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 :

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

Abstract

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

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

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

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

Acknowledgments

Contents

List of Figures

Part I

Software Transactional Memory Algorithms

Chapter 1

Introduction

1.1 Stuff for intro

More explicitly, an STM is a middleware approach that provides the programmers with the *transaction* concept [**?**, **?**]. This concept is close but different from the notion of transactions encountered in databases [**?**, **?**, **?**]. A process is designed as (or decomposed into) a sequence of transactions, each transaction being a piece of code that, while accessing any number of shared objects, always appears as being executed atomically. The job of the programmer is only to define the units of computation that are the transactions. He does not have to worry about the fact that the objects can be concurrently accessed by transactions. Except when he defines the beginning and the end of a transaction, the programmer is not concerned by synchronization. It is the job of the STM system to ensure that transactions execute as if they were atomic.

Of course, a solution in which a single transaction executes at a time trivially implements transaction atomicity but is irrelevant from an efficiency point of view. So, a STM system has to do "its best" to execute and commit as many transactions per time unit as possible. Similarly to a scheduler, a STM system is an on-line algorithm that does not know the future. If the STM is not trivial (i.e., it allows several transactions that access the same objects in a conflicting manner to run concurrently), this intrinsic limitation can direct it to abort some transactions in order to ensure both transaction atomicity and object consistency. From a programming point of view, an aborted transaction has no effect (it is up to the process that issued an aborted transaction to re-issue it or not; usually, a transaction that is restarted is considered a new transaction). Abort is the price that has to be paid by transactional systems to cope with concurrency in absence of explicit synchronization mechanisms (such as locks or event queues).

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

tion, according to the abstraction level supplied to the programmer, a lock may be encapsulated into a linguistic construct such as a *monitor* [**?**] or a *serializer* [**?**].

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

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

The Software Transactional Memory approach The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory was first proposed (nearly twenty years ago!) by Herlihy and Moss to implement concurrent data structures [**?**]. It has then been implemented in software by Shavit and Touitou [**?**] and has recently gained great momentum as a promising alternative to locks in concurrent programming [**?**, **?**, **?**, **?**]. 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* [**?**] that has rang the revival of concurrent programming.

Transactional memory abstracts away the complexity associated with concurrent programming by replacing locking with atomic execution units. 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 generally, STM is a middleware approach that provides the programmers with the *transaction* concept (this concept is close but different from the notion of transactions encountered in database systems [**?**]). A process is designed as (or decomposed into) a sequence of transactions, with each transaction being a piece of code that, while accessing concurrent objects, always appears as if it was executed atomically[1]. The job of the programmer is only to state which units of computation have to be atomic. He does not have to worry about the fact that the objects accessed by a transaction can be concurrently accessed. The programmer is not concerned by synchronization except when (s)he defines the beginning and the end of a transaction. It is then the job of the STM system to ensure that transactions are executed as if they were atomic.

Let us observe that the "spirit/design philosophy" that has given rise to STM systems is not new: it is related to the notion of *abstraction level*. More precisely, the aim is to allow

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

the programmer to focus and concentrate only on the problem (s)he has to solve and not on the base machinery needed to solve it. As we can see, this is the approach that has replaced assembly languages by high level languages and programmer-defined garbage collection by automatic garbage collection. STM can be seen as a new concept that takes up this challenge when considering synchronization issues.

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

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

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

The Software Transactional Memory approach The concept of *Software Transactional Memory* (STM) is an answer to the previous challenge. The notion of transactional memory was first proposed (nearly twenty years ago!) by Herlihy and Moss to implement concurrent data structures [**?**]. It has then been implemented in software by Shavit and Touitou [**?**] and has recently gained great momentum as a promising alternative to locks in concurrent programming [**?**, **?**, **?**, **?**]. 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* [**?**] that has rang the revival of concurrent programming.

Transactional memory abstracts away the complexity associated with concurrent programming by replacing locking with atomic execution units. 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 generally, STM is a middleware approach that provides the programmers with the *transaction* concept (this concept is close but different from the notion of transactions encountered in database systems [**?**]). 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[2]. The job of the programmer is only to state which units of computation have to be atomic. He does not have to worry about the fact that the objects accessed by a transaction can be concurrently accessed. The programmer is not concerned by synchronization except when (s)he defines the beginning and the end of a transaction. It is then the job of the STM system to ensure that transactions are executed as if they were atomic.

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

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

1.2 STM computation model and base definitions

1.2.1 Processes and atomic shared objects

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

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

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

1.2.2 Transactions and object operations

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

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

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

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

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

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

Incremental snapshot As in [**?**], 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

Chapter 3

Universal Constructions and Transactional Memory

3.1 Introduction

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

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

3.1.1 Progress properties

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

3.1.1.1 Blocking properties

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

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

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

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

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

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

3.1.1.2 Non-blocking properties

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

between threads is necessary. Generally programming using these operations in this way is considered to be much more difficut then programming using locks.

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

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

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

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

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

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

3.1.2 Universal Constructions for concurrent objects

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

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

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

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

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

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

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

One intresting example related to the work in this thesis is a universal construction for wait-free *transaction friendly* concurrent objects presented in [**?**]. The words "transaction friendly" means here that a process that has invoked an operation on an object can abort it during its execution. Hence, a 'transaction friendly" concurrent object is a kind of abortable object. It is important to notice that this abortion notion is different from the notion of transaction abortion. In the first case, the abort of an operation is a programming level notion that the construction

has to implement. Differently, in the second case, a transaction abort is due the implementation itself. More precisely, transaction abortion is then a system level mechanism used to prevent global inconsistency when the system allows concurrent transactions to be executed optimistically (differently, albeit very inefficient, using a single global lock for all transactions would allow any transaction to executed without being aborted).

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

3.1.3 Transactional memory and universal constructions

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

3.1.4 Progress properties and transactional memory

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

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

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

We can then simply extend these properties by adding additional requirements for the committing of transactions. For example we might want a lock-free transactional memory protocol to ensure that at least one of the live transactions in the system will eventually commit. A more detailed analysis of non-blocking progress properties and transactional memory has been performed in []. In this work they define the non-blocking properties *solo-progress* as a equivalent to the obstruction-freedom property for transactional memory and *local-progress* as a equivalent of wait-freedome for transactional memory. Informally a protocal that satisfies solo-progress must ensure that every process they executes for long enough must make progress (where progress requires eventually committing some live transaction) while a protocol satisfying local-progress must ensure that "every process that keeps executing a transaction (say keeps retrying it in case

it aborts) eventually commits it." Additionaly in [] they examine how these properties can be appiled in faulty and faut-free systems with or without parisitic transactions (a parisitic transaction is one which is continually executed, but never tries to commit). They show that local-progess is impossible in a faulty system where each transaction is fixed to a certain process. An extended discussion on the possible/impossible liveness properties of a STM system is presented by the same authors in [**?**] where it is also described a general lock-free STM system. This system is based on a mechanism similar to the Compare&Swap used in this paper.

3.1.5 Previous approaches

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

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

Contention Management The idea is here to keep track of conflicts between transactions and have a separate entity, usually called *contention manager*, decide what action to take (if any). Some of these actions include aborting one or both of the conflicting transactions, stalling one of the transactions, or doing nothing. The idea was first (as far as we know) proposed in the dynamic STM system (called DSTM) [**?**] 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 [**?**]. Failure detector-based contention managers (and corresponding lower bounds) are described in [**?**]. 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 [**?**].

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

issued this transaction as well as the process that issued the transaction with the oldest timestamp.

Unfortunately even though such contention managers exist that ensure all transactions commit, most STM implementations do not use them in the interest of performance and as a result provide less strong progress guarentees. As a solution to avoid these performance problems while still eventually providing strong progress, some modern STM's (e.g., for example TinySTM [**?**] or SwissSTM [**?**]) 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 [**?**].

Another interesting approach is called *steal-on-abort* [**?**]. Its base principle is the following one. If a transaction $T1$ is aborted due to a conflict with a transaction $T2$, $T1$ is assigned to the processor that executed $T2$ in order to prevent a new conflict between $T1$ and $T2$. Interestingly in order to help a transaction commit, this scheduler allows a transaction to be executed and committed by a processor different the one it originated from. Like contention managers, these schedulers can provide progress, but none of them ensure the progress of a process with a transaction that conflicts with some other transaction which has reached a point at which it must not be aborted. This means that the progress of a process still depends on the progress of another process.

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

It is interesting to note that (a) an irrevocable transaction must be run exactly once and (b) only one irrevocable transaction can be executed at a time in the system (unless the shared memory accesses of the transaction are known ahead of time). This priority given to the running irrevocable transaction allows it to guarantee to succed, but does so at the cost of preventing other transactions from progressing until it finishes. STM protocols supporting ireovokable transactions are proposed and discussed in [**?**] and [**?**]. Irrevocable transactions suited to deadline-aware scheduling are presented in [**?**].

Robust STMs Ensuring progress even when bad behavior (such as process crash) can occur has been investigated in several papers. As an example, [**?**] 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., [**?**, **?**]), while others are lock-free (in the sense there is no deadlock) [**?**].

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

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

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

3.1.6 Ensuring transaction completion

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

More precisely we want a non-blocking STM protocol which ensures that every transaction issued by a process is eventually committed wheres its progress only depends on the issuing process. Then, the job of a programmer is to write her/his concurrent program in terms of cooperating sequential processes, each process being made up of a sequence of transactions (plus possibly some non-transactional code). At the programming level, any transaction invoked by a process is executed exactly once (similarly to a procedure invocation in sequential computing). Moreover, from a global point of view, any execution of the concurrent program is linearizable [**?**], meaning that all the transactions appear as if they have been executed one after the other in an order compatible with their real-time occurrence order. Hence, from the programmer point of view, the progress condition associated with an execution is a very classical one, namely, starvation-freedom.

The remainder of this chapter focuses on the design of a such protocol that. (As a note it should be said that the presentation of previous approches that deal with aborted transactions is

not entirely fair as they are primarily efficiency-oriented while our construction is more theory-oriented.)

3.1.7 A short comparison with object-oriented universal construction

The first point to notice is that an STM that hides the concept of commit/abort from the programmer has objectives similar to that of a universal consturction described earier in this chapter. A universal construction allows a programmer to turn a sequential object into a concurrent one where each operation is linearizable and completes successfully, while the STM protocol we want here is one that allows a programmer to place transactions in his code each of which are executed successfully and are 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. Albeit the operations on a queue can have many different implementations, their semantics is defined once for all. Differently, each transaction is a specific atomic procedure whose code can be seen as being any dynamically defined code. What is significant here is that any transaction is able to read and write to any location in shared memory while an operation in a universal construction is fixed to a predefined set, which is often a single instance of a data structure. Simply put, a programmer might want to use a universal construction when he has an object with predefined self-contained operations that he wants to use concurrently (such as a data structure) while transactions might be more suitable when the programmer wants to perform general atomic operations within his code.

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

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

Another interesting feature of the proposed construction (which is specifically designed for STM programs) is the systematic use of speculative execution. Even if efficiency is not a first class requirement addressed in our work, the notion of a speculative execution can be a basis for future work on universal construction that will focus on efficiency. As discussed in section **??**, a traditional universal construction can expect best case performance to be equal to that of

a sequental implementation. While in the case of the STM's speculative execution any number of transactions are able to execute concurrently (and successfully if no conflict is found) by performing validations on their read sets. In some ways, the computation cost of performing validation can be seen as a trade off in order to allow for higher concurrency.

3.2 A universal construction for transaction based programs

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

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

3.3 Computation models

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

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

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

3.3.1 The user programming model

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

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

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

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

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

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

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

3.3.2 The underlying system model

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

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

Compare&swap register A compare&swap register X is an atomic object that provides processors with a single operation denoted X.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.

3.4 A universal construction for STM systems

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

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 [**?**].

3.4.1 Control variables shared by the processors

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

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

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

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

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

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

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

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

Helping mechanism: logical time *CLOCK* is an atomic fetch&increment register initialized to 0. It is used by the helping mechanism to associate a logical date with a transaction that has to be helped. Dates define a total order on these transactions. They are used to ensure that any helped transaction is eventually committed.
Helping mechanism: the array $STATE[1..n]$ This array is such that $STATE[i]$ describes the current state of the execution (simulation) of process p_i. It has four fields.

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

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

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

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

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

3.4.3 Behavior of a processor: initialization

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

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

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

Figure 3.1: Initialization for processor P_x ($1 \leq x \leq m$)

3.4.4 Behavior of a processor: main body

The behavior of a processor P_x is described in Figure 3.2. This consists of a while loop that terminates when all transactions issued by the processes owned by P_x have been successfully executed. This behavior can be decomposed into 4 parts.

Select the next transaction to execute (Lines 01-12) Processor P_x first reads (asynchronously) the current progress of each process and selects accordingly a process (lines 01-02). The procedure 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).

3.4.5 Behavior of a processor: starvation prevention

Any transaction issued by a process has to be eventually executed by a processor and committed. To that end, the helping mechanism introduced previously has to be enriched so that no processor either (a) permanently helps only processes owned by other processors or (b) loops forever in an internal while loop (lines 06-12 or 20-25). The first issue is solved by procedure select_next_process() while the second issue is solved by the procedure prevent_endless_looping().

Each of these procedures uses an integer value (resp., $K1$ and $K2$) as a threshold on the length of execution periods. These periods are measured with counters (resp., $k1\_counter$ and $k2\_counter$). When one of these periods attains its threshold, the corresponding processor requires help for its pending transaction. The values $K1$ and $K2$ can be arbitrary.

The procedure select_next_process() This operation is described in Figure 3.3. It is invoked at line 02 of the main loop and returns a process identity. Its aim is to allow the invoking processor P_x to eventually make progress for each of the processes it owns.

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

This is realized as follows. P_x first computes the set *set* of processes p_i for which help has been required (those are the processes whose help date is $\neq +\infty$) and, (as witnessed by the array *LAST_CMT*) either no processor has yet publicized the fact that their last transactions have been committed or p_i is owned by P_x (line 101). If *set* is empty (no help is required), select_next_process() returns the identity of the next process owned by P_x (line 103). If $set \neq \emptyset$, there are processes to help and P_x selects the identity i of the process with the oldest help date (line 104). But before returning the identity i (line 119), P_x checks if it has been waiting for a too long period before having its next transaction executed. There are then two cases.

- If $i \in OWNED\_BY[x]$, P_x has already required help for the process p_i for which it strives to make progress. It then resets the counter $k1\_counter$ to 0 and returns the identity i (line 106).
- If $i \notin OWNED\_BY[x]$, P_x first increases $k1\_counter$ (line 107) and checks if it attains its threshold $K1$. If this is the case, the logical period of time is too long (line 109) and consequently (if not yet done) P_x requires help for the last transaction of the process p_j (such that $my\_next\_proc = j$). As we have seen, "require help" is done by assigning the next clock value to $STATE[j].help\_date$ (lines 109-114). In that case, P_x also resets $k1\_counter$ to 0 (line 115).

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

Figure 3.3: The procedure select_next_process()

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

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

Figure 3.4: Procedure prevent_endless_looping()

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

The time period considered starts at the last committed transaction issued by a process owned by P_x. It is measured by the number of transactions committed since then. The beginning of this time period is determined by P_x's local pointer *my_last_cmt* (which is initialized to *FIRST* and updated at line 45 of the main loop after the last transaction of a process owned by P_x has been committed.)

The relevant time period is measured by processor P_x with its local variable $k2\_counter$. If the process p_i currently selected by select_next_process() is owned by P_x (line 251), then P_x will require help for p_i once this period attains $K2$ (lines 253-256). In that way, the transaction issued by that process will be executed and committed by other processors and (if not yet done)

this will allow P_x to exit the while loop because its local boolean variable *committed* will then become true (line 09 of the main loop).

3.5 Proof of the STM construction

Let *PROG* be a transaction-based n-process concurrent program. The proof of the universal construction consists in showing that a simulation of *PROG* by m processors that execute the algorithms described in Figures 3.1-3.4 generates an execution of *PROG*.

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

Proof Let us assume by contradiction that there is a time after which either P_y is blocked within an internal while loop (Figure 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. $\square_{Lemma\ 1}$

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

Proof Let us first observe that all transactions that require help have bounded and different help dates (lines 37-38, 111-112 or 255-256). Moreover, once defined, the helping date for a transaction is not modified.

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

As T has the smallest help date, it follows from Lemma 1 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. $\square_{Lemma\ 2}$

Lemma 3 *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 251-253 and the current value of *my_last_cmt* (that points to the last committed transaction issued by a process owned by P_x, see line 45) that P_x's local variable $k2\_counter$ is increased infinitely often. Hence, eventually this number of invocations attains $K2$. When this occurs, if not yet done, P_x requires help for the transaction issued by p_i (lines 253-256). It then follows from Lemma 2, that p_i's transaction T is eventually committed. As the pointer *current* of P_x never skips a descriptor of the list and the list contains all and only committed transactions, we eventually have $(\downarrow current).tid = \langle i, i\_tr\_sn\rangle$ (where i_tr_sn is T's sequence number among the transactions issued by p_i). When this occurs, P_x's local variable *committed* is set to *true* and P_x stops looping in an internal while loop.

Let us now consider the case of a processor $P_y \neq P_x$. Let us first notice that the only way for P_y to execute T is when T has requested help (line 101 of operation select_next_process()). The proof follows from the fact that, due to Lemma 2, T is eventually committed. As previously (but now *current* is P_y's local variable), the predicate $(\downarrow current).tid = \langle i, i\_tr\_sn\rangle$ eventually becomes true and processor P_y sets *committed* to *true*. P_y then stops looping inside an internal while loop (line 08 or 23) which concludes the proof of the lemma. $\square_{Lemma\ 3}$

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

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

Let us first observe that, as p_i has issued T, P_x has executed line 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 2. Hence, to show that T is eventually committed, we show that, if P_x does not succeed

in committing T without help, it necessarily requires help for it. This follows from the code of the procedure select_next_proc(). There are two cases.

- select_next_process() returns i. In that case, as P_x does not loop forever in a while loop (Lemma 3), it eventually executes lines 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 3), it follows that it repeatedly invokes select_next_process() and, as these invocations do not return i, the counter $k1\_counter$ repeatedly increases and eventually attains the value $K1$. When this occurs P_x requires help for T (lines 107-116) and, due to Lemma 2, T is eventually committed.

Let us now observe that that, after T has been committed (by some processor), P_x executes lines 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. $\square_{Lemma\ 4}$

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

Proof Let T be a transaction committed by a processor P_y (i.e., the corresponding Compare&Swap() at line 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. $\square_{Lemma\ 5}$

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

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

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

Proof This lemma follows directly from lines 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). $\square_{Lemma\ 7}$

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

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

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

Proof To prove the lemma we have (a) to associate a linearization point with each transaction invocation, and (b) show that the corresponding sequence of linearization points is consistent, i.e., the values read from t-objects by a transaction invocation T are uptodate (there have not been overwritten). 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* 9

Lemma 10 *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 9, The transaction invocations issued by the processes are linearizable, from which we conclude that the set of t-objects (considered as a single concurrent object TO) is linearizable. Moreover, by definition, every nt-object is linearizable.

As (a)linearizability is a *local* consistency property [**?**][2] and (b) TO is linearizable and every nt-object is linearizable, it follows that the execution of the multiprocess program is linearizable.

□*Lemma* 10

Theorem 1 *Let PROG be a transaction-based n-process program. Any simulation of PROG by m processors executing the algorithms described in Figures 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 8 and Lemma 10. The execution of *PROG*

[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 [**?**] that linearizability is a local property.

is obtained by projecting the execution of each processor on the simulation of the transactions it commits and the execution of the non-transactional code of each process it owns. $\square_{Theorem\ 1}$

3.6 The number of tries is bounded

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

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

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

- This first location is in the prevent_endless_looping() procedure. In that case, the transaction for which help is required is the last transaction invoked by process $p_{my\_next\_proc}$.
- The second location is on line 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. $\square_{Lemma\ 11}$

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

Proof Let us first observe that a transaction T (invoked by a process p_i) is executed once before its help date is set to a finite value (if it is not committed after that execution). This is because only the owner P_x of p_i can select T (line 103) when its help date is $+\infty$. Then, after it has executed T unsuccessfully once, P_x requests help for T by setting its help date to a finite value (line 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 11 we know that there is a maximum of $m-1$ transactions that are not T that can be requesting help at this time and in order for them to commit before T they must have a help date smaller than T's. Also by lemma 5 we know that a transaction is committed exactly once so this conflict between $T1$ and T cannot occur again at $P2$. Now after committing $T1$, the next transaction (that asks for help) of a process that is owned by the same processor that owned $T1$ will have a larger help date than T so now there are only $m-1$ transactions that need help that could conflict with T. Repeating this we have at most $O(m)$ conflicts of this type for P.

- Case 3. The third case is that P's execution of T is aborted because some other process has already committed T. Then on line 08 P will see that T has been committed and not execute it again, so we have at most $O(1)$ conflicts of this type.

$\square_{Theorem\ 2}$

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

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

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

3.7 Conclusion

3.7.1 A short discussion

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

- The construction is for the family of transaction-based concurrent programs that are time-free (i.e., the semantics of which does not depend on real-time constraints).

- The construction is lock-free and works whatever the concurrency pattern (i.e., it does not require concurrency-related assumption such as obstruction-freedom). It works for both finite and infinite computations and does not require specific scheduling assumptions. Moreover, it is independent of the fact that processes are transaction-free (they then share only nt-objects), do not have non-transactional code (they then share only t-objects accessed by transactions) or have both transactions and non-transactional code.

- The helping mechanism can be improved by allowing a processor to require help for a transaction only when some condition is satisfied. These conditions could be general or application-dependent. They could be static or dynamic and be defined in relation with an underlying scheduler or a contention manager. The construction can also be adapted to benefit from an underlying scheduling allowing the owner of a process to be dynamically defined.

  It could also be adapted to take into account *irrevocable* transactions [**?**, **?**]. Irrevocability is an implementation property which can be demanded by the user for some of its transactions. It states that the corresponding transaction cannot be aborted (this can be useful when one wants to include inputs/outputs inside a transaction; notice that, in our model, inputs/outputs appear in non-transactional code).

- We have considered a failure-free system. It is easy to see that, in a crash-prone system, the crash of a processor entails only the crash of the processes it owns. The processes owned by the processors that do not crash are not prevented from executing. Furthermore due to the helping mechanism, once a process has asked for help with a transaction that transaction is guaranteed to commit as long as there exsits at least one live failure free process.

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

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

Chapter 4

Ensuring Strong Isolation in STM Systems

4.1 Introduction

STM Systems. Transactional Memory (TM) [**?**, **?**] has emerged as an attempt to allow concurrent programming based on sequential reasoning: By using TM, a user should be able to write a correct concurrent application, provided she can create a correct sequential program. The underlying TM system takes care of the correct implementation of concurrency. However, while most existing TM algorithms consider applications where shared memory will be accessed solely by code enclosed in a transaction, it still seems imperative to examine the possibility that memory is accessed both inside and outside of transactions.

Strong vs Weak Isolation. TM has to guarantee that transactions will be isolated from each other, but when it comes to transactions and non-transactional operations, there are two paths a TM system can follow: it may either act oblivious to the concurrency between transactions and non-transactional operations, or it may take this concurrency into account and attempt to provide isolation guarantees even between transactional and non-transactional operations. The first case is referred to as *weak isolation* while the second case is referred to as *strong isolation*. (This distinction of guarantees was originally made in [**?**], where reference was made to "weak atomicity" versus "strong atomicity".)

While weak isolation violates the isolation principle of the transaction abstraction, it could nevertheless be anticipated and used appropriately by the programmer, still resulting in correctly functioning applications. This would require the programmer to be conscious of eventual race conditions between transactional and non-transactional code that can change depending on the STM system used.

Desirable Properties. In order to keep consistent with the spirit of TM principles, however, 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 [**?**, **?**]. These are the reasons for which strong isolation is desirable. Under strong isolation, the aforementioned scenarios, where non-transactional operations violate transaction isolation, would not be allowed to happen. An intuitive approach to achieving strong isolation is to treat each non-transactional operation that accesses shared data as a "mini-transaction", i.e., one that

contains a single 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. However, while the concept of the memory transaction includes the possibility of abort, the concept of the non-transactional operation does not. This means that a programmer expects that a transaction might fail, either by blocking or by aborting. Non-transactional accesses to shared data, though, will usually be read or write operations, which the programmer expects to be atomic. While executing, a read or write operation is not expected to be de-scheduled, blocked or aborted.

Content of the Paper. This paper presents a TM algorithm which takes the previous issues into account. It is built on top of TM algorithm TL2 [**?**], a word-based TM algorithm that uses locks. More precisely, TL2 is modified to provide strong isolation with non-transactional read and write operations. However, the algorithm is designed without the use of locks for non-transactional code, in order to guarantee that their execution will always terminate. To achieve this, two additional functions are specified, which substitute conventional read or write operations that have to be performed outside of a transaction. Possible violations of correctness under strong isolation are reviewed in Sect. 4.2. The TL2 algorithm is described in Sect. 4.3. Section 4.4 describes the proposed algorithm that implements strong isolation for TL2, while Sect. 4.5 concludes the paper by summarizing the work and examining possible applications.

4.2 Correctness and Strong Isolation

Consistency Issues. Commonly, consistency conditions for TM build on the concept of *serializability* [**?**], a condition first established for the study of database transactions.

A concurrent execution of transactions is serializable, if there exists a serialization, i.e., a legal sequential execution equivalent to it. Serializability refers only to committed transactions, however, and fails to take into account the possible program exceptions that a TM transaction may cause - even if it aborts - when it observes an inconsistent state of memory.

Opacity [**?**], a stricter consistency condition for TM, requires that both committed as well as aborted transactions observe a consistent state of shared memory. This implies that in order for a concurrent execution of memory transactions to be opaque, there must exist an equivalent, legal sequential execution that includes both committed transactions and aborted transactions, albeit reduced to their read prefix. Other consistency conditions have also been proposed, such as *virtual world consistency* [**?**]. It is weaker than opacity while keeping its spirit (i.e., it depends on both committed transactions and aborted transactions).

Transaction vs Non-transactional Code. In a concurrent environment, shared memory may occasionally be accessed by both transactions as well as non-transactional operations. Traditionally, however, transactions are designed to synchronize only with other transactions without considering the possibility of non-transactional code; a program that accesses the same shared memory both transactionally and non-transactionally would be considered incorrect. A TM system that implements opacity minimally guarantees consistency between transactional accesses, however, consistency violations may still be possible in the presence of concurrent non-transactional code. Given this, it can still be acceptable to have concurrent environments that may be prone to some types of violations, as is the case with systems that provide weak isolation [**?**, **?**]. Under

weak isolation, transactional and non-transactional operations can be concurrent, but the programmer has to be aware of how to handle these. Interestingly, this possibility of *co-existence of two different paradigms* between strong and weak isolation 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.

Under weak isolation, transactions are considered to happen atomically only with respect to other transactions. 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. If this behavior is not considered acceptable for an application, then the responsibility to prevent it is delegated to the programmer of concurrent applications for this system. However, in order to spare the programmer this responsibility, both the transactional memory algorithm as well as the non-transactional read and write operations must be implemented in a way that takes their co-existence into account. Such an implementation that provides synchronization between transactional and non-transactional code is said to provide strong isolation.

Providing Strong Isolation. There are different definitions in literature for strong isolation [**?**, **?**, **?**]. In this paper we consider strong isolation to be the following: (a) non-transactional operations are considered as "mini" transactions which never abort and contain only a single read or write operation, and (b) the consistency condition for transactions is opacity.

This definition implies that the properties that are referred to as *containment* and *non-interference* [**?**] are satisfied. 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. An additional feature of strong isolation, implemented in this paper, is that non-transactional read and write operations never block or abort. For this reason, it is termed *terminating strong isolation*.

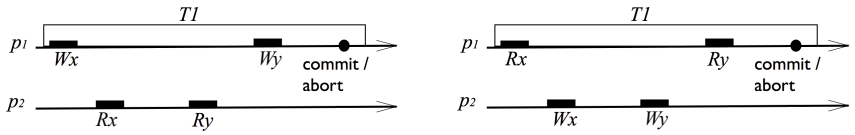

Figure 4.1: Left: *Containment* (operation R_x should not return the value written to x inside the transaction). Right: *Non-Interference* (wile it is still executing, transaction T_1 should not have access to the values that were written to x and y by process p_2).

Privatization/Publication. A discussion of the co-existence of transactional and non-transactional code would not be complete without mentioning the *privatization problem*. An

area of shared memory is privatized, when a process that modified it makes it inaccessible to other concurrent processes[1] with the purpose being that the process can then access the memory without using synchronization operations [**?**]. A typical example of privatization would be the manipulation of a shared linked list. The removal of a node by a transaction T_i, for private use, through non-transactional code, by the process that invoked T_i, constitutes privatization. Then, T_i is called privatizing transaction. While the privatization is not visible to all processes, inconsistencies may arise, given that for T_i's process the node is private but for other processes, the node is still seen as shared. Several solutions have been proposed for the privatization problem such as [**?**, **?**, **?**]. A system that provides *strong isolation* has the advantage of inherently also solving the privatization problem, because it inherently imposes synchronization between transactional and non-transactional code.

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.

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. Read-only transactions in TL2 are considered efficient, because they don't need to maintain local copies of a read or write set and because they need no final read set validation in order to commit. To control transaction synchronization, TL2 employs locks and logical dates.

Locks and Logical Date. A lock is associated with each shared variable. When a transaction attempts to commit it first has to obtain the locks of the variables of 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. 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*. When a transaction attempts to commit, it 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. A transaction must abort if its read set is not valid. 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.

[1] Conversely, a memory area is made public when it goes from being exclusively accessible by one process to being accessible by several processes [**?**] . This is referred to as the *publication problem* and the consistency issues that arise are analogous.

4.4 Implementing Terminating Strong Isolation

A possible solution to the problem of ensuring isolation in the presence of non-transactional code consists in using locks: 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. However, it can be argued that these characteristics are acceptable, given that the programmer accepts the fact that a transaction has a duration and that it may even fail: The fact that there is always a possibility that a transaction will abort means that the eventuality of failure to complete can be considered a part of the transaction concept.

On the contrary, when it comes to single read or write accesses to a shared variable, a non-transactional operation is understood as an event that happens atomically and completes. Unfortunately strong isolation implemented with locks entails the blocking of non-transactional read and write operations and would not provide termination.

Given that this approach would be rather counter-intuitive for the programmer (as well as possibly detrimental for program efficiency), the algorithm presented in this section provides a solution for adding strong isolation which is not based on locks for the execution of non-transactional operations. This algorithm builds on the base of TM algorithm TL2 and extends it in order to account for non-transactional operations. While 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), the proposed algorithm specifies non-transactional read and write operations that are to be used by the programmer, substituting conventional shared memory read and write operations. TM with strong isolation has also been proposed in software [**?**, **?**] in hardware [**?**], and has been suggested to be too costly [**?**]. This work differs from other implementations in that it is terminating and is implemented on top of a state-of-the-art STM in order to avoid too much extra cost.

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.

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. [2]

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.

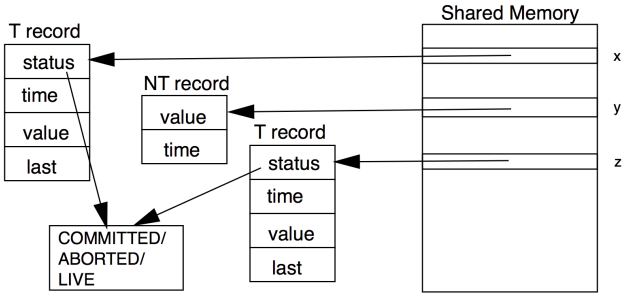

Figure 4.2: The memory set-up and the data structures that are used by the algorithm.

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

[2]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 $\uparrow (pt\downarrow) = pt$.

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. 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. This be done by hand or applied by a complier.

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

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

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

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

Next the process local variable *time* is advanced to the maximal value among its current value and the logical date of the T- or NT-record whose value was read. Finally if *time* was set

to ∞ on line 218 (meaning the T- or NT-record had yet to set its *time*), then it is updated to the *GCV* on line 219. The updated *time* value is used to prevent consistency violations. Once these book-keeping operations are finished, the local variable *value* is returned (line 220).

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

4.4.4 Transactional Read and Write Operations.

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

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

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

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

It should be noted that the reason why the checks are performed differently for NT-records and T-records is because the NT-write operations do not update the global clock value while update transaction do. This means that the checks must be more conservative in order to ensure correctness. If performing per value validation or restarting the transaction as an update transac-

tion is found to be too expensive, a third possibility would be to just increment the global clock, then restart the transaction as normal.

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

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

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

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

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

Begin and End of a Transaction The operations that begin and end a transaction are begin_transaction() and try_to_commit(), presented in Fig. 4.5. Local variables necessary for transaction execution are initialized by begin_transaction(). This includes *rv* which is set to *GCV* and, like in TL2, is used during transactional reads to ensure correctness, as well as *status* which is set to LIVE and the read and write sets which are initialized as empty sets. (lines 248-250).

After performing all required read and write operations, a transaction tries to commit, using the operation try_to_commit(). Similar to TL2, a try_to_commit() operation starts by trivially committing if the transaction was a read-only one (line 251) while an update transaction must announce to concurrent operations what locations it will be updating (the items in the write set). However, the algorithm differs here from TL2, given that it is faced with concurrent non-transactional operations that do not rely on locks and never block. This implies that even after

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

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

Figure 4.5: Transaction begin/commit.

acquiring the locks for all items in its write set, a transaction could be "outrun" by a non-transactional operation that writes to one of those items causing the transaction to be required to abort in order to ensure correctness. As described previously, while TL2 locks items in its write set using a lock array, this algorithm compare and swaps pointers directly to the T-records in its write set (lines 252-260) while keeping a reference to the previous value. The previous value is stored in the T-record before the compare and swap is performed (lines 255-256) with a failed compare and swap resulting in the abort of the transaction. If while performing these compare and swaps the transaction notices that another LIVE transaction is updating this memory, it aborts itself (line 255). By using these T-records instead of locks concurrent operations have access to necessary metadata used to ensure correctness.

The operation then advances the *GVC*, taking the new value of the clock as the logical time for this transaction (line 261). Following this, the read set of the transaction is validated for correctness (line 261). Once validation has been performed the operation must ensure that non of its writes have been concurrently overwritten by non-transactional operations (lines 262-265) if so then the transaction must abort in order to (line 264) to ensure consistency. During this check the transaction updates the *time* value of its T-records to the transactions logical time (line 263) similar to the way TL2 stores time values in the lock array so that future operations will know the serialization of this transaction's updates.

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

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

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

Figure 4.6: Transactional helper operations.

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

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

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

4.5 Conclusion

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

4.6 Version of algorithm that does not use NT-records

This algorithm also provides wait-free NT read and write operations. The difference is that NT-records are not used. Instead NT values are read and written directly from memory. By

doing this, memory allocations are not needed in NT writes and NT reads have one less level of indirection.

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

```
operation non_transactional_read(addr) is
(282)  tmp ← (↓ addr);
(283)  if ( tmp is of type T )
(284)    then if (tmp.status = LIVE)
(285)           then C&S(tmp.status, LIVE, ABORTED)
(286)         end if;
(287)         if (tmp.status = ABORTED)
(288)           then value ← tmp.last
(289)           else value ← tmp.value
(290)         end if;
(291)    else value ← tmp
(292)  end if;
(293)  return (value)
end operation.

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

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

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

This algorithm allows wait-free NT read operations. The only change that is needed to the base TL2 algorithm is that when an item is locked it points to the write-set of the transaction, and that each transaction has a marker that is initialized as *LIVE* and is set to *COMMITTED* just before the transaction starts performing write backs during the commit phase. The NT-read operation is shown in Fig. 4.10.

4.7.1 ***Proof of Linearizability***

4.8 ***An Implementation***

```
operation transactional_read(addr) is
(295)  if addr ∈ ws then return (item.value from addr in ws) end if;
(296)  tmp ← (↓ addr);
(297)  if (tmp is of type T )
(298)    then if (status = LIVE) then abort() end if;
(299)         if (tmp.time > rv) then abort() end if;
(300)         if (status = COMMITTED)
(301)           then value ← tmp.val
(302)           else value ← tmp.last
(303)         end if;
(304)    else
       % Do validation to prevent abort due to a non-transactional write
(305)         rv ← validate_by_value();
(306)         value ← tmp;
(307)  end if;
(308)  if this is an update transaction then add value to rs end if;
(309)  return (value)
end operation.

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

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

```
operation try_to_commit() is
(315)  if (ws = ∅) then return (COMMITTED) end if;
(316)  for each (item ∈ ws) do
(317)    tmp ← (↓ addr);
(318)    if (tmp is of type T)
(319)          then if ((status ← tmp.status) = COMMITTED)
(320)               then item.last ← tmp.value
(321)               else if (status = ABORTED) then item.last ← tmp.last
(322)               else abort()
(323)            end if;
(324)          else item.last ← tmp
(325)    end if;
(326)    if (¬C&S(item.addr, tmp, item)) then abort() end if;
(327)  end for;
(328)  time ← fetch&increment(GVC);
(329)  validate_by_value();
       % Ensure the writes haven't been overwritten by non-transactional writes
(330)  for each (item ∈ ws) do
(331)    if (item ≠ (↓ item.addr)) then abort() end if
(332)    item.time ← time;
(333)  end for;
(334)  if (C&S(status, LIVE, COMMITTED))
(335)    then return (COMMITTED)
(336)    else abort()
(337)  end if;
end operation.
```

Figure 4.9: Transaction commit.

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

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

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

Chapter 5

Contention Friendly Data Structures

Chapter 6

A Contention Friendly Binary Search Tree

Chapter 7

A Contention Friendly Skip-List

Chapter 8

A Contention Friendly Hash-Table

Conclusion

Chapter 9

Survey (to use for intro)

9.1 Transactional Memory

Transactional Memory was first introduced in 1993 [**?**] as a promising alternative to locks for concurrent programming. Since then it has been studied extensively and implemented in many different ways. One of the main reasons for its success as a research topic is that it abstracts away much of the difficulties that occur for programmers writing concurrent programs when using locks, while still promising the performance of fine grained locks. Even though it has been a popular research topic, it has yet to be widely implemented in practical use. I think? The basic concept behind transactional memory may be clear, but when it comes to actually creating and using an implementation, solving the details becomes very difficult. Such details include but are not limited to the following: How should the transactions be exposed to and interact with the programmer? What correctness conditions should transactions follow? How can transactions be implemented efficiently? Is it even possible to create a Transactional Memory that is efficient across all workloads? If not then what designs will be efficient for which workloads, or can the concept of the basic transaction be extened in order to improve performance? Much research has been done on all these topics and many different concepts of Transactional Memory have been considered, this survey gives an overview of some of the research that has been done on Transactional Memory, specifically looking at *Software Transactional Memory*.

9.1.1 What is transactional memory?

Transactions have been originally used in database systems in order to make sure concurrent operations follow desirable properties. Namely the ACID properties: *atomicity*, *consistency*, *isolation*, and *durability*. Transactional memory is similar to this, except instead of having transactions that *atomically* modify the state of the database, transactions are blocks of executable code running concurrently. In both database transactions and transactional memory much of the difficultly of accessing shared objects is abstracted away, making concurrent programming a much more manageable task.

In transactional memory a transaction is a block of code that appears to execute atomically within a multi-threaded application that operates on memory shared with other threads and transactions. To the programmer this might be similar to the idea of having a single lock that is shared throughout the program, whenever the programmer wants a section of code to operate atomically or without interference from other threads he can just encapsulate the block within this single

lock, this concept is called *single lock atomicity*,. So why not just program using a single lock? The problem with just using course grained locks like this, is that it inhibits concurrency resulting in poor performance. On the other hand fine grained locks can have good performance, but are difficult to program well, and can suffer from problems such as *deadlock*. The goal of transactional memory is to be simple for the programmer to understand similar to coarse grained locks, while having good performance similar to fine grained locks. The concept behind the implementation of transactional memory is that a transaction will either commit, meaning that it executed successfully and its changes to shared memory become visible to other transactions, or abort, meaning that it conflicted with a concurrent transaction and that none of its changes to shared memory will be visible, and it must retry by re-executing from the start of the block of code that makes up the transaction. Two transactions conflict when they are run concurrently and access the same memory location, with at least one of the accesses being a write. Aborting a transaction causes overhead, the work it has done so far must be completely redone, so most implementations try to abort transactions as little as possible. But just because two transaction conflict does not mean that one must be aborted. For example assume you have two transactions A and B, a shared memory location X, and the ordered history of events as follows: (Note that in this survey a single transaction's execution will be interpreted as a series of read and write operations ending in a commit (or abort))

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

Now as long as the transactions are ordered first A then B neither one must abort. As an example where two transactions must abort assume you have two transactions A and B, two shared memory locations X and Y, and the ordered history of events as follows:

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

Now the neither the order first A then B or first B then A is possible. A reads X before B writes X, so A must occur before B, and B writes Y before A reads Y, so B must occur before A, creating a contradiction. One of the transactions must be aborted.

Aborting transactions means work has been wasted, but keeping track of all the conflicts between the transactions is not easy and deciding when and if a transaction should be aborted is no simple task, in some cases it can even be more efficient to abort transactions more eagerly than to keep track of all conflicts. There is also liveness to consider, should a transaction be allowed to be repeatedly aborted forever by other transactions? And how should efficiency be measured? For example consider the workload that consists of a set of long running transactions that conflict with a set of short running transactions. Some implementation might allow the short transactions to commit, aborting the long transactions each time, while another system might allow the long transactions to commit, aborting the short transactions, while another might be somewhere in the middle. Should efficiency be measured by the number of transactions committed per unit time, or should it be measured by the total number of aborts, or some other measurement? In addition to this there are different consistency conditions to consider, in some conditions a transaction that is allowed to commit would have to abort in other conditions. Many of the solutions to these questions differ depending on application and workload, and there is often no obvious solution.

STM/HTM There are two main approaches for designing a Transactional Memory, *Hardware Transactional Memory* and *Software Transactional Memory*. Each has its strengths and weaknesses. Hardware Transactional Memory is implemented in the underlying architecture and can

be very efficient, but transactions are bounded in size and restricted to the capabilites of the hardware. Software Transactional memory can be implemented on existing hardware and can have transactions of arbitrary size, but often do not have as good performance as Hardware Transactional Memory. There has also been research done for creating hybrid models combining both hardware and software. This survey will focus on Software Transactional Memory. Articles giving an overview of both STM and HTM can be found in [**?**] and [**?**].

9.2 Characteristics of a STM (for a Programmer)

Deciding how a programmer should interact with an implementation of Software Transactional Memory has been a subject of interest and difficulty. This includes how the programmer creates the transactions themselves as well as how he accesses the shared memory.

9.2.1 Static vs Dynamic Transactions

A *static* transaction has its memory accesses predefined, before the code is run. Knowing what memory access the transaction is going to perform beforehand can help the implementation of the STM in terms of simplicity and performance, but is severely limiting to the programmer. With *if* statements, and loops the exact execution path of a program is usually unknown so it is not practical to have static transactions. The first STM implementation was static [**?**], and certain other static implementations have been designed in the interest of performance gains need a citation. Most STM implementations use *dynamic* transactions, allowing the memory access to be undefined until runtime.

9.2.2 Word Based

With *word based* Software Transactional Memory shared memory locations are accessed as specific words of memory. Many popular STMs are word based, such as TinySTM [**?**], SwissSTM [**?**], Elastic-STM [**?**], AVSTM [**?**], and TL2 [**?**]. An advantage of word based STMs is that all transactional memory access are easily abstracted into reads and writes to the words of memory which is useful for clearly designing and analyzing STM algorithms. Though is might no be clear to the programmer to access memory as words, especially in object oriented languages. The size of the shared memory access is of course important when considering performance, and in order to minimize cache misses the size of a cache-line has been proposed [**?**], but in [**?**] they believe a granularity size of four words gives optimal performance.

9.2.3 Object Based

Object based Software Transactional Memory is where shared memory locations are accessed by the programmer as objects. This is more natural for the programmer to understand, especially in object oriented languages, and is also shown to be easily optimized by compilers [**?**]. The construction of the STM is often different than word based implementations due to the fact that objects are accessed by pointers creating an additional level of indirection. There are also many object based STM implementations with many interesting properties including RSTM [**?**], LSA-STM [**?**], DSTM [**?**], [**?**], SXM [**?**], and JVSTM [**?**].

9.2.4 Programming

Memory organization (into words of objects) is just one aspect of how programmers interact with, meaning create and use transcations. In order to actually define transactions in code, seemingly the most common and most obvious approach is to surround the code by some keywords that indicate the beginning and end of a transaction. For example the keywords *BEGIN_TRANSACTION*, and *END_TRANSACTION* could be used. In a language even, if these are the only two keywords added in order to implement transactions it is not immediately obvious for the programmer how to use transactions. For example some unknows could be, can memory locations accessed inside a transaction be accessed outside of a transaction as well (this is a problem called privatization discussed later), or can a separate transaction be executed from within another transaction (this is called nesting, also discussed later)? In some implementations there are also some additional concepts and keywords that a programmer has access to, such as what to do in case of an abort, or to even release memory locations (memory locations that have been accessed and later released by a transaction will no longer conflict with any other transactions) giving the possibility of increasing performance, but also increasing the chance of programmer error.

Even is a programmer is able to correctly write a program using transactions, it does not mean that it is a good program. As described in this survey there are many different ways to implement transactional memory, and they can each deal with transactions very differently resulting in varying performance for varying workloads. For example certain STM implementations might perform better or worse depending on the length of a transaction, while others might perform better if a programmer puts write acceses to memory early in a transaction, while still others might perform best if some of the additional STM language constructs specific to that implementation are used. Because of this variation, there are currently no *"best practicies"* for programming using transactional memory and if a programmer wants to create the best program he has to find an implementation of Transactional Memory that is well suited for his application, and has to organize his code such that it is most efficient for the implementation he chose. This adds an additional learning curve for programmers, and is partially contradictory to the original idea of transactional memory, which is to abstract away the difficulties of concurrent programming. There are certain approaches to help take some of the burden off programmers to write efficient code such as using compilers to improve efficiency [**?**], or even to completely remove the concept of transactions from the programmer, and have them generated automatically. need citation, check Distributed computing column 29

9.3 Correctness

9.3.1 Consistency

A consistency criterion defines when transctions can be accepted (or commited) and when they must abort. Choosing a consistency criterion for memory transactions is different than from database transactions. A common consistency criterion for database transactions is *serializability*. Serializability guarantees that every committed transaction happened atomically at some point in time, as if they were executed sequentially. It is one of the weaker criterion because the real time order of execution does not need to be ensured. For example a transcation 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* another common criterion, 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. 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 undesireable 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.

9.3.1.1 Opacity

Opacity [**?**] was the first formally defined consistency criterion for transactional memory, it is also the most widely used and understood. It can be simply stated as linearizability for all transactions, including aborted ones. So all transactions must access a state of memory that has been produced by the previous committed transactions, and must be orderd based on real time execution. After a transaction reads an inconsistent state of memory it must not be allowed to execute any following statements. This prevents the bad behaviors (described previously) from happening due to reading an inconsistent state of memory. Consider the prefix of an aborted transaction upto the operation that caused the abort. For all aborted transactions this prefix must be ordered along with the committed transactions for a valid sequential execution. One important thing to notice is that both opacity and linearizability guarentee that a transaction looks to have executed atomically at any one point in its real time execution, not necessarily at its time of completion. So any concurrent transactions can be ordered in any way and depends on implementation, this flexibility gives the possibility to commit more transactions.

9.3.1.2 Virtual World Consistency

Imbs and Raynal recognized that for certain workloads opacity might be too strong of a consistency condition, so they defined *Virtual World Consistency* which is strictly weaker than opacity, but still ensures transactions only see consistent states of memory, preventing the undesirable effects of reading inconsistent states. In opacity aborted transactions can effect whether another transaction can commit or not due to the fact that aborted transactions must be serialized along with committed transactions. They give the following example in their paper: use figure? In

virtual world consistency aborted transactions must see a consistent state of the memory, meaning that if you take the a prefix of the aborted transaction, the events up to, but not including the event that caused it to abort, there must exist a serialization of it with previously committed transactions, but other concurrent and later transactions need not be considered. This serialization of other transacions up to the transaction in question needs not necessarily to be the the same serialization that some other concurrent transaction sees (whether or not the concurrent transaction commits or not). On the other hand all commited transactions must see the same serilization. For example the view of an aborted transaction might not contain a transaction that was serialized before (in the committed history) some other transaction that the aborted transaction does see. Imbs and Raynal also give an STM algorithm that uses sequence numbers to keep track of writes to variables and satisfies virtual world consistency. This algorithm accepts certain histories that are virtual world consistent, but not opaque. They give a formal proof that the algorithm satisfies virtual world consistency. They prove that virtual world consistency will accept more histories than opacity, but how and if this is helpful in parctice is left unexaimed.

9.3.1.3 Snapshot Isolation

Need to write this section

9.3.1.4 Consistency Criterion Limits

After Gerraoui and Kapała define opacity [**?**] they prove an upper bound for a certain type of TM implementations that ensure this consistency condition. Namely single version TMs with invisible reads that do not abort non-conflicting transactions and ensure opacity require in the worst case $\Omega(k)$ steps for an operation to terminate, where k is the total number of objects shared by transactions.

The choice of consistency criterion obviously makes a large impact on a STMs implementation and performance, yet much of this relation is widely unknown. The previous paragraph gives one result showing a cost of choosing opacity for a TM implemented in a certain way. Opacity is the most widely studied and implemented consistency criterion and some research has been done on different aspects of how it effects implemntations, but there is still much to be done. Part of the difficulty is that consistency criterion is just one of many different design choices one can make for a TM that all impact eachother, and there is no universially accepted definate choice for any of these. For example any combination of choices for read visibility, write visibilty, blocking, livness, consistency, nesting, privatiztion, is just as valid as some other combination of these choices (these terms are discussed later in this survey). Many of these different combinations have been shown to work well on certain workloads or benchmarks, but very little theoretical results have been proven. Also few specific TM implementations formally prove their consistency criterion or show how the choice of criterion effects their implementation choices and performance. It is hopeful that as more theory is proven the choice of how to implement a TM will become more clear. As this survey introduces the different concepts behind STMs it will discuss some of the theory that has been proven for them.

9.3.1.5 Other Consistency Criterion

There are many other consistency criterion that have been proposed and implemented. For instance in order to increase the commit rate of transactions, removing the real time requirement

of ordering transactions from opacity has been considered, defined in [**?**] as *real-time relaxation*. In [**?**] Alvisi defines a lock-free STM that follows this criterion. Another STM that implements real-time relaxation and is also decentralized is defined in [**?**]. In the same paper they examine the acceptance of different transaction memory implementations based on different criterion they implement including consistency. Again in order to increase performance certain STMs can allow transactions that will eventually be aborted to access inconsistent states of memory following the definition of serializability or linearizability from database transactions. This usually leaves the programmer to deal with the problems that come with accessing inconsistent states of memory. Even though opacity and virtual world consistency have both been formally defined, and opacity has been partially exaimned, other consistency criterion have been studided even less, and for some TMs implementations their consistency criterion has not even been formally defined.

Certain other mechanisms available to programmers have been proposed such as *early release*, proposed in [**?**] which allows programmers to tell the system to treat some memory location it has read as if the read did not actually happen in order to increase the chance of committing the transaction. Another mechanism proposed in [**?**] is *elastic transactions* which allow the programmer to mark transactions that can be split into multiple transactions at runtime also in order to increase the commit rate. Both these and possibly other mechanism effect the consistency of an implementation, but the changes they cause to consistency, performance, and bounds are widely unstudied, and their effects can often be unclear to a programmer leading to possible errors.

9.3.2 Privatization

Another question on correctness is how should shared memory be dealt with outside of transactions, this is called *privitization*. Many TM implementations do not address the issue of privatization, yet it is important, as a piece of code that is correct in one implementation may be wrong in another given by how they deal with privatization. For example consider the following transactions A and B and a shared list object $list_x$.

Transaction A TODO

Transaction B TODO

This example is similar to the one in []. in the guys phd thesis on page 156 Transaction A reads the head of a list, and then sets the shared pointer to *null* in the hopes that no other transaction will be able to access the list. Once the transaction commits, the code goes through the list performing some operations on the elements non-transactionally. Transaction B reads the head of the list, then goes through the list performing operations on the list while in the transaction. Now assume the following order of execution happens. Transaction B starts executing and gets to line 3, now transaction A starts executing and commits successfully. After transaction A commits, the non-transactional code following A continues to perform some operations on the list concurrently with transaction B, which is performing operations on the same list, resulting in some undesireable behavior. Notice that transactions A and B are concurrent, so in the described execution as long as they are serialized as A first, then B, they can both be committed and satisfy opacity.

In order to avoid problems such as this there are different models to deal with privatization, they each have trade offs on efficiency and programmer involvement.

9.3.2.1 Strong Isolation

A transactional memory that provides *strong isolation* guarantees that transactions are isolated from other transactions operations on shared memory as well as from non-transactional loads and stores. In this model the example above would not be allowed to happen. This could be done in several ways, one way would to not allow the code to compile at all. In *STM Haskell* [**?**] variables are declared as either transactional or non-transactional, and only transactional variables are accessible inside of transactions, and only non-transactional variables are accessible outside of transactions, preventing code like the example above. does this weaken the programming in any way? Another solution would be for the transactional memory implementation to execute the code in a way so that the non-transactional accesses do not interfere with the transactional accesses. This is called *transparent privatization*, where the STM itself guarantees all transactions privatize data correctly. In [**?**] they introduce a way of implementing transparent privatization called *validation fences* where after a transaction commits, it waits until all concurrent transactions to finish executing (committing or aborting) before executing any non-transactional code that occurs after the transaction. Of course this can limit performance and scalability, especially in workloads that have long transactions, and in order to write efficient code the programmer may have to know that privitization is implemented in this way. In order to implement a more efficient and scalable privitization, SkySTM [**?**] uses *conflict-based* privatization which allows transactions to only wait on conflicting transactions to finish.

9.3.2.2 Semi-visible Privitization

mention semivisible privitization from the guys phd thesis?

9.3.2.3 Weak Isolation

A transactional memory that satisfies *weak isolation* guarantees only that transactions are isolated from each other. In this model the example above would be allowed to happen. Most transactional memory implementations that do not mention privatization probably satisfy weak isolation. Usually these implementations assume a model similar to single lock atomicity, where if there is concurrent access to the same memory location within the lock as there is outside a lock, then this is considered a bug, likewise in a TM if there is a concurrent access to shared memory inside a transaction as there is outside a transaction, then this is programmer error. Certain implementations deal with this by allowing the programmer indicate when he wants to privatize data, this is called *explicit privatization*. So that when a programmer privitizes the data he can be sure that it is no longer accessable by transactions. Again this is a trade off, weak isolation and explicit privatization are aimed at improving performance, but has the side effect of making code more difficult for the programmer to write.

9.3.3 Error Handling

An overview of error handling in transactional memory is given in [**?**], may of the concepts introduced there are repeated here. What should happen when an exception occurs midway through a transaction? For example say within a transaction there is some code that opens a file

for reading, but when it tries to open it, the file has been previously deleted and an exception is thrown. Or say a transaction for a bank application transfers money from one account to another, but in some case when it tries to do this an exception is thrown because there is not enough money in the first account.

one way of dealing with this could be, abort the transaction, and restart it, similar to if it was aborted due to a shared memory conflict. With the bank application this could possibly be acceptable, because the transaction will be executed successfully once enough money is deposited in the first account. For the file transaction this may or may not be acceptable, for example the file might have been permanently deleted, and the transaction will be aborted and re-executed indefinitely.

Another solution might be to partially commit the transaction up to the exception, then propagate the exception. This can be viewed as similar to what happens in sequential code, when an exception is thrown in a sequential execution, the code up to what caused the exception was executed sucessfully. Unlike in sequential code doing this in transactions violates the expected atomic all or nothing behavior which can cause problems if the programmer does not deal with this correctly. For example say the money transfer transaction first increments the balance of the destination account and then decrements the source account, and if there is not enought money in the source account then some exception specifc to this is thrown. This may seem rediculous, but the programmer writes the code that handles this exception by consequently removing the amount added to the destination. But after the transaction is partially committed, and before the exception is handled, the system is in some sort of inconsistent state, and any number of bad things can happen, for example the money desposited in the destination account might be withdrawn by some other transaction. maybe can think of a better example?

A third solution might be to abort the transaction and propagate the exception, but this requires the programmer to make sure he handles the exception correctly knowing that the code that caused the exception and the code prior to it in the transaction appears as if it was never executed to the rest of the system.

A fourth solution might be to require exceptions to be dealt with from within the transaction, similar to the following try, catch syntax in Java.

TDODO!!

Certain STMs implement their own mechanisms not specifically designed for handling exceptions, but that can be used for exception handling. For example STM Haskell [**?**] has a mechanism that allows a transaction to follow a different execution path after it is aborted and retried. This mechanism even allows the programmer to abort the transaction for anywhere within the transaction's code.

Due to the notion that transactions observing inconsistent states of shared memory can cause problems as described in the section of this survey on consistency criterion, most implementations ensure the opacity consistency criterion, where all live transactions observe a consistent view of shared memory. In the interest of efficiency weaker consistency conditions that allow inconsistent views of memory have been considered, but rarely examined deeply because of possible problems. Well done error handling could likely deal with these problems and lead to the possibility of these and other new consistency conditions be more widely considered.

It is interesting to study how exception handling and its implementations can effect and bound correctness, liveness, and performance theoretically and in practice. Currently there is no obvious answer to handling exception in transactional memory and what works best might vary depending on the application and implementation. But if implemented correctly error handling

in TMs could actually benefit code execution over sequential code. In transactional memory, a sequence of code that caused an exception could be automatically rolled back and then executed in a way that would prevent the exception. While in sequential code it might be difficult to make sure the sequence of code up to the exception is rolled back.

9.3.4 I/O

I/O is also a subject of difficulty in transactional memory because it is not always obvious how to abort and rerun an I/O operation. Take for example a transaction that writes some data to the screen, and then is aborted, the characters written could then be deleted from the screen, but this would be strange for the user. Output could be buffered until a transaction commits and then displayed, but this could cause performance issues. Input is also difficult because it often requires real time performance, and there are questions like should the input be repeated on abort, or should the previous values be reused? The answers might be application specific.

A simple solution might be to disable I/O operations from being executed within transactions, such as in STM Haskell [**?**], but this might not always be possible, and might create difficulties in programming which may vary in conjunction with the chosen privitization technique. what effect does this exactly have on restricting power of programming?

Another solution implemented in [**?**], called *inevitability*, allows a transaction with I/O operations to be marked as inevitable so that it is never aborted, whenever it conflicts with a transaction the other transaction will be aborted. The problem with this solution is performance, only one transaction at a time can be inevitable, and could cause a workload to execute at the speed of a single processor. Like with exception handling is also interesting to study how the implementation of I/O in transactional memory can effect and bound correctness, liveness, and performance theoretically and in practice. An overview of the difficulties associated with I/O in transactional memory is given in [**?**] and [**?**].

9.3.5 Nesting

Part of the difficulty caused by programming with locks is that they are not composable. It is often not obvious and not even always possible to create new code using locks that reuses code that also uses locks and still perform the correct function while avoiding deadlock. 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 reuseability. Still it is not obvious how implement composibility 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*, 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 just creates larger and larger transactions, limiting performance.

A slightly more complex model *closed nesting* 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 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. In [**?**]

an implementation of closed nesting is given for the Java language along with some additional mechanisms that can be used when a nested transaction retries or aborts.

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 conclicts with concurrent transactions. 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. [**?**] gives a comprehensive overview of open nesting. Allowing a transaction to commit to shared memory fromm within a transaction obviously violates the idea that everything within a transaction is executed atomically, so a new consistency model has to be considered, one such model, *abstract* serializability, is described in [**?**].

In open nesting handling the situation where a parent transaction aborts after a nested transaction has committed becomes very difficult. It is not sufficient to just roll back the nested transaction as that could lead to inconsistencies with other transactions that had committed after the nested transaction committed. In order to deal with this [**?**] introduces a set of extra mechanisms that a programmer can use, but this also introduces quite a bit of difficulty for the programmer, and if used incorrectly can cause deadlock. Because of these difficulties, they [**?**] conclude that open nesting is only useful for expert programmers, but it allows them to create efficient libraries of data structures and algorithms that can benefit largely from open nesting. These libraries can then be reused by non-expert programmers when writing transactional code where they themselves cannot create open nested transactions.

In [**?**] a sort of hybrid compromise between open nesting and closed nesting is suggested in what they call an *ownership-aware commit mechanism*. I haven't read this paper, but it looks like it might be interesting, but the work package does not mention nesting anywhere as something to work on??

9.4 Liveness

Liveness is an important and well studied topic in transactional memory research. It deals with the concepts of what guarantees should an implementation ensure for the progress and completion of any workload. For example should every workload be guaranteed to complete, or is it acceptable to have the posibility for some workloads to be stalled indefinitely by concurrent conflicting transactions? At first intuition it might seem essential that a work load should be guaranteed to complete, but this and other guarantees of liveness often result in performance overhead, so the question has to be asked: it is worth it?

9.4.1 Non-blocking

Non-blocking implementations guarantee that threads cannot be postponed indefinitely while waiting on a competing shared resource. Traditional lock based programming does not guarantee this and many programs suffer from deadlock which is considered to be programmer error. The goal of transactional memory is to prevent this, so most transactional memories try to prevent programmer error from causing blocking. In the intrest of efficiency, as will be shown in the following sections, many implemntations still use locks so they are inheriently blocking. They are mostly written in a way such that dead lock cannot occur, but one transaction can still block another. For example a transaction that holds a lock could be deshceduled, thereby blocking any other transaction that must acquire that lock before continuing. There are three levels of nonblocking studied each with guarenteeing different levels of progress, namely obstruction freedom, lock freedom, and wait freedom.

9.4.1.1 Obstruction Freedom

Obstruction freedom is the weakest of the non-blocking guarantees, it says that any thread, if run by itself for long enough will make progress. Meaning that any thread running with no concurrent conflicting threads will make progress eventually. This seems like a nice property for transactional memory and under obstruction freedom the following are not true:

It prevents dead lock.

A crashed thread cannot prevent further progress.

A thread that is descheduled cannot prevent further progress.

The first STM implementation [**?**] was obstruction free, but it only allowed *static* transactions. Since then other obstruction free STM implementations have come along such as DSTM [**?**], which allows *dynamic* transactions, WSTM [**?**] what is new about WSTM?, RSTM [**?**], which aimed at reducing the overhead of non-blocking vs locking implementations, Adaptive-STM [**?**], which adapts itself based on the workload with the goal of improving performance, and LSA-STM [**?**], which uses a *time-based* approach to conflicts and implements *multi-versioning*. Even though the properties guaranteed by obstruction freedom are nice, they are not free, guaranteeing obstruction-freedom can create a large amount of overhead for a transactional memory implementation.

In [**?**] Ennals gives a strong argument against current obstruction-freedom implementations. Most STMs use pointers to memory loactions in order to implemnt obstruction freedom creating at least one level of indirection. This allows the pointers to be atomically changed to a new memory location at commit time using *compare and swap* operations. Because of this Ennals argues that obstruction freedom implementations by design result in increased cache misses, which in certain workloads can drastically hurt performance. Additionally he explains that if you have more transactions ready to execute than number of cores, in obstruction freedom all of them must start executing immediately, while a better solution would be to only execute as many transactions concurrently as you have cores because the fewer transactions running currently, the fewer possible conflicts, and less overhead from sharing a core. He also argues that the benefits of obstruction freedom are not entirely necessary. For example a descheduled transaction will only be paused temporarily, and if the system is implemented correctly this should not be a common problem because there should only be as many transactions running as there are cores, and a system could be discouraged from descheduling a live transactions. In [**?**] the kernel of the operating system is modified to allow *time-slice extension* which allows a thread's time-slice to

be extended up to some chosen amount of time if it is running a transaction. Ennals also argues that a crashed or failed thread would prevent further progress in a non-STM system, so it is also acceptable to do so in an STM program.

Reguardless, if the good properties of obstruction freedom can be ensured without too much overhead this gives TMs even more benifits over shared code. A deeper study of the complexity of obstruction-free implementation of concurrent objects in general has been done in [**?**] and I need to read this paper so I know what it says. Even though obstruction-freedom ensures some good properties, bad properties can still exist though such as starvation and livelock so stronger non-blocking properties are needed in oder to ensure these do not happen.

9.4.1.2 Lock Freedom

In an algorithm that guarantees *lock freedom* the system as a whole must make progress after a finite amount of time. Meaning that at least one thread is guaranteed to make progress eventually. This can be desirable for an STM because it prevents *livelock* and at any point when there are active transactions executing, at least one of those transactions must commit. Being an even stronger guarantee that obstruction freedom, it requires even more overhead and a decision has to be made if the beneficial properties are more valuable than the additional overhead.

Most transactional memory implementations that are lock free do so by informing other transactions of the locations it will write *lazily* (meaning at commit time) and recursively *helping* conflicting transactions commit. For example if a transaction containing write a is descheduled while it is committing, and just after this some other conflicting transaction comes along wanting to commit, instead of waiting for the descheduled transaction to commit, it will start helping the other transaction to commit. The problem with helping is that an arbitrary number of transactions can be concurrently trying to help another transaction commit, performing many costly low level operations such as *compare-and-swap*, resulting in many cache misses, contention, and likely poor performance. Some STMs that guarantee lock freedom are OSTM [**?**] and AVSTM [**?**], and another in [**?**]. Lock freedom still does not prevent *starvation* so stronger guarantees of progress are also considered.

9.4.1.3 Wait Freedom

Wait freedom is the strongest of the non-blocking guarantees and it says that each thread will make progress after a certain finite amount of time. Wait freedom prevents *starvation*. I am not aware of any wait-free implementations, are there any?

9.4.2 Contention Management

Only very few transactional memory implementations are lock-free, most are either blocking or obstruction free, which allow starvation and livelock, and even lock-free implementations do not prevent starvation. Yet starvation and livelock are very undesirable properties to have, and if they are ignored can ruin the performance and progress of a system on certain workloads. *Contention Management* is used in order to discourage livelock and starvation, and in certain cases, provably eliminate one or both completely [**?**].

9.4.2.1 Contention Management Implementations

Contention management was originally implemented in DSTM [**?**] 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. DSTM was proposed with two possible contention management policies *aggressive* and *polite*. In the aggressive policy when a transaction detects a conflict, it immediately forcefully aborts the conflicting transaction. In the polite policy, when a transaction detects a conflict, it will back off and wait a certain amount of time. When the transaction starts back up if the conflict still exists the transaction will back off again for some exponential and possibly randomized amount of time, this cycle will continue until some given time threshold is reached, at which time the conflicting transaction will be forcefully aborted. The hope of the back off is that the extra time will allow the conflicting transaction to commit or abort possibly removing the conflict and avoiding aborts all together. 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.

9.4.2.2 Difficulties

Mostly these contention managers have been proposed and designed by making certain assumptions about what appications transactional memory will be used for, then validating (or disproving) their assumptions by examining the contention manager's performance on a limited set of implementations and workloads. Part of the difficulty is that an contention management strategy that may work well for one STM implementation may not work at all for another, based on how the STM implements things such as visibility of reads, and eagerness of acquire for writes [**?**]. Given this, certain TM implementations such as LSA-STM have been tested using a wide array of different contention managers [**?**] but there is no definiate 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 managment is know to perform well on workloads with a regular access pattern [**?**], while polka [**?**] works well on small scale benchmarks [**?**]. The obvious problem with this is that there is no "best" contention manager that performs well in all reasonable situations, in [**?**] 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. In addition, 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 livelock or starvation. The contention manager *greedy* [**?**] is an exception to this, when it was introduced it was also proven to prevent livelock 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 [**?**]. Similar to being livelock free, but not quite as powerful, greedy ensures the *pending commit* property, which ensures that at any time, some running transaction will run uninterrupted until

it commits. Greedy works by assigning a timestamp to each transaction when it starts, then when a transaction *A* discovers a conflict with a transaction *B* it will abort *B* only if *B* has a later timestamp or if *B* is waiting on another transaction, otherwise transaction *A* will start waiting. Note that the timestamp is kept even after a transaction is aborted and restarted.

9.4.2.3 Improved Techniques

Some solutions have been proposed in order to increase the efficiency across all workloads, for example in [**?**] *polymorphic contention management* is defined. In this paper a generic framework for a contention manager is defined, allowing multiple different types of contention managers to be used throughout a workload. A specific contention manager can be defined for each transaction, or even on the abort of a transaction a new contention manager can be used for it when it is retried. This allows the best contention manager to be chosen at any point in a program's execution. But unfortunately this does not answer the difficult question of what manger will be of best use at what time, and it introduces more difficulty for the programmer to decide what to use and when.

A recent STM implementation *Swiss-STM* [**?**] uses *two-phase* contention management in order to deal with varying workloads. It combines the passive contention manager for short transactions, with the Greedy contention manager for longer transactions. They define a long transaction as one that has performed more than 10 writes (although this can be changed to any number), if a transaction has performed less writes then this whenever it detects a conflict it will abort itself, as soon as the transaction performs 10 writes it is given a timestamp to be used with the greedy contention manager. If a transaction that is using the greedy contention manger detects a conflict with a transaction using the passive contention manager, it immediately aborts the conflicting transaction. By combining the two contention managers, Swiss-STM, guarantees freedom from starvation and livelock ensured by the greedy contention manager for long transactions, without introducing too much overhead for short transactions. They also expand on this by adding a randomized linear (in the number of successive aborts) back-off for when aborted transactions restart, similar to the polite contention manger, except in polite the back-off is done before the abort. This is done because if there is certain meta data in an implementation or shared memory in a workload that is updated frequently restarting a transaction immediately after a conflict can increase contention on this data and cause scalability issues [**?**].

In [**?**] they recognize that most contention management schemes take the approach of *conflict resolution* so they try to add to this approach by examing other aspects of conflicts. In order to discourage livelock no transactions announce their writes until commit time (called *lazy acquire*), so conflicting writes can only cause an abort at commit time. This is usually a short time window compared to the total duration of the transaction, resuting in it being likely that at any time at least one running transaction will commit. They also suggest using a randomized back off on abort mechanism similar to Swiss-STM. A separate mechanism is also implemented to discourage starvation which is more similar to other contention management implementations. This works by assigning a priority to a transaction so that a lower priority transaction cannot abort a higher priority transaction. Implementing priority for every transaction causes a large amount of overhead, so they implement a system where only certain transactions are assigned priority, which lowers the overhead, but does not make any guarantees of progress for transactions without priority. They suggest two ways of assigning a priority to a transaction, the first is by allowing the programmer to give a transaction priority when writing the code, and the second is increasing the priority of a transaction after it has been aborted a certain amount of times.

9.4.3 Transactional Scheduling

Transactional Scheduling is similar to contention manager in that it tries to improve and ensure progress of transactions. But instead of just deciding to abort or wait when transactions conflict, it also decides how to order the execution of transactions. For example assume there are N cores in a system and at some point in time there are $2N$ different program threads each with a transaction ready for execution. There are different ways to choose how to execute the transactions, a system could just execute all $2N$ transaction concurrently, but this could decrease performance for two reasons. The first more obvious reason is that the more transactions that are run in parallel, the higher the chance of conflict, the second reason is that since there are more transactions then cores, a transaction could get descheduled, possibly causing problems such as preemption. The purpose of a *transactional scheduler* is to improve on situations such as this by attempting to optimize the order of execution of the transactions. This is done by having a queue of pending transactions waiting to be executed at each core. A simple scheduler might just assign transactions to a processor based on how many items it has in its queue. So in this example each processor would be assigned tow transactions each. This is obviously not the perfect solution, for example one processor might be assigned two short transactions while another might be assigned two long ones, so a possible improvement might be to assign a single transaction to a process as it completes the previous transaction. There are other issues to consider, such as how to deal with aborts, if a transaction aborts, should it immediately be restarted again risking the same conflict, or should it be queued somehow and executed later.

Transactional schedulers are implemented to improve efficiency and often define their own conflict resolution protocols, but it is also interesting to examine how they can effect liveness properties, and where possible how the combination of different schedulers with contention managers effect liveness and performance.

9.4.3.1 Work stealing

The idea behind *work stealing* [**?**] is to prevent cores from being idle when there are pending transactions in the system. When a core is ready to execute a transaction, first it tires to get one from its own queue, but if it is empty, it then randomly selects another core with a nonempty queue, and removes the transaction at the tail of that queue and starts executing it.

9.4.3.2 Steal-on-Abort

Steal-on-abort is a transactional scheduler and contention manager proposed in [**?**]. Its goal is to reduce *repeat conflicts*, which are defined at as conflicts that occur between two transactions that had previously occurred. Many contention managers will restart an aborted transaction immediately, likely resulting in the same conflict happening again. Steal-on-abort deals with this by taking the aborted transaction and placing it in the queue of the core that is executing the transaction that caused the abort. Different implementations might place it at different locations in the queue. This way the aborted transaction will (likely) not execute again until the transaction it conflicted with has completed execution, thereby avoiding repeat conflicts. Steal-on-abort also implements work stealing so that there are no idle cores when there are pending transactions, but note that this also creates the possibility for repeat conflicts.

9.4.3.3 BIMODAL Scheduler

The *BIMODAL scheduler* [**?**] is an advanced transactional scheduler designed to be efficient on *bimodal* workloads, which are workloads containing only early-write and read-only transactions. In the BIMODAL scheduler there is a single FIFO queue called the RO-queue (or read-only queue) shared among all cores, along with a double-ended queue, or dequeue at each core. As transactions arrive they are inserted at the tail of the cores' dequeues in round-robin order. Each transaction is assigned a timestamp when it first starts, that it retains even after aborts. If a write transaction conflicts with another write transactions, the one with the later timestamp is aborted and placed on the dequeue of the conflicting transaction's core (similar to steal-on-abort). The system alternates between *writing epochs* and *reading epochs* in which conflicts between read-only and write transactions are handled differently. In a writing epoch transactions are executed from the work dequeues of the cores, if a read-only transactions conflicts with a write transaction, then the read-only transaction is aborted and is placed at the end of the read-only (or RO) queue. Note that here read only transactions are thought of as ones that have only done reads up to the point they conflict (it is possible they might do a write later). During a reading epoch transactions are executed by taking them from the shared RO-queue, if there is a conflict between a read-only and write transaction, the write transaction is aborted, and placed at the head of the work-dequeue of the core that was executing it. Epochs switch after either a given number of transactions have been executed, or when their corresponding queues are empty.

9.4.4 Contention Management and Transactional Scheduling Bounds

Some work has been done on proving the efficiency bounds of contention managers and schedulers. In order to show these results, first some definitions will be reproduced from [**?**].

Définition 9.1 ***(Makespan)*** *Given scheduler or contention manager A and a workload* Γ*,* $makespan_A(\Gamma)$ *is the time A needs to complete all the transactions in* Γ*.*

Définition 9.2 ***(Competitive ratio)*** *The* competitive ratio *of a scheduler or contention manager A for a workload* Γ*, is* $\frac{makespan_A(\Gamma)}{makespan_{Opt}(\Gamma)}$*, where OPT is the optimal, clairvoyant scheduler or contention manager that has access to all the characteristics of the workload before execution.*

The competitive ratio *of A is the maximum, over all workloads, of the competitive ration of A on any* Γ*.*

Définition 9.3 ***(Conservative scheduler or contention manager)*** *A scheduler or contention manager A is* conservative *is it aborts at least one transaction in every conflict.*

In [**?**] they prove that any contention manager that insures the pending commit property has an upper bounded competitive ratio of $O(s^2)$, where s is the number of shared objects in the system, this includes the greedy contention manager. Specifically for greedy this upper bound is improved to $O(s)$, and an equivalent lower bound of $\Omega(s)$ is given in [**?**] for any deterministic contention manager, creating a tight bound of $\Theta(s)$ for greedy. The effect of *failed* i.e. crashed (i think?) transactions on the bounds of the competitive ratio of greedy are also examined [**?**].

Some bounds are also proven for *randomized contention* managers [**?**] and a decentralized randomized implementation satisfying the lower bound within a logarithmic factor is given in [**?**].

In addition to defining the BIMODAL scheduler in [**?**] they also prove some efficiency bounds for certain classes of tranactional schedulers. They prove that any non-clairvoyant conservative scheduler is $\Omega(m)$ competitive for some workload with late writes, where m is the number of cores. This is obviously bad because the optimal solution uses all cores at all times, while the transactional memory version is no more efficient than using a single core. Note that not all contention managers are conservative, for example some will have a transaction back-off in presence of a conflict instead of immediately aborting. Although this is not a promising result for transactional memory, it is very common to have *read dominated* workloads in transactional memory, and for these they show that every deterministic scheduler is $\Omega(s)$ competitive, where s is the number of shared resources.

Concentrating on workloads without late writes *i.e.* bimodal workloads, they show that steal-on-abort also has this poor $\Omega(m)$ competitive makespan even for some bimodal work loads. The BIMODAL scheduler improves on this by having a $O(s)$ competitive ratio for for any bimodal workload, which they also show is optimal for a conservative scheduler. Even though the BIMODAL scheduler provides some nice bounds it requires quite a bit of extra computation including visible reads, and a shared counter, which introduces contention and increased meta data, so it might not be a good solution for practical implementations. Most other schedulers and contention mangers have the opposite problem of BIMODAL, that have only been shown to be efficient on certain benchmarks, they might have very poor bounds, and worst case workloads, which could cause problems that do not show up in the benchmarks.

9.5 Alternative Models

This section will look at some of the alternative mechanisms that have been proposed that are extentions of the basic idea of transactional memory that are not needed for an implmentation to be defined as a transactional memory.

9.5.1 Early Release

Early Release was first introducted as along with DSTM [**?**] as a method to increase concurrency. In the code of a transaction a programmer can *release* objects that were previously read by the transaction. Releasing a memory location that has been read means that the location will no longer be considered a point of conflict, to the underlying implementation it looks like that value was never read by that transaciton. This is useful for situations such as traversing through a list, as a transaction traverses through a list, it can release previous locations it had read as long as it does not access them again. This decreases conflicts without the posibilty of unwanted side-effects. Unfortunately this is not true in all cases, if used incorrectly releasing memory can cause the system to view inconsistent states of the memory. This can voilate the consistency condition, and introduces more difficulty for the programmer as he must make sure releasing memory does not cause problems. Since DSTM many other STMs have also incuded early in their implementations release in the intrest of performance.

9.5.2 Elastic Transactions

Like early-release, *elastic* transactions [**?**] were introduced as a way to increase concurrency. Also similiar to early-release they provide an extra mechanism to the programmer which if used correctly can increase poerformance, and if used incorrectly can cause the transaction to view

inconsistent states of the data. The idea behind elastic transactions is that a programmer declares a transaction as elastic or as normal, and the implementation can split transactions defined as elastic into multiple transacions so that they can be committed more often. If transactions could be cut anywhere then obvoiulsy this would not be any different than not using a transaction at all, so they restrict cuts to certain curcumstances. For example they do not allow a transaction to be cut inbetween two acceses to the same varible, where a concurrent transaction writes to that variable inbetween the cut. Elastic transactions allow things like searching a linked list to be implemented safely, but traversing the list while summing the vales for example could not be implemented safely. Since this changes the consistency condition for elastic transactions a new consistency criterion called *elastic-opacity* is introduced and a TM that implements this criterion is defined [**?**]. They also show they there are histories accepted by this that could not be accepted by using early release.

9.5.3 Abort/Retry/Blocking

A consistency criterion defines when a transaction must abort, and a conflcit managers decides how to deal with transactions in case of a conflict. In the basic model the programmer is not concerned with aborts, he just knows that all the code in a transaction will execute atomically. Ceratin implementations give more options to the programmer to deal with aborts.

STM Haskell [**?**] allows the programmer to call a function called *retry* from within a transaction which will immediately abort the transaction. STM Haskell also implements strong isolation, so only transactional variables are accessable from within a transaction. This allows the implementation to *block* until one of the transactional variables have been modified, so that in a well written program the retry should not be called again for the same reason. An additional mechanism that STM Haskell provides is what is called *choice*. This is implemented by an *orElse* keyword, this allows the programmer to define two blocks of code, the transaction executes the first block, and if retry is called from within this block, then it is aborted, and the second block is executed, if the second block calls retry, then the whole transaction is aborted. Two other TM implementations [**?**] and [**?**] also give similar retry and orElse constructs and [**?**] shows how they can be optimized by a compliler.

In [**?**] not only is the programmer able to define a specifc contention manager for each transaction, but on abort the posibility is given to be able to change the conention manager.

RSTM [**?**] introduces a construct called ON_RETRY, which a progammer can put around a block of code at the end of a transaction. When a transaction aborts the code in this block will be exectued. It is intended to allow the programmer to clean up memory that was allocated from within the transaction.

These and many other constructs can add different and new, interesting and helpful properties to a TM, yet most of them break the original simplicity of transactions. They can create different effects such as increased difficulty for programmers, modified consistency condidtions, and changed theoretical aspects of the TM. Many of these effects have yet to be fully studied or understood and before they are it is unlikely that these constructs will be accepted or integrated into the core concepts of transactional memory.

9.6 Implementation

The previous sections discussed what an STM should do, and looked at why this is not an easy question to answer. This next section will look at how to to actually implement these things, and some of the difficulties that go along with this. For many of these design options there is not a clear choice, in [**?**] they perfrom benchmarks on many of these options and come conclusions as to which choices are prefered, but it is not clear that their decisions are conclusive.

9.6.1 Write Buffering vs Undo locking

When performing a write within a transaction, to the rest of the world the value written is not known until the transaction commits. 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*, 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 buffereing* when a transaction wants to perform a write to a shared memory location first it will make a local copy of that variable only visble 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 be put into the shared memory.

There are some advantages and disadvanteges to both solutions. Undo locking is nice because it does not have to keep track of local memory for writes, and when write transactions committ it does not have to copy the values from local to shared memory. Write buffering is nice when transactions abort because it does not have to perform roll back on the shared memory, and it allows the posibility to use invisible writes which are described later in this section. Different STMs choose to use one or the other, given that write buffering can implement invisible as well as visible reads it has been implemented in many TMs, but other then this there is often no definite or obvious reason to choose on over the other, or why one is better. What limitations does one have over the other?

9.6.2 Compilation

Special complication technicques specifically oriented towards transactions can improve the performance of STM. Different design decisions impact the performace of compiled transactional code. For example certain implementations require that a programmer define what variables will be access transactionally, while others require the compiler determine these variables, and in order to be correct the compiler might declare more variables as transactional then nesssary causing increased overhead during execution. In [**?**] an efficent STM is designed with optimizations for their JIT complier. [**?**] discusses some of the dificulties with STM code complication.

9.6.3 Cache misses

In his paper about efficient software transactional memory [**?**], Enanals especially brought forward the problem of cache misses where he designed an STM that performed as few cache misses as possilbe and showed it to be much faster than other STM designs. One of his main claims was against obstruction free implementations where, he claims, that since they require at least one level of inderection between object metadata and actual data they introduce many cache misses. Cache misses are also common when there is high contention over shared objects such as a commonly accessed global counter, so cache misses and sclibility are inherently related.

Other then Ennals design [**?**], many STMs take cache misses directly into consideration. RSTM [**?**] is an obstruction free implementation that organizes its meta and object data in a way to reduce cache misses, and SwissTM [**?**] is designed for efficency where they consider the size of shared memory words and components of the STM in order to reduce cache misses. It could be interesting to study what components of a TM effect cache misses, and how cache misses can effect other TM properties such a scalibility.

9.6.4 Non-blocking & Obstruction-free

As discussed in the section of this survey on obstruction-free liveness properties, implementing a STM with locks is much more straightforward then implementing one that is obstruction-free, but obstruction freedom insures some nice properties. In [**?**] they discuss some of the complexity required for implementing obstruction-free objects in general, which could have some implications towards how TMs should be implemneted. Need to read this paper

The previous section shows that cache misses have been a concern as something that could hinder performance for obstruction free implementations. Recently what are considered and are designed as the more efficient implementations such as TL2 [**?**] and SwissSTM [**?**] use locks. It is difficult to tell though if this is because lock based implementations are easier to design, or because obstruction freedom is actually inherently slower. Research still needs to be done on these subjects and even less is know as how to efficently implement even stronger guarentees of progress such as lock-freedom and wait-freedom.

9.6.5 Kernel Modification

Modifying the kernal in order to better support the exection of transactions from the operating system could be important for the implementation of effective and efficient transactions. In [**?**] they modify a linux kernel in order to better support transactional contention managment and scheduling. From within the OS they provide what they call *serialization*, which will yield an aborted transaction from executing until its conflicting transaction has completed. They also provide a sort of contention managment similar to serialization that uses prioirty, so that when an aborted transaction is restarted, it is started with lower priority which is then used by the OS thread scheduler. They show with benchmarks that implementing these funtions in the OS is more efficient the performing them at user level.

Another mechanism implemted in the kernel is *time-slice extention*, which allows a thread that is executing a transaction to request its time slice to be extended when it has expired and the OS is about to deschedule it. The thread can request extentions up to some maximum ammount defined by the OS. Desceduling a transaction during execution delays it, increasing its likelyhood to conflict with another transaction, time-slice extention tries to prevent this.

There are likely many other ways that transactional memory can be supported in the operating system, and if a mainstream OS supports some transactional memory implementation it could help lead to the wide acceptance of transactional memory.

9.6.6 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. Earlier in this survery in the section on liveness it was shown that deciding what to do after detecting a conflict is no easy task. This section will discuss some of the ways transactional memory implementations detect conflicts, and like conflict resolution, there is no simple answer to conflict detection.

9.6.6.1 Visibility

Transactional reads and writes can either be *visible* or *invisible*. 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. Whenever a read occurs 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 has occured.

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 worrysome for read dominated workloads beceause contention is being introduced when there are no conflicts and can limit scalibility. Invisibe reads have the disadvantage that they have to perfrom *validation* every time the read a new location, otherwise they might have an inconsistent view of the shared memory. With visible reads, when a location that has been read gets overwritten by a writing transaction the contention manager is called, so validation is not needed.

While it is fine to have mulitple readers for a memory location, there can only be one writer per location, so visible writes are usually implemented by acquiring a revokable lock (or something similar) for the memory location. A possible disadvantage of invisible writes is that whenever a transaction performs a read it has to perform a write set lookup. Meaning that the implementation has to check if the value read should be loaded from shared memory or from a local copy, this is usually done by traversing the local set of writes that the transaction has done so far, causing overhead. In order to get around this, in [**?**] they use a hash table to map addresses to indexes in the local write set. They show this gives negligible overhead for write set lookup when comparted to visible writes.

9.6.7 Eager vs Lazy

There are two basic conecepts for conflict detection, *eager* and *lazy*. 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 choosen makes an implact on many different parts of a TM.

9.6.7.1 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 implemenation use visible writes, while eager write/read conflict detection can use either visible reads or visible writes..

If visible writes are used than read/wrtie conflicts are detected more eagerly than if invisible writes are used. With visible writes the conflict is detected as soon as the wrting transaction performs its write and aquires the lock for the memory location, with invisible writes the conflict is 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 confict has occured 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 seperate 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 occured and is dealth 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 occured, and the conflict is dealt with according to the implementation.

9.6.7.2 Lazy

Read/write, write/read, and write/write conflicts can also be detected lazily. This means they are not dected until committ 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 conflit 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 occured 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 choosen consistency criterion. Since the transaction doing the write has no idea the read has occured 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 conflit 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 thier serialzation point as time of their committ 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.

9.6.8 Choosing a conflict detection scheme

The thought behind choosing lazy vs eager is choosing between future wasted work vs past wasted work. By detecting and handling a conflict as soon as possible, an eager detection scheme is attempting to avoid future wasted work by assuming the conflict and consistency criterion will most likely require that one of the transactions abort. On the other hand, wating as long as possible before detecting conflicts, a lazy detection scheme is attempting to avoid past wasted work hoping that most conflicts and the consistency criterion will not necessarily require a transaction to abort.

The differences between lazy and eager detection have been examined and benchmarked in quite a few different ways, but there is currently no obvious choice of one over the other. In fact many of the benchmakrs done so far show that the best choice depends on the worload. Certain implementations such as RSTM [**?**] are designed to support both invisible and visible reads and lazy and eager aquire, giving the programmer the freedom to choose which to use. They [**?**] run benchmarks on all different posibilities and come to the conclusion that there is no clear choice that works best in all situations.

One argument for having at least some conflicts detected eagerly is conflict managment. When a conflict is detected between two transaction is detected early then the conflict manager is able to choose what to do. Certain conflict managers have been designed to promote desirable properties so the more often and the earlier conflicts are detected the more chances a conflict manager has to promote liveness.

A recent paper [**?**] has suggested using lazy detection for increased performance. They claim that using lazy detection promotes good livness properties such as freedom from livelock because locks are only aquired during committ time and it is unlikely that two transaction will repatibly try to commit at exaclay the same time (this is given along with other reasons). They perform benchmarks to support their claims. This works along side an additional mechanism they propose to discourage starvation. Another recent paper [**?**] seems to support their claims. In this paper a mechanism called *input acceptance* is proposed which measures the ammount of histories an implementation will accept for their consistency criterion. They show that a lazy implementation will accept more histories than an eager one, and come to the conclusion that an implementation that accepts more histories is likely to provide better performance across different workloads. More about input acceptance is give later in this svrvey in the section about measuring TMs.

An implementation does not have to be all lazy or all eager, for example SwissTM [**?**] uses a *mixed invlidation* scheme. They employ lazy detection of read/write conflicts in the hope that this type of conflict will often not require a necessary abort. Write/write conflicts on the other hand are detected eagerly on the assumption that they will most likely require one of the transactions to necessarily abort.

9.6.9 Implementing Conflict Detection

The underlying mechanisms that are used to implemt visible or invisible reads and writes and lazy or eager acquire are often used as part of the reason for choosing one or the other. This section will go over some of the ways they can be implemented as well as some of the extentions that have been proposed.

9.6.9.1 Validation for invisible reads

As described previously for a TM implementation that uses invisible reads each time a transaction does a read its read set must be validated, otherwise the transaction might see an inconsistent view of the memory. Of course the validation that needs to be done depends on the consistency condition, but here we will consider opacity because it is the most widely used consistency condition. In order to perform read set validataion, every time a read occured, early STMs would check to see if every item previously read was still valid (*i.e.* it had not been overwritten), if a value had been overwritten, then the transaction would abort. Each transaction then has a quadratic cost on the number of reads to perform validation, which can drastically hurt performance [**?**].

Logical Clock / Time In [**?**] they introduce an improvement to this called the *lazy snapshot algorithm* or LSA which allows for fewer long validations. This algorithm uses a logical clock that is incremented each time a writing tranaction commits and the shared memory objects that are updated are assigned this clock value. While a transaction is active it maintains a *snapshot*, or valid range of linearization points, this range is based on the shared memory objects it has read so far. If a transaction reads a value that is valid within its snapshot, then the value is read and the snapshot is updated with no other validation required. In the case that a transaction reads a value that is not valid within its snapshot it first tries to see if it can extend the range of its snapshot, in which it checks to see if each value it has read so far is still valid (similar to the normal validation process), if this succeeds the snaphot is extended and the value is read, otherwise the transaction aborts. They show that using the LSA improves performance over invisible reads using standard validation throughout many benchmarks.

TL2 [**?**] introduced a simplier, but similar clock based scheme where read validations always occur in constant time. It works very similar to LSA without the extensions, so TL2 will abort the transactions where the extentions would have occured.

9.6.9.2 Multi-versioning

Multi-versioning was introducted in [**?**] as a way to prevent read only transactions from conflicting with any other transactions. By keeping past versions of objects a transaction only reads form the state of the memory was when it started allowing read only transactions to always commit.

Along with proposing using a clock to reduce the cost of validating invisible reads in [**?**] they also extend on the idea of multi-versioning with the use of clocks. They keep available multiple older versions of the object in the hopes of committing more read only transactions. In LSA if a snaphost cannont be extended to be valid for the most recent version of the object to be read, then an older version can be read that is within the snapshot if it is available. The more versions that are kept, the more memory overhead required by the implementation, so they suggest multiple

ways of choosing the amount of versions to keep, including dynamically choosing how many current versions to keep based on if the could be useful for any live transactions. Through benchmarks they find that keeping 8 versions seems to work best. Since then some efficency improvements have been proposed for multi-versioning such as garbage collection and more info, cite? In [**?**] they study some of the theroetical limitations of multi-versioning to be disjoint access parallel (see the section in this survey on disjoint access parallelism).

9.6.9.3 Announcing for visible reads

In order to implement visible reads a transaction must somehow annouce to other transactions that it has read this object. Normally this is done by having each shared object keep a list of live transactions that have read it. This list is one of the main arguments against using visible reads, because it is a source of contention preventing scalablility. When a transaction reads a shared object it has to add itself to this shared list, and if a variable is read often there could be high levels of contention on this list, even if these are all read only transactions and have no reason to conflict.

RSTM RSTM [**?**] tries to avoid this contention by keeping a limited number of visible readers in the header for each shared object, if there is a spot open a reading transaction can just perform a compare and swap opertaion to place a pointer to itself in the header. If there are no spots open then the transaction will read the object invisibly. A transaction will only have to validate the set of reads that it was not able to do visibly.

9.6.9.4 Semi-visible reads

The idea behind semivisible reads [**?**] is to avoid the scalibility problems of visible reads, while reducing the cost of constantly validating the read set necessary for insiible reads. It is implemented as an additional mechanism on top of invisible reads. A read counter is assigned for each shared memory object, and each time a transaction reads an object, it increases its counter, when a transaction that writes to this object committs, it resets this counter to zero. In addition to the read counter there is a global counter used for two things. First when a transaction starts it reads and stores the value of this counter. Second when a transaction performs a write it increments this counter if the read counter for any of the objects it is writing are non-zero. Now when a transaction performs validation it checks to see if the value of the gobal counter is different than the value it had stored, and if it is unchanged then the transaction knows none of its reads have been invlidated so it can continue. Otherwise it performs the normal valididation for invisible reads. If validation succeds then it updates its stored value of the global counter to the current value. If validation fails it aborts. They expect in many read dominated workloads this will keep transactions from having to do the expensive validation process very often. Note that they also introduce a *scalible non-zero indicator* or SNZI as a replacement for the read counter at each shared memory object that is more scalible then a traditional counter.

9.6.10 Scalibility

As the number of cores on a processor keeps increasing every year, the scalibility of a transactional memory implementation gets more and more important. By design certain programs might not be able to scale well, and a programmer should take care to not create such programs,

but he should not have worry that his program will not scale due to the implementation of the underlying TM system. As mentioned in the section on read visibility, having visible reads is worried to harm scalibility, but it is also possible for any other component of a TM implementation to become a bottle neck for scalibility and it is important examine where these happen. Following this, two similar concepts, *disjoint access parallelism* and *conflict-based synchronization*, have been introduced as concepts for STM implementations to follow in order be scailble, but it is still unknow in many ways what exactly makes a TM scalible or not.

9.6.10.1 Conflict-based Synchronization

Certain recent STM designs have been concerned with scalibility such as SkySTM [**?**], where in designing the system they take a *conflict-based* approach to synchronization in order to promote scalibility. Conflict-based synchroniztion is defined as where contention on STM medatadata in induced only (or at least primarily) when there is contention on application data. If this goal is accomplished then when an application is not scaling well it is the fault of the application or the workload, and not the underlying STM implementation.

9.6.10.2 Disjoint Access Parallelism

As a general definition, in order for a STM implementation to be *disjoint access parrallel*, transactions that do not concurrently access the same shared memory location must not interfere with eachother. For example if every transaction accesses a global counter at creation to get its start time, then the counter becomes a point of centention for all transactions and the TM implementation is not disjoint access parallel. It is not always possible for a TM to be disjoint access parrallel and still be implemented in a desired way, in [**?**] they examine some of these limitations. Specifically they show that it is not possilbe to have an implementation with invisible reads and read-only transactions that always commit and still be joint access parallel. They also show that a disjoint access parrallel implementation with read only transaction that always commit must write to meta data a number of times at least in the order of number of objects it reads for any transaction. This could be used as an argument against multi-versioning (which allows every read only transaction to commit), because ether these implementations can have visible reads and not be disjoint access parrallel, limiting scability, or they can be disjoint access parallel and perform quite a bit of work for read only transactions, which might be too much overhead for read dominated workloads which thought of as common. Although disjoint access parallelism is an viewed as an interesting property to study and a crucial component of scailible transactional memory, many of its other theroetical limitations remain unknown.

9.7 Measuring Efficiency

Apart from the progress guarantees such as liveness, and starvation freedom, and the cost of implementing different components, measuring efficiency an important part of gaging how effective a transactional memory is. There are multiple ways to measure efficiency looking at different parts of a TM and its execution.

9.7.1 Commit-abort Ratio

The *commit-abort ratio* is defined in [**?**] as the ratio of the number of committing transactions over the total number of complete transactions (committed or aborted) for some execution of a workload. Being able to commit a lot of transactions while aborting very might be a good goal for a TM to achieve, but it does not necessarily mean a efficiency. For example an implementation could just only run one transaction at a time and have a perfect commit abort ratio.

9.7.2 Makespan

The *makespan* is the amount of time it takes a given TM implementation to complete a workload. The measurement is often used on benchmarks to compare one TM to another, but just because a TM has a good makespan for one workload does not mean it will do well on another. This is a common problem with STM implementations, often when they are introduced they are only tested on a limited set of benchmarks showing good performance, then some new implementation comes along showing the old implementation has poor performance on a different set of benchmarks and this new implementation is better, and the process repeats.

9.7.3 Competitive Ratio

Competitive ratio is the ratio between the makespan of two different implementations. This can be used to compare two different TM implementations.

9.7.4 Throughput

The *throughput* of a TM is the amount of transactions it commits per unit time and is commonly used in benchmarks to compare TMs. This can be especially useful on workloads with equal size transactions to compare the cost of the meta data computations that different TMs use. A shortcoming of this measure is, for example a TM that favors short transactions over longer ones might have high throughput, but low progress compared to one that might sometimes abort short transactions in favor of long ones.

9.7.5 Scalability

It is important that a transactional memory implementation scale to multiple threads as they are designed for multi-core processors. Other measures such as throughput and makespan are often crossed with number of threads for testing *scalability* when performing benchmarks. It is not an obvious thing to measure scalability though. For one, workloads themselves might only scale to a certain point. Also certain mechanisms used in a TM to provide it with some good property, such as obstruction freedom, might limit scalability to a certain amount, or some implementation choice, such as visible reads, might make it fast only up to a certain point. The obvious optimal implementation might be able to scale infinitely, but this might not always be possible given some implementation choices, or even necessary for the hardware it is used on. It is also interesting to look at what mechanisms and properties of TMs limit or encourage scalability. Some of this has been examined in [**?**] and [**?**]. should write more on these and need to read the second one

9.8 Measuring Properties ensured by a TM

Other then measuring efficiency there are ways to measure a TM by certain properties that it ensures.

9.8.1 Blocking and Liveness

Earlier in this survey different blocking and liveness conditions were discussed, and they are also used to gage the capabilities of a TM. For example many TMs use locks and even though locks do not provide the good properties of a non-blocking TM, they can be more efficient. Or a TM that is shown to be very fast on certain workloads but not proven to avoid livelock might not be preferable to a slower TM that it lock-free for certain applications where progress is essential. There is no clear choice as to what is always best, and it could depend on the application or workload choice.

9.8.2 Bounds on performance

As discussed in the efficiency section, the competitive ratio can be used to compare two TMs. Competitive ratio is used in [**?**] and [**?**] by comparing the makespan of a clairvoyant implementation (one that knows all the characteristics of the workload beforehand) with realistic implementation to prove bounds over all workloads. This can be important because even though a TM might be efficient on certain benchmarks, there might be some worst case workloads where it performs much worse than another TM. It is interesting to know just by general characteristics of a TM implementation, such as how visible its reads are, what possible bounds on performance it has. Note that proving some theoritecial bounds does not actaully mean a TM will be efficient in practical cases, the bookkeeping required may make the TM too slow for many real world applications. It might be interesting to see what bounds on bookkeeping a TM would need in order to ensure certain properties or perform certain operations.

9.8.3 Conflict/Arbitration Functions

Different TM implementations have different ways of defining and identifying a conflict. For example some implementations might identify a conflict as soon as two concurrent transaction access the same memory location, while others might wait until when a transaction tries to commit. In [**?**] they define a *conflict function* as a way to characterize how a TM identifies conflicts. The conflict function has 3 inputs: a total order history of an execution (containing read, write and commit events), and two transactions. And a single boolean output: true if the two transactions conflict, otherwise false. They define six different conflict functions and show how the sets of conflicts they identify are related. For example the set of conflicts identified by a function that returns true whenever two concurrent transactions access the same object, is contained within set of conflicts from a function that returns true whenever two transactions are concurrent. This is interesting because identifying a smaller set of conflicts means that there is less often a reason to abort a transaction.

When a conflict occurs between two transactions in a TM, most implementations will abort one transaction, in [**?**] they define an *arbitration function* as a function that decides which of the conflicting tractions to abort. It takes as input a total ordered history, and two transactions T1 and T2, and outputs true if the transactions conflict and that T1 should abort, and false otherwise.

They use these functions as a way of characterizing contention managers. The way in which a certain arbitration function is defined can ensure liveness properties such as starvation freedom, livelock freedom, and nonblocking.

9.8.4 Input Acceptance

In [**?**] they compare TM implementations based on the *input patterns* they accept. The *input history* of a TM on some workload consists of the total order of all read, write, and commit (where commit is the execution of the try to commit operation, it could return success or abort) operations. An *input pattern* is some total order of read, write, and commit operations. A TM *accepts* an input pattern if when it executes this pattern it commits all transactions. In [**?**] they use regular expressions to represent classes of input patters with special wild cards for example, star '*' meaning zero or more of the preceding event, and compliment '$\neg$' meaning anything but the preceding event. An example of an *input class* would be the following:

$$C_1 = \pi^*(r_i w_i | w_i r_i)\pi^*$$

Here π represents any operation, so input class C_1 would be any sequence of events followed by either a read then write by transaction i or a write then read by transaction i, followed by any sequence of events. A TM *does not accept* an input class if for every input pattern of this class, it aborts at least one transaction. They use different input classes to upperbound the input acceptance of different TMs based on their characteristics. A higher upper bound means that at TM is able to accept more input patterns. By observation it seems that a higher input acceptance would mean that the implementation is more likely to have a higher *commit-abort* ratio across most workloads and this assumption is supported by some benchmarks they did.

What also might be valuable would to be show that certain workloads or classes of workloads are likely to have a high occurrence of some input pattern. This can then be matched to an input acceptance class of a TM implementation to say the implementation would be good or bad to use with the given workloads. It would also be interesting to see if input acceptance can be connected in any way liveness or other conditions. In addition, input acceptance as it stands is not a throughogh benchamrk as does not capture everything about a TM such as how much additional work it is doing on meta data, or things like waiting. Note that input acceptance is similar to the idea of a conflict function [**?**]. A conflict function says some pattern will cause a conflict, while an input acceptance defines a pattern that cannot occur because it would have caused a conflict. An important difference is that in [**?**] where conflict functions are defined, transactions must be serialized at their commit point in real time which makes the consistency condition more strict than opacity, but input acceptance is examined on various consistency properties, including serializability.

9.8.5 Obligation

In [**?**] a safety property called *obligation* that can be used on transactions is defined. If a TM satisfies some given obligation property, then every transaction that satisfies that given property must commit. Safety properties such as opacity ensure some level of consistency, but they do not prevent unnecessary aborts. Liveness and non-blocking properties such as lock freedom and livelock freedom guarantee some sort of progress, but still do not prevent unnecessary aborts. A trivial example of this is a TM implementation that always commits transactions running on a given core, while aborting every transaction on every other core. This is similar to just running

the program sequentially, and is obviously opaque and as can even be wait-free. In order to avoid unnecessary aborts, obligation can be used in addition to other safety properties ensuring consistency, and liveness.

In [**?**] they give two obligation properties, and define a TM that satisfies the properties. Other that just defining when a transaction must commit, they prove that a TM that implements this two properties contains only transactions that see consistent views of the shared memory, and are atomic. It could be interesting to look at what other guarentees can obligation properties ensure, for example other consistency properties, or liveness? Also, are the properties proposed in [**?**] too strong? *i.e.* Do they prevent certain types of TM implementations? Note that obligation is similar to the idea of input acceptance and conflict functions, except it is looked at from the point of view of where a transaction will commit vs where a transaction will conflict.

9.8.6 Permissiveness

A TM that is *permissive* [**?**] will only abort a transaction if by committing that transaction the chosen correctness criterion will be violated, these unnecessary aborts are called *spare aborts*. In fact most TM implementations have contention managers that will abort at least one transaction whenever there is a conflict, which it likely too often. In [**?**] it is shown show that for any deterministic TM that aborts at least one transaction on a conflict, there exist workloads where the TM will execute at the parallelism of a single core (due to a large number of aborts), when the optimal (clairvoyant) contention manager and transactional ordering will execute maximumally parrell on all cores in the system. Creating a TM that does not allow any or even few spare aborts is difficult due to the amount of meta data needed to keep track of all transactions that might conflict, especially when considering more relaxed consistency criterion such as serializability where transaction ordering needs not to respect real time order. In [**?**] they look at the problem of spare aborts in depth and show some limitations of TMs and preventing spare aborts. need to finish reading this one

An interesting approach to avoiding spare aborts is taken in [**?**] where they create a TM, called AVSTM, that is *probabilistically permissive* with respect to opacity. This means that it has a positive probability (greater than zero) to accept any history that is opaque. This is accomplished by whenever a transaction tries to commit, a valid sterilization point is randomly chosen from any time within its execution window (most implementations will simply chose the time of commit as the serialization point). The basic reason why this allows any opaque history to be possibly accepted is because in any valid opaque history a transaction is serialized to sometime within its execution window, so there is a positive probability that this point will be chooen randomly by AVSTM.

For an online TM to keep track of what that serialization point should be in order to avoid spare aborts is expensive, the only TM they know that does this requires $O(n^2)$ cost to perform operations. By randomly choosing a serialization point they avoid this cost while still having the possibility of avoiding spare aborts. AVSTM is shown to exhibit good performance in selected benchmarks, while performing poorly in others, but it is important to note that AVSTM is implemented as lock-free and it is compared to implementations that are obstruction free or use locks so it is hard to tell what is causing the performance differences. It might be interesting to see if AVSTM can be implemented as obstruction-free or with locks.

They also suggest the idea of k-permissiveness, which is defined as the maximum ratio of spare aborts to the total number of transactions for any history that satisfies the consistency condition as a way to compare non-permissive TMs. It is not exactly clear how to measure this

though. Input acceptance [**?**] is a step in this direction because it gives patterns of histories that a TM does not accept, and TMs can be compared on how general the patterns they accept are, but this does not consider number of aborts.

9.9 Implementations

This section introduces some of the interesting features of some popular STM designs. [**?**] gives an introductory overview of the algorithms and principles behid three different STM implementations including TL2 and JVSTM.

9.9.1 DSTM

Dynamic STM or DSTM [**?**] was introduced as the first dynamic software transactional memory, meaing the memeory accesses made by a transaction did not need to be defined beforehand by the programmer. It also introduced the idea of *contention managment* and *early release*. It is obstruction-free.

9.9.2 ASTM

Adaptive STM or ASTM [**?**] was designed with the ability to dynamically adjust how it performs underlying operations based on the workload in order to improve performance. It has the ability to change between eager and lazy acquire for writes, and change metadata structure between direct or indirect object referencing. It is obstruction-free.

9.9.3 RSTM

RSTM [**?**] is an obstruction-free implementation designed with the idea of improving the performance of non-blocking TMs. It does this by organizing sharded data and transactional metadata in memory efficently, as well as using visible reads designed for low contention in combination with invisible reads.

9.9.4 TL2

Transactional Locking 2 or (TL2) [**?**] was developed as an efficient TM that ensures opacity. It uses invisible reads along with a global clock in order to perform constant time read set validation. It is implemented using locks.

9.9.5 JVSTM

With the goal of improving concurrency JVSTM [**?**] introduces multi-versioning allowing a read-only transaction to never conflict with any other transaction. It uses locks.

9.9.6 TinySTM/LSA-STM

TinySTM and LSA-STM [**?**] introduce the lazy snapshot or LSA algorithm which uses a logical clock. This allows invisible reads with mostly constant time read set validation and less aborts than TL2. LSA-STM is obstruction free and can use multi-versioning. TinySTM uses locks.

9.9.7 McRT-STM

McRT-STM [**?**] was designed based on testing many different impementation aspects of sn STM in order to choose the most efficient ones. It supports closed nested transactions and lets the programmers use retry and orElse constructs. It uses locks.

9.9.8 SwissTM

SwissTM [**?**] is a recent STM designed through a process of trial and error in order to find the best performance across a wide range of workloads. It uses a mixed eager/lazy conflict detection scheme, and a two phase contention manager intended to put no overhead on read-only and short read-write transactions while insuring the progress of larger transactions. Its performance is shown in a wide range of benchmarks in [**?**] and [**?**]. It is implemented using locks.

9.9.9 SkySTM

SkySTM [**?**] is a recent implementation designed especially for scalibility and performance. It introduces *semivisibile* reads as well as *conflict* based waiting to implement privitization. It is implemented using locks.

9.9.10 STM Haskell

STM Haskel [**?**]l is an implementation designed in and for the Haskell programming language. It provides some interesting properties such as strong isolation, composable transactions, disallows I/O from within transactions, as well as the language constructs *retry* and *orElse*.

9.10 Conclusion

Software transactional memory has been widely studied and many problems have been solved since the first STM was desribed in 1995 [**?**]. Still though many problems remain unsolved and many new questions have been introduced since then and because of this STMs are not yet widely used in practice. Yet it is still a promising solution to the difficulty of writing concurrent programs and there is much reserach still to be done.

List of Publications

International Conferences Articles

Technical Reports