer when it was $\perp$ (before it was changed to point to the new node).

Physical removal Removing a node that has no children is as simple as unlinking the node from its parent (lines 5.2–5.2). Removing a node that has 1 child is done by just unlinking it from its parent, then linking its parent to its child (also lines 5.2–5.2). Each of these removal procedures is a very small transaction, only performing a single transactional write. This transaction conflicts only with concurrent transactions that read the link from the parent before it is changed.

Upon removal of a node i with two children, the node in the tree with the immediately larger key than i's must be found at the bottom of the tree. This performs reads on all the way to the leaf and a write at the parent of i, creating a conflict with any operation that has traversed this node. Fortunately, in practice such removals are not necessary. In fact only nodes with no more than one child are removed from the tree (if the node has two children, the remove operation returns without doing anything, cf. line 5.2). It turns out that removing nodes with no more than one children is enough to keep the tree from growing so large that it affects performance.

The removal operation is performed by the maintenance thread. While it is traversing the tree performing rotation and propogate operations it also checks for logically deleted nodes to be removed.

Limitations The traversal phase of most functions is prone to false-conflicts, as it comprises read operations that do not actually need to return values from the same snapshot. Specifically,

```
State of node n
(01) node the same record with an extra field:
(02)    rem ∈ {false, true, true_by_left_rot} indicate whether physically deleted, initially false
operation find(k)_p
(03) curr ← root; next ← root
(04) while(true) do
(05)    while(true) do
(06)        rem ← false; parent ← curr; curr ← next ;val ← curr.k
(07)        if(val = k)
(08)          then if(¬(rem ← read(curr.rem))) then break end if
(09)        end if
(10)        if(val > k ∪ rem = true_by_left_rot)
(11)          then next ← uread(curr.r)
(12)          else next ← uread(curr.ℓ) end if
(13)        if(next = ⊥) then
(14)          if(¬(rem ← read(curr.rem))) then
(15)              if(val > k) then next ← read(curr.r)
(16)                else next ← read(curr.ℓ) end if
(17)              if(next = ⊥) then break end if
(18)            else
(19)              if(val ≤ k) then next ← uread(curr.r)
(20)                else next ← uread(curr.ℓ) end if
(21)          end if
(22)        end if
(23)    end while
(24)    if(curr.k > parent.k) then tmp ← read(parent.r)
(25)        else tmp ← read(parent.ℓ) end if
(26)    if(curr = tmp) then break
(27)        else next ← curr; curr ← parent end if
(28) end while
(29) return(curr).
```

Figure 5.5: Algorithm for the operations of the protocol

by the time a traversal transaction reaches a leaf, the value it read at the root likely no longer matters, thus a conflict with a concurrent root update could simply be ignored. Nevertheless, the standard TM interface forces all transactions to adopt the same strongest semantics prone to false-conflicts [67]. In the next paragraphs we discuss how to extend the basic TM interface to cope with such false-conflicts.

5.3.3 Optional improvements

In previous sections, we have described a speculation-friendly tree that fulfills the standard TM interface [71] for the sake of portability across a large body of research work on TM. Now, we propose to further reduce aborts related to the rotation and the find operation at the cost of an additional lightweight read operation, uread, that breaks this interface. This optimization is thus usable only in TM systems providing additional explicit calls and do not aim at replacing but complementing the previous algorithm to preserve its portability. This optimization complementing Algorithm **??** is depicted in Algorithm **??**, it does not affect the existing contains/insert/delete operations besides speeding up their internal find operation. Here the left rotation is not the exact symmetry of the right rotation code.

```
operation remove(parent, left-child)_p
(01) begin_transaction()
(02) if(read(parent.rem)) then return(false) end if
(03) if(left-child) then n ← read(parent.ℓ)
(04)    else n ← read(parent.r) end if
(05) if(n = ⊥ ∨ ¬read(n.deleted)) then return(false) end if
(06) if((child ← read(n.ℓ)) ≠ ⊥)
(07)    then if(read(n.r) ≠ ⊥) then return(false) end if
(08)    else child ← read(n.r) end if
(09) if(left-child) then write(parent.ℓ, child)
(10)    else write(parent.r, child) end if
(11) write(n.ℓ, parent); write(n.r, parent); write(n.rem, true)
(12) update-balance-values()
(13) try_to_commit()
(14) return(true).
operation right_rotate(parent, left-child)_p
(15) begin_transaction()
(16) if(read(parent.rem)) then return(false) end if
(17) if(left-child) then n ← read(parent.ℓ)
(18)    else n ← read(parent.r) end if
(19) if(n = ⊥) then return(false) end if
(20) ℓ ← read(n.ℓ)
(21) if(ℓ = ⊥) then return(false) end if
(22) ℓr ← read(ℓ.r); r ← read(n.r)
(23) allocate a new node
(24) new.k ← n.k; new.ℓ ← ℓr; new.r ← r
(25) write(ℓ.r, new); write(n.rem, true)
(26) In the case of a left rotate set n.rem to true_by_left_rot
(27) if(left-child) then write(parent.ℓ, ℓ)
(28) else write(parent.r, ℓ) end if
(29) update-balance-values()
(30) try_to_commit()
(31) return(true).
```

Figure 5.6: Algorithm for the operations of the protocol

This change to the algorithm requires that each node has an additional flag indicating whether or not the node has been physically removed from the tree (a node is physically removed during a successful rotate or remove operation). This removed flag can be set to false, true or true_by_left_rot and is initialized to false. In order not to complicate the pseudo code true_by_left_rot is considered to be equivalent to true, only on line 5.3.2 of the find operation is this parameter value specifically checked for.

Lightweight reads The key idea is to avoid validating superfluous read accesses when an operation traverses the tree structure. This idea has been exploited by elastic transactions that use a bounded buffer instead of a read set to validate only immediately preceding reads, thus implementing a form of hand-over-hand locking transaction for search structure [66]. We could have used different extensions to implement these optimizations. DSTM [42] proposes early release to force a transaction stop keeping track of a read set entry. Alternatively, the current distribution of TinySTM [36] comprises unit loads that do not record anything in the read set. While we could have used any of these approaches to increase concurrency we have chosen the

unit loads of TinySTM, hence the name uread. This uread returns the most recent value written to memory by a committed transaction by potentially spin-waiting on the location until it stops being concurrently modified.

A first interesting result, is that the read/write set sizes can be kept at a size of $O(k)$ instead of the $O(k \log n)$ obtained with the previous tree algorithm, where k is the number of nested contains/insert/delete operations nested in a transaction. The reasoning behind this is as follow: Upon success, a contains only needs to ensure that the node it found is still in the tree when the transaction commits, and can ignore the state of other nodes it had traversed. Upon failure, it only needs to ensure that the node i it is looking for is not in the tree when the transaction commits, this requires to check whether the pointer from the parent that would point to i is $\perp$ (i.e., this pointer should be in the read set of the transaction and its value is $\perp$). In a similar vein, insert and delete only need to validate the position in the tree where they aimed at inserting or deleting. Therefore, contains/insert/delete only increases the size of the read/write set by a constant instead of a logarithmic amount.

It is worth mentioning that ureads have a further advantage over normal reads other than making conflicts less likely: Classical reads are more expensive to perform than unit reads. This is because in addition to needing to store a list keeping track of the reads done so far, an opaque TM that uses invisible reads needs to perform validation of the read set with a worst case cost of $O(s^2)$, where s is the size of the read set, whereas a TM that uses visible reads performs a modification to shared memory for each read.

Rotation Rotations remain conflict-prone in Algorithm **??** as they incur a conflict when crossing the region of the tree traversed by a contains/insert/delete operation. If ureads are used in the contains/insert/delete operations then rotations will only conflict with these operations if they finish at one of the two nodes that are rotated by rotation operation (for example in Figure 5.2(a) this would be the node i or j). A rotation at the root will only conflict with a contains/insert/delete that finished at (or at the rotated child of) the root, any operations that travel further down the tree will not conflict.

Figure 5.2(c) displays the result of the new rotation that is slightly different than the previous one. Instead of modifying j directly, j is unlinked from its parent (effectively removing it from the tree, lines 5.3.2–5.3.2) and a new node j' is created (line 5.3.2), taking j's place in the tree (lines 5.3.2–5.3.2). During the rotation j has a removed flag that is set to true (line 5.3.2), letting concurrent operations know that j is no longer in the tree but its deallocation is postponed. Now consider a concurrent operation that is traversing the tree and is preempted on j during the rotation. If a normal rotation is performed the concurrent operation will either have to backtrack or the transaction would have to abort (as the node it is searching for might be in the subtree A). Using the new rotation, the preempted operation will still have a path to A.

As previous noted the removed flag can be set to one of three values (false, true or true_by_left_rot). Only when a node is removed during a left rotation is the flag set to true_by_left_rot. This is necessary to ensure that the find operation follows the correct path in the specific case that the operation is preempted on a node that is concurrently removed by a left rotation and this node has the same key k as the one being searched for. In this case the find operation must travel to the right child of the removed node otherwise it might miss the node with key k that has replaced the removed node from the rotation. In all other cases the find operation can follow the child pointer as normal.

Find, contains and delete The interesting point for the find operation is that the search continues until it finds a node with the *removed* flag set to false (line 5.3.2 and 5.3.2). Once the leaf or a node with the same key as the one being searched for is reached, a transactional read is performed on the *removed* flag to ensure that the node is not removed from the tree (by some other operation) at least until the transaction commits. If *removed* is true then the operation continues traversing the tree, otherwise the correct node has been found. Next, if the node is a leaf, a transactional read must be performed on the appropriate child pointer to ensure this node remains a leaf throughout the transaction (lines 5.3.2–5.3.2). If this read does not return $\perp$ then the operation continues traversing the tree. Otherwise the operation then leaves the nested while loop (lines 5.3.2 and 5.3.2), but the find operation does not return yet.

One additional transactional read must be performed to ensure safety. This is the read of the parent's pointer to the node about to be returned (lines 5.3.2–5.3.2). If this read does not return the same node as found previously, the find operation continues parsing the tree starting from the parent (lines 5.3.2–5.3.2). Otherwise the process leaves the while loop (line 5.3.2) and the node is returned (line 5.3.2). By performing a transactional read on the parent's pointer we ensure the STM system performs a validation before continuing.

The advantage of this updated find operation is that ureads are used to traverse the tree, it only uses transactional reads to ensure atomicity when it reaches what is suspected to be the last node it has to traverse. The original algorithm exclusively uses transactional reads to traverse the tree and because of this, modifications to the structure of the tree that occur along the traversed path cause conflicts, which do not occur in the updated algorithm. The contains/insert/delete operations themselves are identical in both algorithms.

Removal The remove operation requires some modification to ensure safety when using ureads during the traversal phase. Normally if a contains/insert/delete operation is preempted on a node that is removed then that operation will have to backtrack or abort the transaction. This can be avoided as follows. When a node is removed, its left and right child pointers are set to point to its previous parent (lines 5.3.2–5.3.2). This provides a preempted operation with a path back to the tree. The removed node also has its *removed* flag set to true (line 5.3.2) letting preempted operations know it is no longer in the tree (the node is left to be freed later by garbage collection).

5.3.4 Garbage collection

As explained previously, there is always a single rotator thread that continuously executes a recursive depth first traversal. It updates the local, left and right heights of each node and performs a rotation or removal if necessary. Nodes that are successfully removed are then added to a garbage collection list. Each application thread maintains a boolean indicating a pending operation and a counter indicating the number of completed operations. Before starting a traversal, the rotator thread sets a pointer to what is currently the end of the garbage collection list and copies all booleans and counters. After a traversal, if for every thread its counter has increased or if its boolean is false then the nodes up to the previously stored end pointer can be safely freed. Experimentally, we found that the size of the list was never larger than a small fraction of the size of the tree but theoretically we expect the total space required to remain linear in the tree size.

5.4 Correctness Proof

There are two parts in the proof. First we have to ensure the structure of the tree is always a valid binary search tree. This is important because in a binary search tree at any time there is exactly one correct location for a key k, the term used for such a tree is *valid binary tree*. A binary tree that does not have the previous property is simply called a *binary tree*. Second we have to show the insert, delete, and contains operations are linearizable.

It is important to remember for this proof that when a transaction commits the transactional reads and writes appear as if they have happened atomically and that unit-read operations only return values from previously committed transactions (or the value written by the current transaction, if the transaction has written to the location being read).

Another important part of this proof is the way the tree is first created. It is created with a root node with key ∞ so that all nodes will always be on its left subtree. This node will always be the root (i.e. it will not be modified by rotate or removal operations), this makes simpler operations and proofs.

Binary trees Each node has two boolean state variables. When the variable *deleted* is false it entails that the value of *node.key* is in the set represented by the tree. When it is true it entails that the value of *node.key* is not in the set represented by the tree. When the variable *removed* is false it entails that the node exists in the tree (meaning a path exists from the root of the tree to the node). When it is true (or true_by_left_rot) it entails that the node does not exist in the tree (meaning no path exists from the root of the tree to the node). For simplicity throughout the pseudo code true_by_left_rot is considered to be equivalent to true, only on line 5.3.2 of the find operation is this parameter value specifically checked for.

In the proof we will use the phrase *a node can reach a range of keys* which is explained here. Take the root, from this node there is a path to any node in the tree meaning any key that is in the set is reachable from the root. Furthermore for any key that is not in the tree the root has a path to where it would be inserted (the leaf that would be its parent). This means that the root with key k node has a range $[-\infty,\infty]$. Now its left child has range $[-\infty,k]$ and its right child has range $[k,-\infty]$. Or for example consider some node with range $(10,20]$ and key 14. Its left child will have a range $(10,14)$ and its right child will have a range $(14,20]$. have a path to it.

The phrase *a node n has a path to a range of keys at least as large as some other node* n' is also used in the proof. It means that every key that is in the range of n' is also in the range of n (and possibly some more). For example any node will have a path to a range of keys at least at large as its left child (in fact it has the exact range of its left and right child combined).

Set operations Here traditional operations on the set are defined in the context of transactions. It is important to remember that the TM guarantees a linearization of transactions.

The following definitions are used in defining the operations. Saying a key k is in (not in) the set before the committal of a transaction T means that if some transaction $T1$ performs a contains operation on key k and is serialized as the transaction immediately before T, $T1$ would return true (false).

Saying a key k is in (not in) the set after the committal of a transaction T means that if some transaction $T2$ performs a contains operation on key k and is serialized as the transaction immediately after T, $T2$ would return true (false).

delete For transaction that commits a successful delete operation of key k, before the commit k was in the set and afterwards k is not in the set. For transaction that commits a failed delete operation of key k, before and after the commit k was not in the set.

insert For transaction that commits a successful insert operation of key k, before the commit k was not in the set and after the commit k is in the set. For transaction that commits a failed insert operation of key k, before and after the commit k is in the set.

contains For transaction that commits a successful contains operation of key k, before and after the commit k was in the set. For transaction that commits a failed contains operation of key k, before and after the commit k was not in the set.

Lemma 32 *A node has at most one parent with removed* = false.

Proof. Assume by contradiction that a node has more then one parent with *removed* = false. There are three operation where a node can be given a new parent. First during the insert operations on lines 5.2–5.2, but this is a new node so before this line it has no parent. Second during the remove operation on lines 5.3.2–5.3.2, but by line 5.3.2 the other parent has *removed* set to true. Third during the *right-rotate* operation the node l gets a new parent on lines 5.3.2–5.3.2, but by line 5.3.2 the other parent has *removed* set to true. Also during the right_rotate operation the node $n2$ gets l as a parent (line 5.3.2), but since it is a new node it has no other parent. (This holds for the left_rotate operation by symmetry) Given this, a node will have at most one parent with *removed* = false. □

Lemma 33 *A node with removed* = false *only has paths from it to other nodes with removed* = false.

Proof. This proof is by induction on the number j of operations done on a node with key k.

The base case is when a new node is created and added to the tree. This can happen in the insert operation. During the operation a new node is created with no children and for itself it has *removed* = false (line 5.2) and the proof holds.

Now for the induction step from $j = m-1$ to $j = m$, this could be a contains, delete, insert, remove, or rotate operation. First note that the contains and delete operations do not change any children pointers. If this is an insert operation then at $j = m-1$ *left* or *right* must be $\perp$ (line 5.3.2 of the find operation) and the proof obviously still holds. If this is a remove operation, then the child of the node is being removed. This means that the new child will either be $\perp$ or the child of the child (lines 5.3.2–5.3.2), but by induction these nodes have *removed* = false. By symmetry this holds for the right and left children. Otherwise this is a rotate operation. First consider right rotations. By induction we know that node n only has paths to nodes with *removed* = false. After the rotation node l points to nodes that had paths from n before the rotation as well as $n2$ (line 5.3.2) which has *removed* = false. $n2$ points to nodes that had paths from n before the rotation (lines 5.3.2–5.3.2). By symmetry this holds for left rotations. □

Lemma 34 *A node with removed* = false *has at least one parent with removed* = false *that points to it (except the root, as it has no parent).*

Proof. Assume by contradiction that there is no parent with *removed* = false. When a node is first added to the tree it has a parent with *removed* = false (line 5.3.2 of the find operation). The only operations that removes links from nodes are the remove (line 5.3.2) and rotate (line

5.3.2) operations, but in each case when a link is removed, a new link from a different node with *removed* = false is added (lines 5.3.2–5.3.2 of remove and 5.3.2–5.3.2 of rotate). □

Lemma 35 *A node with removed* = false *has a path from the root node to it.*

Proof. Given that the root is always the same node and it always has *removed* = false the proof of this lemma follows directly from lemmas 33 and 34. □

Lemma 36 *A node with removed* = true *has no path from the root node to it.*

Proof. By the way the tree is structured the root node always has *removed* = false. Now it follows directly from lemma 33 that there is no path from the root to a node with *removed* = true. □

Lemma 37 *The nodes with removed* = false *make up a single binary tree.*

Proof. From the structure of a node, it can have at most 2 children. Now by lemmas 32, 35, and 36 the proof follows. □

Lemma 38 *A rotation operation on a valid binary search tree results in a valid binary tree of the nodes with removed* = false.

Proof. From Figure 5.2 which describes the *rotation* operation and due to the use of transactional reads/writes we can see that the resulting tree is equivalent to a tree with a classical binary tree rotation performed on it. □

For the following Lemma, we assume that the find operation is performed correctly (this will be proved in a later lemma).

Lemma 39 *Assuming a correct* find *operation, a successful* insert *operation on a valid binary search tree results in a valid binary tree of the nodes with removed* = false.

Proof. There are two cases to consider.

First the key k that we are inserting is already contained in the tree with *removed* = false (lines 5.3.2–5.3.2 of find). In this case the structure of the tree will not be modified, thus the resulting tree will still be valid.

Second consider that there is no node in the tree with key k. In this case the find operation will return the correct node that will then become the parent of the new node with key k (line 5.2–5.2). Using transactional reads, the find operation ensures that the parent node has *removed* = false (line 5.3.2) and $\perp$ (line 5.3.2) for the child pointer where the new node will be inserted. The new node is then created and added to the tree as the child of the node returned from find. This is done using a transactional write (lines 5.2–5.2) which ensures that the value of the pointer was $\perp$ before the write, and finally resulting in a valid tree containing the new node after the transaction commits. □

Lemma 40 *A successful* remove *operation on a valid binary search tree results in a valid binary tree that does not contain the node removed.*

Proof. A removal can only be preformed on a node n with at least one $\perp$ child which is ensured by a transactional read (lines 5.3.2–5.3.2). Node n being removed is unlinked from its parent (lines 5.3.2-5.3.2) (the parent is ensured to be in the tree by a transaction read on *removed* = false) effectively removing it from the tree (lemma 32). If both children of n are set to $\perp$, then the parent's new child becomes $\perp$ (lines 5.3.2–5.3.2) leaving the tree still valid. If node n has a child c such that $c \neq \perp$, then c becomes the parent's new child (lines 5.3.2–5.3.2). By lemma 33 this c must have *removed* = false and by lemma 35 it is part of the valid binary tree during the transaction (until the transaction commits). Thus the resulting tree is still valid. □

Lemma 41 *Modifications to the tree structure are only performed on nodes with removed* = false.

Proof. A rotate, insert or remove operation can modify the tree.

During a right rotate operation the nodes that are modified are n, l and the parent of n (lines 5.3.2–5.3.2). A transactional read is performed on the remove variable of the parent node (line 5.3.2), this along with lemma 33 ensures that *removed* = false for the parent of n as well as n, and l. By symmetry the left rotate operation also only modifies nodes with *removed* = false.

During a successful insert operation a new node might be added to the tree, in this case its parent node is modified (lines 5.2–5.2), and a transactional read is performed on the parent to ensure that *removed* = false (line 5.3.2 of the find operation).

During a remove operation a node is removed from the tree. A transactional read is performed on the node and its parent (line 5.3.2), this along with lemma 33 ensures that *removed* = false for both nodes. □

Lemma 42 *From a node with removed* = true *there is always a path to some node with removed* = false.

Proof. There are two places where a node can be set to *removed* = true. First during the remove operation, in this case the node that is removed has both of the nodes child pointers are set to a node with *removed* = false (lines 5.3.2–5.3.2). Second during the rotate operation, in this case the node that is removed does not have its child pointers changed. By line 5.3.2 n must have at least 1 child and using lemma 33 this child must have *removed* = false.

Now notice that once the *removed* field of a node is set to true it will never be reverted to false (lemma 41). This along with the use of induction on the length of the path to a node with *removed* = false completes the proof. □

Lemma 43 *From any node every path leads to a leaf node (or $\perp$).*

Proof. This proof is the same as lemma 42 with a small modification.

First consider a node with *removed* = false. By lemma 37 it is clear that there is a path from this node to a leaf node.

Now consider a node with *removed* = true. There are two places where a node's *removed* flag can be set to true. It can either be set in the remove operation, but in this case the node's left and right pointers are set to a node with *removed* = false (lines 5.3.2–5.3.2).

Or it can be set in the rotation operation, first notice that in this case the node's left and right pointers are not changed. Then before the rotation the node has *removed* = false and therefore

by lemma 33 the (node's left and right) pointers will point to either nodes with *removed* = false or $\perp$.

Now that after a node has had *removed* set to true the node will not be modified again (lemma 41), this along with lemma 37 and the use of induction on the length of the path to a node with *removed* = false or $\perp$ completes the proof. □

The phrase *a node that has removed* = true *from a rotation operation* refers to the node that is no longer in the tree after the rotation. For example in figure 5.2(c) this would be the node j. This phrase is used in the following lemmas.

Lemma 44 *A node n with key k that has removed* = true *(*true_by_left_rot*) from a right_rotation operation for its **left** child has a path to a range of keys at least as large as n did before the rotation (including the new node $n2$ with key k), and for its **right** child a path to a range of keys at least as large as the right child before the rotation, or $\perp$.*

Proof. Assume that the node n has key k and before the rotation has a path to a range $[a,b]$. This means that node l (the left child of n) has a path to the range $[a,k]$. Assume node l has key j. The right child of n is either *bot* or some node r with a path to the range $[k,b]$.

After the rotation l keeps it left child with its right child changing to $n2$. $n2$'s right child becomes n's right child, and $n2$'s left child becomes l's old right child. This leaves $n2$ with a path to the range $[j,b]$, and l with a path to the range $[a,b]$.

During the *rotation* operation no modifications are made to node n, except setting *removed* = true. Now n's left child is l giving it a path to the range of keys $[a,b]$. Node n still has the same right child who still has the same range, $[k,b]$. □

Lemma 45 *A node n with key k that has removed* = true *from a left_rotation operation for its **right** child has a path to a range of keys at least as large as n did before the rotation (including the new node $n2$ with key k), and for its **left** child a path to a range of keys at least as large as the left child before the rotation, or $\perp$.*

Proof. This proof follows from symmetry (the left rotation is the mirror of the right rotation) and lemma 44. □

Lemma 46 *A node that has removed* = true *from a* remove *operation has a path to a range of keys as least as large as just before the* remove *operation took place.*

Proof. Assume that the node n (where n is the node to be removed) has key k. Assume that n has a parent node p with range $[a,b]$ with key j. This leaves n with range $[a,k]$ if n is the left child (or $[k,b]$ if n is the right child, the proof of this case will follow by symmetry).

After the removal *removed* will be set to true (line 5.3.2 of remove) for n and both its child pointers will point to p (lines 5.3.2-5.3.2 of remove). Node p will still have a range of $[a,b]$ and will be reachable from n, which completes the proof. □

Lemma 47 *A node that has removed* = true *has either:*

- *for each child a path to a range of keys at least as large as they did when it had removed* = false *or*

- *a single* $\perp$ *child with the other child having a path to a range of keys at least as large as both children before when it had removed* = false.

Proof. The proof follows using lemmas 44, 45, and 46 as well as induction on the length of the path from the node to a node with *removed* = false. □

Lemma 48 *A* find *operation on a valid binary search tree always returns the correct location from the valid binary tree.*

Proof. Assume by contradiction that a find operation does not return the correct location from the valid binary tree. By lemma 37 we know that every node with *removed* = false is a node in the valid binary tree so there are two possibilities. The operation never returns or the operation returns the wrong location. First for the operation never returning. From lemma 43 we know that there are no cycles in the path from any node so the operation will not get stuck in an endless loop. In order for the operation to complete it must reach a node with *removed* = false that either matches the key being searched for (lines 5.3.2–5.3.2) or has $\perp$ as the appropriate child (lines 5.3.2–5.3.2). Lemma 42 is enough to show that this will always happen and the operation will terminate.

The second possibility where the operation returns the wrong location is more difficult to prove. To this end, we prove by induction on the number of nodes traversed by the operation that the find operation will always have a path to the correct location. In order for the operation to reach the correct location the following must be true: After each traversal from one node to the next the operation must either be at the correct location or have a path from the current location to a range of keys that includes the key being searched for. Once the correct location is reached transactional reads are used to ensure that the location remains correct throughout the transaction (lines 5.3.2, and 5.3.2–5.3.2).

- The base case is easy given that the operation starts from the root which is always part of the valid tree and can reach all nodes with *removed* = false (lemma 35).
- Now the induction step. Assume that after $n-1$ nodes have been traversed the operation has a path to a range of keys that includes the key k that is being searched for. If the $n-1^{th}$ node is the correct location the the proof is done, otherwise the operation must travel to the n^{th} node. Now it must be shown that the after the operation travels to the n^{th} node it is still on the correct path. There are 2 possibilities.
  1. The operation can move from a node with *removed* = false (at the time of the load of the child pointer, lines 5.3.2–5.3.2). By lemma 33 the child must also have *removed* = false (at the time of the load of the child pointer), in this case the traversal is performed on a valid binary tree (lemma 37). Since the choice of the the child is based on a standard tree traversal the operation must still be on the correct path.
  2. The operation can move from a node with *removed* = true. By the assumption given by induction the $n-1^{th}$ node has a path to the range of keys that includes k. From the $n-1^{th}$ node either the right or left child must be chosen. The node is chosen based on a standard tree traversal with one exception: If the node has the same key k that is being searched for and the node was removed by a left rotation then the right child is chosen (lines 5.3.2–5.3.2). With this exception then in all cases if the node is not $\perp$ then by lemma 47 it must have a path to the same range as when it

has *removed* = false, which includes k (the $n-1^{th}$ node's range includes k so the n^{th} node's range must also). Otherwise if one child is $\perp$ then the other child is chosen (lines 5.3.2–5.3.2), which must have a path to a range at least as large as the $n-1^{th}$ node by lemma 47 (which includes k).

□

Theorem 8 *An* insert *operation is valid.*

Proof. The insert operation starts by performing a find operation (line 5.2). From lemma 48 we know that the find operation will return the correct place. There are two possibilities, either the find operation returns a node in the tree with key k (line 5.3.2), or it returns a node in the tree that has $\perp$ for the child where k must be inserted (line 5.3.2).

First consider the case where a node with key k is returned. The transactional read on *removed* (line 5.3.2 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 5.2) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns false then the insert operation can return false because the node exists in the set. Otherwise if the read on *deleted* returns true then a transaction performs a transactional write setting *deleted* to false (line 5.2). The transactional read on *deleted* ensures that k is not in the set before the transaction commits. The transactional write on *deleted* ensures that k is in the set after the transaction commits.

Second consider the case where a node with $key \neq k$ is returned. On lines 5.3.2–5.3.2 of the find operation a transactional read has been done on the child pointer of this node, returning $\perp$, which must be valid throughout the transaction. The transactional read on *removed* (line 5.3.2 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. By lemma 37 this node belongs to a correct binary tree, this along with lemma 48 ensures that the child of this node is the only place where a node with key k could exist (and since the value of the child pointer of $\perp$ a node with key k is not in the tree). A new node with key k is created and then added to the tree using a transactional write to the child pointer (lines 5.2–5.2). The *removed* and *deleted* fields of a newly created node can only be false so when the transaction commits the new node will be in the tree and k will be in the set. □

Theorem 9 *A* contains *operation is valid.*

Proof. The contains operation starts by performing a find operation (line 5.2). From lemma 48 we know that the find operation will return the correct place. There are two possibilities, either the find operation returns a node in the tree with key k (line 5.3.2), or it returns a node in the tree that has $\perp$ for the child where k could exist (line 5.3.2).

First consider the case where a node with key k is returned. The transactional read on *removed* (line 5.3.2 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 5.2) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns false then the *contains* operation can return true because the node exists in the set otherwise false can be returned because the node does not exist in the set.

Second consider the case where a node with $key \neq k$ is returned. On lines 5.3.2–5.3.2 of the find operation a transactional read has been done on the child pointer of this node, returning $\perp$,

and due to the transactional read the pointer must remain as $\perp$ throughout the transaction. The transactional read on *removed* (line 5.3.2 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. By lemma 37 this node belongs to a correct binary tree, this along with lemma 48 ensures that the child of this node is the only place where a node with key k could exist (and since the value of the child pointer is $\perp$ a node with key k is not in the tree). The contains operation can then return false. □

Theorem 10 *A* delete *operation is valid.*

Proof. The delete operation is almost the same as the contains operation. The only difference is in the case where a node with key k is returned from the find operation. The transactional read on *removed* (line 5.3.2 of the find operation) ensures that the node will be in the tree throughout the duration of the transaction. Next a transactional read is done on the node's *deleted* variable (line 5.2) ensuring that this value will remain the same throughout the transaction. If the read on *deleted* returns true then the delete operation can return false because the node does not exist in the set. Otherwise k is in the set and a transactional write is performed setting the nodes *deleted* variable to true. Since *removed* is false this node is part of the valid tree so it is the only place where key k can be so by setting *deleted* to true k must not be in the set. □

5.5 Experimental Evaluation

We experimented our library by integrating it in (i) a micro-benchmark of the synchrobench suite to get a precise understanding of the performance causes and in (ii) the tree-based vacation reservation system of the STAMP suite and whose runs sometimes exceed half an hour. The machine used for our experiments is a four AMD Opteron 12-core Processor 6172 at 2.1 Ghz with 32 GB of RAM running Linux 2.6.32, thus comprising 48 cores in total.

5.5.1 Testbed choices

We evaluate our tree against well-engineered tree algorithms especially dedicated to transactional workloads. The red-black tree is a mature implementation developed and improved by expert programmers from Oracle Labs and others to show good performance of TM in numerous papers [6, 42, 61, 70, 36, 82, 65]. The observed performance is generally scalable when contention is low, most of integer set benchmarks on which they are tested consider the ratio of attempted updates instead of effective updates. To avoid the misleading (attempted) update ratios that capture the number of calls to potentially updating operations, we consider the *effective* update ratios of synchrobench counting only modifications and ignoring the operations that fail (e.g., remove may fail in finding its parameter value thus failing in modifying the data structure).

The AVL tree we evaluate (as well as the aforementioned red-black tree) is part of STAMP [61]. As mentioned before one of the main refactoring of this red-black tree implementation is to avoid the use of sentinel nodes that would produce false-conflicts within transactions. This improvement could be considered a first-step towards obtaining a speculation-friendly binary search tree, however, the modification-restructuring, which remains tightly coupled, prevents scalability to high levels of parallelism.

To evaluate performance we ran the micro-benchmark and the vacation application with 1, 2, 4, 8, 16, 24, 32, 40, 48 application threads. For the micro-benchmark, we averaged the data

over three runs of 10 seconds each. For the vacation application, we averaged the data over three runs as well but we used the recommended default settings and some runs exceeded half an hour because of the amount of transactions used. We carefully verified that the variance was sufficiently low for the result to be meaningful.

5.5.2 Biased workloads and the effect of restructuring

In this section, we evaluate the performance of our speculation-friendly tree on an integer set micro-benchmark providing remove, insert, and contains operations, similarly to the benchmark used to evaluate state-of-the-art TM algorithms [36, 66, 63]. We implemented two set libraries that we added to the synchrobench distribution: our non-optimized speculation-friendly tree and a baseline tree that is similar but never rebalances the structure whatever modifications occur. Figure 5.7 depicts the performance obtained from four different binary search trees: the red-black tree (RBtree), our speculation-friendly tree without optimizations (SFtree), the no-restructuring tree (NRtree) and the AVL tree (AVLtree).

The performance is expressed as the number of operations executed per microsecond. The update ratio varies between 5% and 20%. As we obtained similar results with 2^{10}, 2^{12} and 2^{14} elements, we only report the results obtained from an initialized set of 2^{12} elements. The biased workload consists of inserting (resp. deleting) random values skewed towards high (resp. low) numbers in the value range: the values always taken from a range of 2^{14} are skewed with a fixed probability by incrementing (resp. decrementing) with an integer uniformly taken within $[0..9]$.

On both the normal (uniformly distributed) and biased workloads, the speculation-friendly tree scales well up to 32/40 threads. The no-restructuring tree performance drops to a linear shape under the biased workload as expected: as it does not rebalance, the complexity increases with the length of the longest path from the root to a leaf that, in turn, increases with the number of performed updates. In contrast, the speculation-friendly tree can only be unbalanced during a transient period of time which is too short to affect the performance even under biased workloads.

The speculation-friendly tree improves both the red-black tree and the AVL tree performance by up to $1.5\times$ and $1.6\times$, respectively. The speculation-friendly tree is less prone to contention than AVL and red-black trees, which both share similar performance penalties due to contention.

5.5.3 Portability to other TM algorithms

The speculation-friendly tree is an inherently efficient data structure that is portable to any TM systems. It fulfills the TM interface standardized in [71] and thus does not require the presence of explicit escape mechanisms like early release [42] or snap [62] to avoid extra TM bookkeeping (our uread optimization being optional). Nor does it require high-level conflict detection, like open nesting [75, 76, 56] or transactional boosting [70]. Such improvements rely on explicit calls or user-defined abstract locks, and are not supported by existing TM compilers [71] which limits their portability. To make sure that the obtained results are not biased by the underlying TM algorithm, we evaluated the trees on top of $\mathscr{E}$-STM [66], another TM library (on a 2^{16} sized tree where $\mathscr{E}$-STM proved efficient), and on top of a different TM design from the one used so far: with eager acquirement.

The obtained results, depicted in Figure 5.8 look similar to the ones obtained with TinySTM-CTL (Figure 5.7) in that the speculation-friendly tree executes faster than other trees for all TM

settings. This suggests that the improvement of speculation-friendly tree is potentially independent from the TM system used. A more detailed comparison of the improvement obtained using elastic transactions on red-black trees against the improvement of replacing the red-black tree by the speculation-friendly tree is depicted in Figure 5.9(a). It shows that the elastic improvements (15% on average) is lower than the speculation-friendly tree one (22% on average, be it optimized or not).

5.5.4 Reusability for specific application needs

We illustrate the reusability of the speculation-friendly tree by composing remove and insert from the existing interface to obtain a new atomic and deadlock-free move operation. Reusability is appealing to simplify concurrent programming by making it modular: a programmer can reuse a library without having to understand its synchronization internals. While reusability of sequential programs is straightforward, concurrent programs can generally be reused only if the programmer understands how each element is protected. For example, reusing a library can lead to deadlocks if shared data are locked in a different order than what is recommended by the library. Additionally, a lock-striping library may not conflict with a concurrent program that locks locations independently even though they protect common locations, thus leading to inconsistencies.

Figure 5.9(b) indicates the performance on workloads comprising 90% of read-only operations (including contains and failed updates) and 10% move/insert/delete effective update operations (among which from 1% to 10% are move operations). The performance decreases as more move operations execute, because a move protects more elements in the data structure than a simple insert or delete operation and during a longer period of time.

5.5.5 The vacation travel reservation application

We experiment our optimized library tree with a travel reservation application from the STAMP suite [61], called vacation. This application is suitable for evaluating concurrent binary search tree as it represents a database with four tables implemented as tree-based directories (cars, rooms, flights, and customers) accessed concurrently by client transactions.

Figure 5.10 depicts the execution time of the STAMP vacation application building on the Oracle red-black tree library (by default), our optimized speculation-friendly tree, and the baseline no-restructuring tree. We added the speedup obtained with each of these tree libraries over the performance of bare sequential code of vacation without synchronization. (A concurrent tree library outperforms the sequential tree when its speedup exceeds 1.) The chosen workloads are the two default configurations ("low contention" and "high contention") taken from the STAMP release, with the default number of transactions, 8× more transactions than by default and 16× more, to increase the duration and the contention of the benchmark without using more threads than cores.

Vacation executes always faster on top of our speculation-friendly tree than on top of its built-in Oracle red-black tree. For example, the speculation-friendly tree improves performance by up to 1.3× with the default number of transactions and to 3.5× with 16× more transactions. The reason of this is twofold: (i) In contrast with the speculation-friendly tree, if an operation on the red-black tree traverses a location that is being deleted then this operation and the deletion conflict. (ii) Even though the Oracle red-black tree tolerates that the longest path from the root to a leaf can be twice as long as the shortest one, it triggers the rotation immediately after this

threshold is reached. By contrast, our speculation-friendly tree keeps checking the unbalance to potentially rotate in the background. In particular, we observed on 8 threads in the high contention settings that the red-black tree vacation triggered around 130,000 rotations whereas the speculation-friendly vacation triggered only 50,000 rotations.

Finally, we observe that vacation presents similarly good performance on top of the no-restructuring tree library. In rare cases, the speculation-friendly tree outperforms the no-restructuring tree probably because the no-restructuring tree does not physically remove nodes from the tree, thus leading to a larger tree than the abstraction. Overall, their performance is comparable. With 16× the default number of transactions, the contention gets higher and rotations are more costly.

5.6 Related Work

Aside from the optimistic synchronization context, various relaxed balanced trees have been proposed. The idea of decoupling the update and the rebalancing was originally proposed by Guibas and Sedgewick [68] and was applied to AVL trees by Kessels [72], and Nurmi, Soisalon-Soininen and Wood [78], and to red-black trees by Nurmi and Soisalon-Soininen [77]. Manber and Ladner propose a lock-based tree whose rebalancing is the task of separate maintenance threads running with a low priority [73]. Bougé et al. [59] propose to lock a constant number of nodes within local rotations. The combination of local rotations executed by different threads self-stabilizes to a tree where no nodes are marked for removal. The main objective of these techniques is still to keep the tree depth low enough for the lock-based operations to be efficient. Such solutions do not apply to speculative operations due to aborts.

Ballard [57] proposes a relaxed red-black tree insertion well-suited for transactions. When an insertion unbalances the red-black tree it marks the inserted node rather than rebalancing the tree immediately. Another transaction encountering the marked node must rebalance the tree before restarting. The relaxed insertion was shown generally more efficient than the original insertion when run with DSTM [42] on 4 cores. Even though the solution limits the waste of effort per aborting rotation, it increases the number of restarts per rotation. By contrast, our local rotation does not require the rotating transaction to restart, hence benefiting both insertions and removals.

Bronson et al. [60] introduce an efficient object-oriented binary search tree. The algorithm uses underlying time-based TM principles to achieve good performance, however, its operations cannot be encapsulated within transactions. For example, a key optimization of this tree distinguishes whether a modification at some node i grows or shrinks the subtree rooted in i. A conflict involving a growth could be ignored as no descendant are removed and a search preempted at node i will safely resume in the resulting subtree. Such an optimization is not possible using TMs that track conflicts between read/write accesses to the shared memory. This implementation choice results in higher performance by avoiding the TM overhead, but limits reusability due to the lack of bookkeeping. For example, a programmer willing to implement a size operation would need to explicitly clone the data structure to disable the growth optimization. Therefore, the programmer of a concurrent application that builds upon this binary search tree library must be aware of the synchronization internals of this library (including the growth optimization) to reuse it.

Felber, Gramoli and Guerraoui [66] specify the elastic transactional model that ignores false conflicts but guarantees reusability. In the companion technical report, the red-black tree library

from Oracle Labs was shown executing efficiently on top of an implementation of the elastic transaction model, $\mathscr{E}$-STM. The implementation idea consists of encapsulating the (i) operations that locate a position in the red-black tree (like insert, contains, delete) into an *elastic* transaction to increase concurrency and (ii) other operations, like size, into a regular transaction. This approach is orthogonal to ours as it aims at improving the performance of the underlying TM, independently from the data structure, by introducing relaxed transactions. Hence, although elastic transactions can cut themselves upon conflict detection, the resulting $\mathscr{E}$-STM, still suffers from congestion and wasted work when applied to non-speculation-friendly data structures. The results presented in Section 5.5.3 confirm that the elastic speedup is even higher when the tree is speculation-friendly.

5.7 Conclusion

Transaction-based data structures are becoming a bottleneck in multicore programming, playing the role of synchronization toolboxes a programmer can rely on to write a concurrent application easily. This work is the first to show that speculative executions require the design of new data structures. The underlying challenge is to decrease the inherent contention by relaxing the invariants of the structure while preserving the invariants of the abstraction.

In contrast with the traditional pessimistic synchronization, the optimistic synchronization allows the programmer to directly observe the impact of contention as part of the step complexity because conflicts potentially lead to subsequent speculative re-executions. We have illustrated, using a binary search tree, how one can exploit this information to design a speculation-friendly data structure. The next challenge is to adapt this technique to a large body of data structures to derive a speculation-friendly library.

Source Code

The code of the speculation-friendly binary search tree is available at **http://lpd.epfl.ch/gramoli/php/synchrobench.php**http://lpd.epfl.ch/gramoli/php/synchrobench.php.

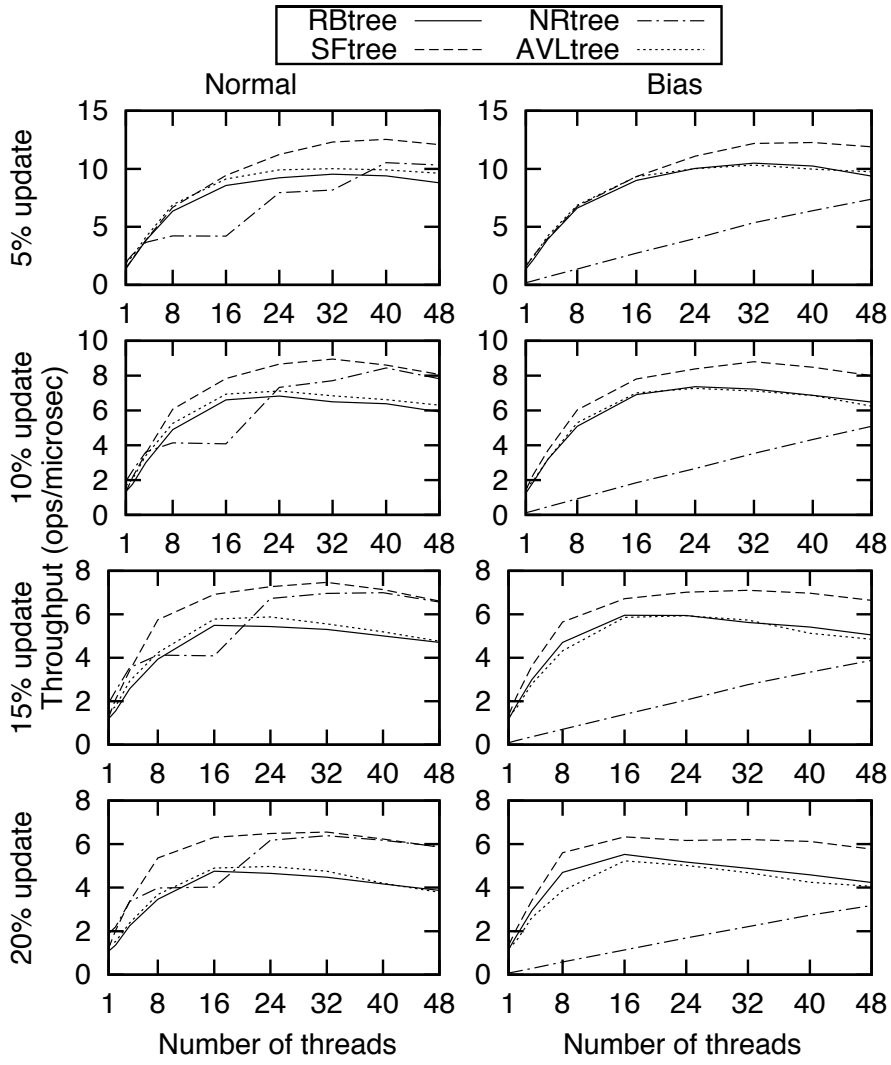


Figure 5.7: Comparing the AVL tree (AVLtree), the red-black tree (RBtree), the no-restructuring tree (NRtree) against the speculation-friendly tree (SFtree) on an integer set micro-benchmark with from 5% **(top)** to 20% updates **(bottom)** under normal **(left)** and biased **(right)** workloads

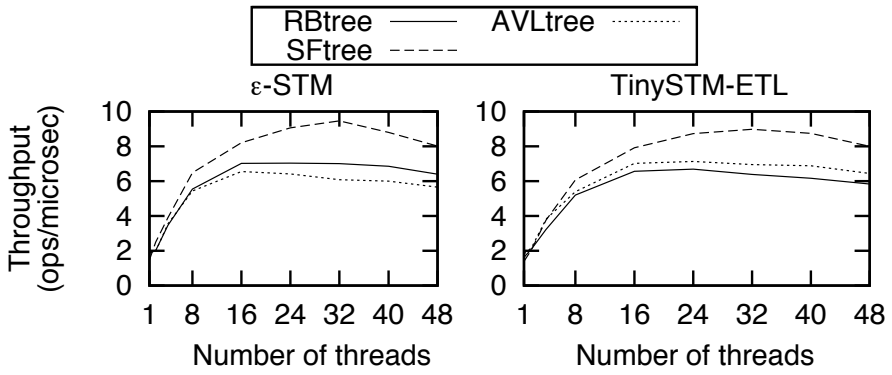
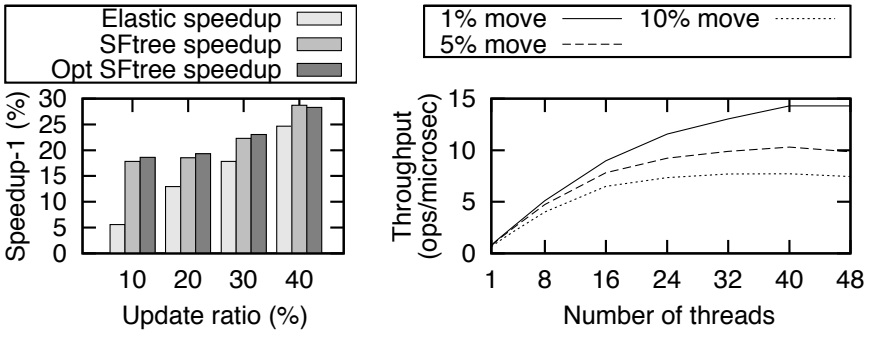


Figure 5.8: The speculation-friendly library running with **(left)** another TM library ($\mathscr{E}$-STM) and with **(right)** the previous TM library in a different configuration (TinySTM-ETL, i.e., with eager acquirement)



(a) Elastic transaction speedup vs. speculation-friendly tree speedup

(b) Performance when reusing the speculation-friendly tree

Figure 5.9: Elastic transaction comparison and reusability

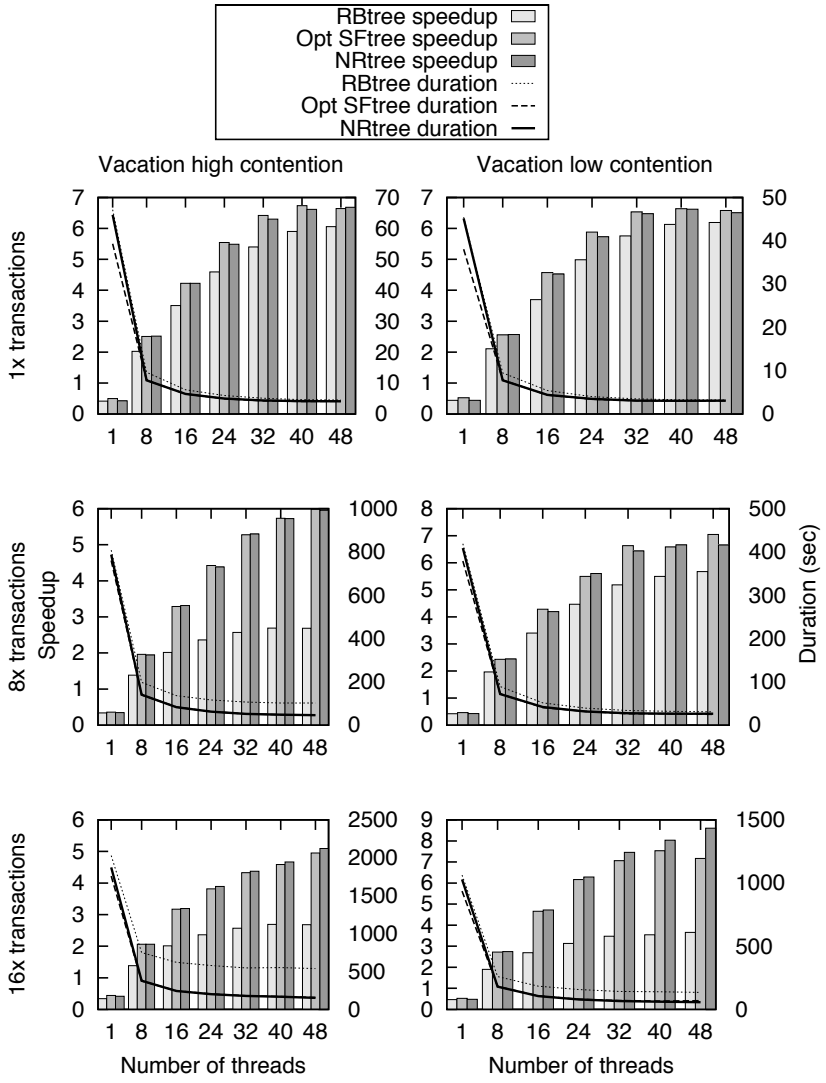


Figure 5.10: The speedup (over single-threaded sequential) and the corresponding duration of the vacation application built upon the red-black tree (RBtree), the optimized speculation-friendly tree (Opt SFtree) and the no-restructuring tree (NRtree) on **(left)** high contention and **(right)** low contention workloads, and with **(top)** the default number of transaction, **(middle)** 8× more transactions and **(bottom)** 16× more transactions

Chapter 6

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] Victor Bushkov, Rachid Guerraoui, and MichaÅĆ KapaÅĆka. On the liveness of transactional memory. *Proc. 7th Int'l ACM symposium on Principles of distributed computing (PODC '12)*. ACM, New York, USA, Pages 9-18 2012.

[131] 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.

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

[133] 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.

[134] 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.

[135] 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.

[136] 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.

[137] 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.

[138] 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.

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

[140] 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