Optimizing Software Transactional Memory Replication via Speculation

João Fernandes
joao.fernandes@ist.utl.pt

Instituto Superior Técnico
(Advisor: Professor Luís Rodrigues)

Abstract. Software Transactional Memories (STMs) are emerging as a potentially disruptive programming paradigm. Due to scalability and fault-tolerance requirements, Distributed STMs (DSTMs) are receiving more attention. Database management systems and STMs share the same key abstraction: atomic transactions. However, database and memory transactions have very distinct characteristics. Database replication protocols do not provide proper performance when applied to STMs. This work studies existing techniques to maintain the consistency of the STM replicated data, identifies limitations on those techniques, and proposes the architecture for building a novel replication protocol for STMs, that aims to overcome the limitations of current techniques by taking advantage of available computational resources to perform speculative operations. We end with an evaluation methodology for the future protocol implementation.

1 Introduction

Moore's law [1] has driven a steady increase in single processor performance in the past decades. Because of that, it has been possible to increase the complexity of sequential programs while maintaining or even improving their perceived performance. Unfortunately, the performance of a single core processor has reached a plateau. On the other hand, the number of cores available in a single chip is now increasing. Therefore, in order to keep providing acceptable performance, while adding functionality to programs, we need to use parallel programming techniques. In this scenario, Transactional Memory (TM) can become a key abstraction, as it aims to simplify concurrent programming.

When using a Software Transactional Memory (STM), the programmer is freed from having to explicitly deal with concurrency control mechanisms. Instead, the programmer only has to identify the regions of code that need to access or modify shared data in an atomic fashion. Given that low level concurrency control mechanisms such as locks are known to be extremely error prone [2], the use of STMs has the potential to increase the reliability of code and shorten the software development cycle.

Although STMs were first proposed for cache-coherent shared memory architectures, the need to increase the scalability and fault-tolerance of STM-based

systems motivated the development of STMs for distributed memory architectures and replication protocols to employ on them. Replication protocols are responsible for coordinating the replicas in a way that ensures the consistency of the transactional memory at all nodes.

Committing a transaction on a replicated Distributed Software Transactional Memory (DSTM) system is a relatively slow operation when compared to the execution time of the transaction. Since the replication protocol has to guarantee the consistency of the transactional memory on all replicas, it has to handle the synchronization problems that arise from concurrent modifications made to the transactional memory by transactions that run on different replicas. One way to tackle this problem is to use a global agreement protocol to establish a global serialization order that becomes the order for processing transactions at all replicas. The establishment of a global serialization order is a procedure that takes a non-negligible amount of time to conclude. Therefore, such an amount of time may be used to perform ahead computation that can be rolled back, like memory transactions, and may or may not become useful, depending of the fate of the ongoing commit procedure of previous transactions. This procedure is called speculative execution.

In this report we propose the development of a novel replication scheme for STMs that takes advantage of speculative execution of memory transactions in order to boost performance. Consider a sequential program that is composed of two memory transactions, T_a and T_b that are executed one after the other, respectively. With current replication protocols, when T_a concludes its execution and tries to commit, the execution flow stalls. Therefore, T_b has to wait for the conclusion of T_a's commit phase to be able to start its execution. However, since transactions can be rolled back, we propose starting the execution of T_b speculatively, based on a state that includes the expected outcome of T_a. If T_a eventually aborts, then T_b has also to abort. However, on the best case scenario, T_a commits and therefore, T_b's execution phase and T_a's commit phase overlap. To the best of our knowledge, no existing DSTM system employs the described speculative process. This work studies the state-of-the-art in replication protocols for software transactional memories, their roots, and how to boost the performance of DSTM systems.

The rest of the report is organized as follows. Section 2 briefly summarizes the goals and expected results of our work. In Section 3 we present all the background related with our work. Section 4 describes the proposed architecture to be implemented and Section 5 describes how we plan to evaluate our results. Finally, Section 6 presents the schedule of future work and Section 7 concludes the report.

2 Goals

This work addresses the problem of optimizing the performance of replicated software transactional memory systems. Namely:

Goals: This work aims at analyzing, designing, and implementing a replication protocol for software transactional memories that operates efficiently under heavy and heterogeneous workloads.

For many benchmarks and workloads, the time that takes to commit a memory transaction in a distributed software transactional memory is typically much longer than the time that takes to execute the transaction [3]. We propose the use of a replication protocol that commits transactions in a speculative fashion. The speculative procedure will allow the application to continue its execution while the global commit phase is still ongoing in background, which results in an effective overlap between transaction processing and communication. The STM replication manager is responsible for ensuring that the application-level never sees an inconsistent state outside the transactional code.

We will develop an implementation of the proposed protocol for the Fénix framework [4]. The implementation will be evaluated with well established benchmarks for transactional systems.

Expected results: The work will produce (i) a specification of the protocol; (ii) an implementation for the Fénix framework; (iii) an experimental evaluation using well established benchmarks for transactional systems.

3 Related Work

In order to understand the underlying problems that this project poses, we need to review the fundamentals of various areas in distributed systems. Section 3.1 presents a review on replication techniques and mechanisms. It addresses basic concepts, such as passive and active replication, but goes beyond that, covering fundamental group communication primitives and database replication, as they share the same key abstraction with TM: atomic transactions. Section 3.2 addresses software transactional memory. It describes the motivations, properties and concepts of transactional memory systems, compare the pros and cons of hardware assisted, pure software and hybrid approaches, discuss implementation decisions and concludes with STM proposals. Finally, in Section 3.3 we address distributed software transactional memory. We describe the motivations, requisites and challenges for STM replication, and conclude with DSTM proposals.

3.1 Replication Techniques and Mechanisms

3.1.1 Basic Concepts

Replication is one of the main techniques to build highly available, fault-tolerant systems. There are two main models to ensure consistency among replicas: passive and active replication.

Passive replication, also known as master-slave replication or primary-backup, is characterized by the existence of a replica, known as the master, that processes

all requests and transfers the state updates to the remaining replicas, known as the slaves or backups. In most cases, slaves can also process read-only transactions. Passive replication often assumes the fail-stop model. When the master fails one of the slaves is elected to replace it, becoming the new master. One of the problems of this technique is that in write intensive workloads the master may become a bottleneck in the system, as it is responsible for all the computation, since the slaves do not share any workload.

Active replication is characterized by having all replicas processing the same sequence of requests, exactly by the same order. In order to ensure replica's consistency, active replication requires operations to be deterministic, i.e., the state of a replica should depend exclusively on the initial state and on the sequence of operations executed. To ensure that all replicas process the same requests in the same order, a total order reliable broadcast primitive, also called Atomic Broadcast (AB) [5], is typically used. This group communication primitive is typically provided by a Group Communication System (GCS), such as the Appia framework [6], that includes membership and communication services. One of the limitations of the approach is that AB is an expensive communication primitive, as it requires consensus to be reached among replicas. Also, since all replicas execute all update requests, it does not provide scalability for update requests. Like primary-backup replication, read-only requests can be executed in parallel at different replicas.

Other replication schemes combine aspects of the two previous techniques. An important example is certification-based replication, a scheme that is commonly employed on transactional systems and relies on group communication to validate state changes across replicas. An example of a certification-based protocol is described in Section 3.1.3.

3.1.2 Group Communication

Group communication [7, 8] is a powerful paradigm for performing multipoint to multipoint communication by organizing processes in groups. Typically, a system that implements this paradigm is called a Group Communication System (GCS) and offers membership and reliable broadcast services with different ordering guarantees. GCSs allows programmers to concentrate on what to communicate rather than on how to communicate. Our focus will be on view-synchronous GCSs [9].

View-synchronous GCSs model the system as a group of n processes $\prod = \{p_1, ..., p_n\}$. The *group-membership service* exports *join*(S) and *leave*(S) primitives (where S is set of processes such that $S \subset P$) and runs a failure detector to discover faulty processes. It outputs a sequence of group membership sets called *views*. Every view $V \subseteq P$ is delivered through a *v-change*(vid, V), where $vid \in \mathbb{N}$ denotes a monotonically increasing *view identifier*. When this occurs, we say that the process *installs* the new view V.

Apart from handling explicit join and leave operations, a membership service also plays the role of failure detector: it excludes crashed processes from the group membership and detects the "stable" components of the system, i.e., the set of

processes that are correct and that can reliably communicate with each-other. A *stable* component is defined as a set of processes that is eventually permanently connected.

Reliable broadcast services can provide various types of ordering disciplines, such as FIFO, Causal and Total Order. FIFO ordering guarantees that messages from the same sender are delivered in the order in which they were sent at every process that receives them. Causal ordering extends FIFO ordering and guarantees that if message m' is causally related to message m (according to Lamport's "happened-before" relation [10]), then m is delivered before m' to every process that receives them. Total ordering guarantees that all messages are delivered in the same order to every receiver. We will now cover in more detail the properties of Atomic Broadcast.

Atomic Broadcast Atomic Broadcast (AB) is a reliable broadcast that ensures that every participant receives all messages by the same order. It can be defined by two primitives:

- **AB-cast**(m) broadcasts the message m to all sites in the system.
- **AB-deliver**(m) delivers message m definitively to the application. The sequence of AB-delivered messages is called the *definitive order* and is the same at all sites.

There are two types of AB: regular and uniform. Uniform AB ensures the following properties:

- **Validity:** if a correct process AB-broadcasts a message m, then it eventually AB-delivers m.
- **Uniform Agreement:** if a process AB-delivers m, then all correct processes eventually AB-deliver m.
- **Uniform Integrity:** for any message m, every process AB-delivers m at most once, and only if m was previously AB-broadcast by its sender.
- **Uniform Total Order:** if processes p and q both AB-deliver messages m and m', then p AB-delivers m before m' only if q AB-delivers m before m'.

Validity and agreement properties are liveness properties, i.e., they ensure that something "good" eventually happens. In turn, integrity and total order properties are security properties, i.e., they ensure that nothing "bad" happens.

Regular AB has the same validity and integrity properties as uniform AB, but has the following non-uniform properties:

- **(Regular) Agreement:** if a *correct* process AB-delivers a message m, then all correct processes eventually AB-deliver m.
- **(Regular) Total Order:** if two *correct* processes p and q both AB-deliver messages m and m', then p AB-delivers m before m' if and only if q AB-delivers m before m'.

Uniform properties make life easier for application developers, as they apply to both correct and faulty processes. However, enforcing uniformity often has a cost and for this reason it is important to consider whether uniformity is strictly necessary. Non-uniform properties can lead to inconsistencies at the application level if they are not considered properly and, therefore, applications that rely on them should be prepared to take the necessary corrective actions during failure recovery.

Optimistic Atomic Broadcast Optimistic Atomic Broadcast (OAB), is defined by all the primitives of AB plus one more:

- **OAB-deliver**(m) delivers a message m optimistically to the application. OAB-deliver does not guarantee total order. The sequence of OAB-delivered messages is called the *tentative order*.

OAB is a very important building block for high-performance distributed systems. The OAB-deliver primitive enables applications to overlap computation with communication: messages are OAB-delivered as soon as they arrive and when their final order is known, they are AB-delivered. Between these two events, applications can perform optimistic, possibly useful computation. When a message is AB-delivered, one of two possible scenarios occur: either the message's final order matches the tentative order or it does not. In face of the first scenario, the optimistic computation performed was useful and so its modifications to the application's state can be applied safely. However, in face of the second scenario, the optimistic computation performed was useless and, therefore, all its modifications should be rolled back.

OAB has all the properties of AB plus one more:

- **Optimistic Order:** if a process p AB-delivers message m, then p has previously OAB-delivered m.

Aborting and rolling back modifications has also a cost, so the use of OAB is suitable for network environments where the percentage of out-of-order messages is low [11].

3.1.3 Database Replication

Database management systems (DBMS) are a key component of many information systems. As with other applications, replication is supported in a DBMS system to provide increased fault-tolerance and scalability (namely of read-only requests). We shall begin with the description of a number of concepts related to transactional systems:

- **Read-set:** the set of data items that are read by a transaction.
- **Write-set:** the set of data items that are written by an update transaction.
- **Data-set:** the union of the read-set and write-set of a transaction.

– **Commit:** the transaction finishes successfully, therefore its modifications are made visible.
– **Abort:** the transaction was unable to finish successfully, therefore are not made visible or undone if necessary.
– **Conflict:** two concurrent transactions conflict if the write-set of one overlaps with the data-set of the other.

Database transactions satisfy *atomicity*, *isolation*, *consistency* and *durability* properties. These are commonly referred to as *ACID* properties [12] and feature the following characteristics:

– **Atomicity** ensures that modifications must follow an "all or nothing" rule, i.e., either all the modifications made by a committed transaction are made visible or none is.
– **Consistency** ensures that each transaction transform the database from one consistent state to another consistent state.
– **Isolation** ensures that individual memory updates within a memory transaction are hidden from concurrent transactions.
– **Durability** ensures that once a transaction is committed, its updates will survive any subsequent malfunctions.

During its life-cycle, a transaction can pass through four distinct, well-defined states: *executing*, *committing*, *committed* and *aborted*. Both the executing and committing states are transitory, while both the aborted and committed states are final.

– **Executing:** the transaction's operations are executing.
– **Committing:** the transaction has completed the execution of its operations, and therefore, the client requested the transaction's commit.
– **Committed:** the transaction was committed.
– **Aborted:** the transaction was aborted.

In the next paragraphs we describe one of the most relevant database replication protocols ever proposed.

The Database State Machine Approach In [13], Pedone et al. propose a replication scheme that is designed to synchronize a cluster of database servers, in a multi-master environment. This scheme is based on the state machine approach [14]. However, unlike classical state machine replication, a transaction is not executed at all replicas. Instead, a transaction is executed at a single node and, upon commit, its data-set is atomically broadcast to all replicas, starting the *certification* phase of the transaction. Certifying a transaction consists on detecting conflicts and checking that its commit does not violate *one-copy serializability* [15]. It decides to abort a transaction if the transaction's commit would lead the database to an inconsistent state (i.e., non-serializable). This step is executed by all replicas in a deterministic manner, i.e., it produces the same

decision at all replicas. This technique reduces the cost of inter-replica coordination, by executing a single interaction when the transaction commits. It also simplifies recovery as intermediate results of a transaction are never propagated and, therefore, there is no need for implementing distributed rollback schemes. The main drawback of this technique is that the lack of synchronization during transaction execution may lead to a large transaction abort rate in face of workloads with many conflicting transactions. For this reason, the authors propose an optimization technique called *reordering certification test*. The optimization is based on the observation that the serial order in which transactions are committed does not need to be the same order in which transactions are delivered to be certified. Therefore, in order to reduce the abort rate, once a transaction T_a passes to the committed state it acquires the needed locks over the items it will update but does not perform those updates immediately. Instead, the transaction T_a is placed in a buffer called the *reorder list*, that has a defined size called the *reorder factor*, in a position p such that any given transaction T_b at position $pos(t_b) < p$ precedes or is commutable with T_a and any transaction T_b at position $pos(T_b) \geq p$ does not read from any location that is updated by T_a and, neither precedes T_a nor updates any location that is read by T_a. When the reordering factor is reached, the leftmost transaction T_a in the reorder list is removed, its updates are applied to the database, and its write locks are released.

The size of the reorder list has to be configured with carefully. Since all transactions in the list already have their locks granted, data contention rises considering that the items stay locked for a longer period of time, which can lead to an increase on the abort rate that is greater than the reduction obtained with its use.

3.2 Software Transactional Memory Systems

Concurrent programs make use of multiple threads of control (named threads or processes) to execute multiple sequences of instructions in parallel. Concurrent access and modifications of shared data by these threads can cause inconsistencies that may lead to incorrect states. For this reason, access to shared data has to be subject to some sort of concurrency control mechanism. Classic concurrency control mechanisms, such as locks, can ensure mutually exclusive access on data. Unfortunately, locks are hard to use in a correct manner as a single misplaced or missing lock can easily create an error. Indeed, locks comprise so much complexity that even when placed in a way that ensures mutual exclusion, they can still lead to undesired effects, such as: deadlock, priority inversion, poor performance in face of preemption and page faults, lock convoying or incorrect behaviors (e.g., infinite loops). To avoid these problems, the programmer needs to have a deep knowledge of low-level properties of the execution environment and a complete knowledge of all the concurrent execution flows in the application, which can be cumbersome.

Transactional memories support the *transaction* abstraction, as a high-level concurrency control mechanism. Transactions release the programmer from explicitly dealing with the low-level details of concurrency control. When using a

TM system, the programmer only has to identify the regions of code that need to access or modify data in an atomic fashion. TM allows programmers to express what should be executed atomically, rather than requiring them to specify how to achieve such atomicity. This translates into what is argued to be the main advantage of TM: *composability* [16]. Unlike locks, TM enables software composition, i.e., correctly-implemented concurrency abstractions can be composed together to form larger abstractions.

3.2.1 Concepts and Properties

Memory transactions have many similarities with database transactions [12]. The concepts of *read-set*, *write-set*, *data-set*, *commit*, *abort* and *conflict*, as well as the transaction's life-cycle, all described in Section 3.1.3, hold for memory transactions. Although similar, memory transactions and database transactions are not equal. Memory transactions satisfy only both *atomicity* and *isolation* properties (also described in Section 3.1.3).

Data Version Management By ensuring a strong consistency model like *serializability*, a transactional memory system requires that *data version management*, in STM systems often called *update strategy* or *update policy*, is implemented. It is necessary so that the system can maintain both old values of data items (i.e., valid when the transaction starts) that are needed if the transaction aborts, and new values of data items (i.e., uncommitted values written during the execution of a transaction) that are needed when the transaction commits.

There are two basic approaches: *lazy* and *eager* data versioning. *Lazy data versioning*, also called *deferred update*, works by performing all writes in a transaction in a private buffer until the transaction commits, after which they are applied. *Eager data versioning*, also called *direct update*, performs all writes within a transaction directly in memory, while the old values of the data items are stored in an undo log that is needed if the transaction aborts. Where eager data versioning is optimistic and favors fast commits at the price of slower aborts, lazy data versioning is pessimistic and favors fast aborts at the price of slower commits.

Concurrency Control A transactional memory system must also implement *concurrency control*, i.e., *conflict detection and resolution*, in order to detect and handle conflicts between concurrent transactions accessing (and at least one updates) the same data item(s).

Similarly to data versioning, there are two basic approaches: *lazy* (i.e., late) and *eager* (i.e., early) conflict detection and resolution. *Lazy conflict detection and resolution* detects conflits at commit time, i.e., the conflict itself and its detection occur at different points in time. Upon commit, the write-set of the committing transaction is compared to the read and write-sets of other transactions. In case of a conflict, the committing transaction succeed (i.e., commit) and the other transactions abort. *Eager conflict detection and resolution* checks for conflicts upon each load/store operation, i.e., the conflict is detected as soon

as it occurs. In software, this is typically done by using locks and/or version numbers. Where eager conflict detection is pessimistic and favors the execution of less useless operations at the cost of slower read and write operations (plus data contention), lazy conflict detection is optimistic and favors execution speed at the cost of performing more useless computation (plus longer undo work).

The *granularity* of conflict detection is a key design decision in a TM system, since the read and write-sets are maintained for the data items at the conflict detection granularity. On STM systems, conflict detection is usually done at *the word granularity* or *object granularity*, although other granularity may be chosen. Tracking read and write-sets at the word granularity ensures that no false sharing occurs, i.e., no false conflicts occur because they are detected upon concurrent access to the *same* memory addresses. However, this approach introduces higher overhead in terms of time and space (state information). Tracking read and write-sets at the object granularity matches the programmer's reasoning, has low overhead in terms of time and space and is suitable for software and hybrid TM implementations. However, there is a risk of false sharing on large objects, which may lead to unnecessary aborts.

Nested Transactions A nested transaction is a transaction that has one or several transactions inside of it. There are two types of support for nested transactions: *closed* or *open* nested transactions [17]. Closed nested transactions extend atomicity and isolation of an inner transaction until the outermost (top-level) transaction commits, whereas open nested transactions allow a committing inner transaction to release isolation immediately, which will potentially result in higher parallelism at the cost of more complex hardware and/or software.

Strong and Weak Atomicity *Weak atomicity* [18] is when non-transactional code can read non-committed updates, while *strong atomicity* [18] is when non-committed updates cannot be read from the outside of a transaction. Although strong atomicity provides a simple and intuitive model to the programmer, it may be difficult to implement efficiently. It is also important to note that applications that assume and execute correctly under weak atomicity do not necessarily execute correctly under strong atomicity as shown by Martin et al. in [18].

Opacity *Opacity* [19] can be viewed as an extension of the classical database *serializability property*, with the additional requisite that even non-committed transactions are prevented from accessing inconsistent states (DBMSs only guarantee that committed transactions do not see inconsistent states). It is an important safety property because memory transactions may be coded in a wide range of programming languages and might not be executed on a sandboxed environment. The lack of this property can lead to exceptions or incorrect behaviors on otherwise correct code (e.g., infinite loops).

3.2.2 Hardware vs Software vs Hybrid Transactional Memory

Currently, there are several proposals for *hardware transactional memory* (HTM) [20–24], purely software based ones, i.e. *software transactional memory*

[25–29], and *hybrid transactional memory* (HyTM) schemes that combine both hardware and software [30–32].

Regardless of the layer where the implementation is made, most TM implementations support *unbounded* and *dynamic* transactions [33]. Supporting unbounded transactions (as opposed to *bounded* transactions [33]) means that there is no limit on how many items a transaction can read or modify. Supporting dynamic transactions (as opposed to *static* transactions [33]) means that there is no need to known the transaction's data-set a-priori, since it is determined at runtime.

Some HTMs, such as the one proposed by Herlihy et al. in [22], only support bounded, static transactions. This forces the programmer to be aware of HTM limitations and to write the code in a way that circumvents these limitations, something that contradicts the goal of simplifying the programming of concurrent programs, as promised by the TM paradigm. Providing large scale transactions in hardware tends to introduce large degrees of complexity into the design. Because of this, unbounded and dynamic HTMs, like the one proposed by Ananian et al. in [20], are unlikely to be adopted by mainstream commercial processors in the near future.

STMs have fewer limitations, when compared to HTMs. Since STMs make no hardware support assumptions, they can be implemented on commodity hardware, a factor that increases their usability. Further, software is more flexible and easier to evolve than hardware. However, one of the most troublesome drawbacks of STMs is performance. Although STM performance has improved over the years, it is still significantly slower than traditional lock-based and HTM solutions. The design of STM systems is subject to many choices, and each one carries advantages and shortcomings [34, 33, 35, 36, 27].

Hybrid TM aim at combining the best of both approaches. The main idea is to use hardware support to boost performance and fallback to software when facing hardware limitations (or when the hardware support is simply not available). As shown by Dice et al. in [33], hardware support for read-set validation, as opposed to full blown HTM, may deliver significant performance benefits.

This work focus on STM.

3.2.3 Implementation

Metadata structures are necessary in a software transactional memory system in order to manage the state of the ongoing transactions. For example, conflict detection is done by executing software routines. To track the relation between a transaction and a shared object, the system can either record the objects read or updated by a certain transaction (i.e., track the transaction's read and write-set) or record the transactions that have read or updated a certain object in a *reader set* and a *writer set*, respectively.

Some software transactional memory systems perform *invisible reads*, i.e., transactional reads of shared objects are hidden from concurrent transactions. Thus, such systems cannot detect read-write conflicts. In order to handle possible inconsistencies, three approaches exist: (i) *validation*, i.e., the transaction

validates that no other transaction has modified any of the objects in its read-set, (ii) *invalidation*, i.e., track which transactions read an object and abort them when a transaction performs an update operation, and (iii) *tolerate inconsistency*, i.e., allow the transactions to execute with an inconsistent state, which in some situations can be tolerable but violates the opacity property.

One issue when implementing concurrency control in software transactional memory systems is how to handle *synchronization* and *forward progress*. There are two basic approaches: *blocking* and *non-blocking* [37]. Programs written using blocking synchronization cannot guarantee the forward progress of the system, since they can incur in deadlocks and priority inversion. A software transactional memory implementation using non-blocking synchronization can support three levels of forward progress guarantee: (i) *wait-freedom*, (ii) *lock-freedom*, and (iii) *obstruction-freedom* [37]. Wait-freedom is the strongest of the three and guarantees that *all* threads that contend for a set of shared objects make forward progress in a finite amount of time, i.e., forward progress is guaranteed. Lock-freedom only guarantees that at least one thread of those contending for a set of shared objects makes forward progress in a finite amount of time. Finally, obstruction-freedom, which is the weakest of the three, only guarantees that if only one thread is contending for a set of shared values, that thread makes forward progress in a finite amount of time. Unlike wait and lock-freedom, obstruction-freedom does not guarantee starvation-freedom.

In order to resolve conflicts between concurrent transactions we often need to abort one of the conflicting transactions. A *contention manager* typically implements one or several *contention policies* in order to decide which transaction(s) to abort. An example of a contention policy can be to always abort the "newest" transaction(s).

3.2.4 Proposals

STM The Software Transactional Memory (STM) transactional engine was proposed by Shavit and Touitou in [25]. It is the first implementation of a software transactional memory system. At the start of a transaction, it identifies and tries to obtain the ownership of all the memory words used in the transaction. If this process fails, the transaction aborts and releases the ownership of all the memory locations it already has acquired. By acquiring the ownership of memory words in an increasing order, deadlocks are avoided. Ownership information is stored in a separate metadata structure besides the actual data.

It detects conflicts at the word level, performs early conflict detection, and uses a direct update strategy since it can complete the transaction when it has acquired the ownership of all memory locations in its write-set. It only supports static transactions and uses non-blocking synchronization that ensures lock-freedom.

JVSTM The JVSTM [27] transactional engine is a pure Java software transactional memory library.

It introduces the concept of versioned boxes, which are containers that keep the history of values of an object, each of these corresponding to a change made by a committed transaction. The usage of versioned boxes ensures that read-only transactions always have access to a consistent snapshot. Therefore, they are abort and wait-free. This favors applications with high read/write transaction ratio, which is the more common case.

It detects conflicts at the object level, performs lazy conflict detection, uses a deferred update strategy, supports dynamic and nested transactions (nested transactions follow the *linear nesting* [38] model), provides strong atomicity and employs non-blocking synchronization that ensure wait-freedom for read-only transactions, and obstruction-freedom for update transactions.

3.3 Distributed Software Transactional Memory Systems

STMs were initially developed for cache-coherent shared memory architectures. DSTMs only recently began to raise attention, mainly due to scalability and fault-tolerance concerns, but also because distributed memory and non-cache-coherent architectures are receiving more attention. As a remark, Intel dumped cache-coherency on their latest many-core research prototype processor, the "Single-chip Cloud Computer".

Systems for distributed shared memory architectures typically communicate through a network, using a message-passing interface. The most popular computer networking technology – Ethernet – features much lower bandwidth and higher latency (due to the distance and the propagation speed of the medium) than the bus of a computer, and does not provide any type of ordering guarantee. Because of these differences, STM systems initially designed for shared memory architectures have to be adapted to work on distributed memory architectures.

The main motivations behind distributing a software transactional memory are scalability and reliability. Regarding scalability, it is much cheaper to build a commodity cluster than to buy a supercomputer. However, it is also typically more difficult to scale systems horizontally than vertically, due to the inter-node communication cost. Regarding reliability, high-availability requisites are very common in real world applications. However, highly-availability must be achieved with the minimum possible cost. Replication protocols should be both effective and efficient. The concepts and properties of STM systems, described in Section 3.2.1, are applicable to DSTM systems.

In the next paragraphs, we describe the following DSTM systems: DMV [39], Cluster-STM [40], DiSTM [41], D^{2}STM [42] and AGGRO [43].

3.3.1 Implementation

A DSTM system requires the use of a *distributed concurrency control* algorithm in order to guarantee the coherence of the transactional memory. A distributed concurrency control algorithm *synchronizes* nodes at some point. There are two basic approaches: *eager* and *lazy* synchronization. Eager synchronization

ensures that the transaction can commit. This can be be done with the acquisition of a cluster-wide unique token/lease. Lazy synchronization propagates the updates made by a transaction upon its commit. This commit is optimistic, because, depending of the global serialization order, the modifications made by a concurrent transaction that executed at a different node may have to be applied before. Where eager distributed concurrency control is pessimistic and favors the execution of less useless operations at the cost of higher data contention and diminished concurrency, lazy conflict detection is optimistic and favors concurrency at the cost of performing more useless computation (plus longer undo operations).

3.3.2 STM Replication

Challenges Software transactional memories and databases share the key abstraction concept of atomic transaction. However, memory and database transactions have very different characteristics. Memory transactions are typically several orders of magnitude shorter than their database counterparts [3], as DBMSs have to deal with SQL parsing and data loading from high latency secondary storage such as hard disk drives.

Since memory transactions tend to be small small, their replication cost is amplified when compared to database transactions. For example, plugging OAB is a successful technique in database replication that has limited success in STM replication [44] because, since memory transactions are so small, their execution overlaps with only a very small fraction of the time that the group communication service takes to establish the global serialization order.

In order to overcome these challenges, replication protocols for software transactional memories should develop techniques to reduce the time taken by the commit phase or to overlap more useful computation with the global commit phase.

Taxonomy Replicated software transactional memories that communicate through AB can be classified by a set of protocols.

When a node receives a request for processing a transaction it can either broadcast the request or it can process the request and then broadcast to all nodes the transaction's data-set upon the transaction's commit request. In the former case, the approach is an active replication scheme, like described in Section 3.1.1. If the latter approach is taken, it is commonly referred to as certification-based replication, in which a transaction is globally validated after its commit is requested. The validation is done based on the transaction's read and write-set. Certification-based replication can be further classified into voting and non-voting schemes. Voting schemes need to broadcast only the transaction's write-set but incur into an additional broadcast along the critical path of the commit phase, whereas non-voting schemes need to broadcast the transaction's read-set and write-set only once, but since the read-set is typically large, the message to broadcast is also larger.

3.3.3 Distributed Multiversioning

Distributed Multiversioning (DMV) [39] is a page-level distributed concurrency control algorithm that exploits the presence and easy maintenance of different versions of the transactional data-set across the nodes. Like local multiversioning schemes, DMV allows read-only transactions to execute in parallel with conflicting update transactions. This is done by ensuring that read-only transactions can always access to a consistent snapshot, i.e., a snapshot that represents the "newest" committed version of the transactional memory before the read-only transaction. However, in DMV each node maintains a single copy of each transactional item. Therefore, the system delays applying (local or remote) updates to the transactional items in order to maximize the probability of not having to invalidate the snapshot of any active transaction, and thus forcing them to abort. Updates are only applied when it is strictly necessary.

DMV is proposed on two cluster configurations: (i) a completely decentralized, update-anywhere configuration, with no scheduler support and (ii) a centralized, master-update configuration, with scheduler support. In the following paragraphs, we briefly describe how data consistency and conflict resolution are handled in both approaches.

Update-Anywhere The update-anywhere replication protocol works on totally decentralized configurations, on which read-only transactions are executed locally and update transactions perform modifications that are applied cluster-wide.

In order to maintain data consistency, at commit time, an update transaction (i) creates a new cluster-wide version for the transactional memory, (ii) it broadcasts to all other nodes the modifications made in the form of a *diff*, which is tagged with a unique system-wide (monotonically increasing) identifier that represents the version of the newly created transactional memory version (this process is called *diff flush*) and finally, (iii) it waits for the acknowledgments from all other nodes before committing the transaction locally. In order to enforce a consistent serialization order of update transactions, each update transaction obtains a unique system-wide token during commit, so that only one transaction can perform a *diff flush* at any given time. All nodes receiving a *diff flush* store it, immediately acknowledge its reception in order to minimize the delay of the committing transaction, but delay the application of the received modifications. These modifications are applied later, on-demand (lazily), upon access to a stalled page. A page is considered stalled and needs to be updated if its version number is lower than the locally maintained version number of the last commit seen by the node.

Read-only transactions iteratively create a snapshot of the data they read that reflects the state of the transactional memory at their beginning. This way, no read-only transactions are affected by incoming *diff flushes* during their execution, as modifications are not applied until all ongoing conflicting transactions are successfully committed.

Conflicts between two remote update transactions are detected when an incoming *diff flush* for a page accessed locally is received while an update transaction is executing at the local node. In this case, if any of the pages included in *diff flush* were either written to or read by the local node, the local update transaction is aborted and restarted. All transactions begin marked as being read-only and are reclassified during execution, upon trying to modify a page. The reclassification of a transaction implies a validation phase, in which the runtime system checks if the transaction being validated has already ignored a *diff flush* when reading a page, by not applying all the stored *diffs* to that page. If this occurred, the transaction is restarted as an update transaction. Otherwise, the transaction is reclassified and can safely continue its execution as an update transaction.

Master-Update The master-update replication protocol supports conflict wait avoidance. This scheme is composed by three components: a scheduler, a master node and a set of slave nodes. The scheduler is aware of the type of transactions and the versions that they are supposed to read. Based on this knowledge, it schedules the execution of update transactions only on the master node and distributes conflicting read-only transactions across the set of slave replicas.

As only the master node processes update transactions, it decides the global serialization order. At commit time, an update transaction (i) creates a new cluster-wide version for the transactional memory by generating a unique system-wide (monotonically increasing) identifier for it, (ii) the master node broadcasts to all slave nodes the modifications made in the form of a *diff*, which are tagged with a previously generated unique identifier (i.e., it performs a *diff flush*) and finally, (iii) it waits for the acknowledgments from all slave nodes before committing the transaction locally. As in the update-anywhere protocol, slaves acknowledge incoming *diff flushes* immediately after receiving them and delay the application of the received modifications, thus not delaying any longer the committing master node in order to favor scalability. When committing the update transaction, the master node communicates the unique identifier of the transaction to the scheduler. The scheduler tags each read-only transaction with the "newest" identifier that it knows and schedules it for execution on a slave. As in the update-anywhere protocol, the slave that processes the read-only transaction applies stored *diffs* when it recognizes stalled pages, based on the transaction's identifier.

Since the master node is responsible for processing all update transactions, there are no conflicts between distributed update transactions (as there are no distributed update transactions). However, read-only transactions that execute at the same node in parallel, have non-disjoint read-sets and need different versions of the same data items, can incur in local conflicts, because one transaction may update a page to a version that is "newer" than the version needed by other executing transactions.

Discussion Distributed multiversioning presents a good cost-benefit relation. Unlike classic local multiversioning, DMV does not impose the overhead of main-

taining multiple copies of the same item. However, local multiversioning is capable of providing wait and conflict-free read-only transactions, while in DMV the success of a read-only transaction is dependent on the timing of the concurrent accesses to data by conflicting transactions.

The pessimistic approach used in the update-anywhere protocol for the serialization enforcement (i.e., distributed mutual exclusion) can seriously hamper the system performance, considering that only one node commits at any time, and that consequently, all other nodes can be blocked waiting for the token and, thus, are not performing useful computation.

In the master-update scheme, the master node can become the performance bottleneck, since it is responsible for the execution of all update transactions.

3.3.4 Cluster-STM

Cluster-STM [40] is a DSTM designed for high performance on large-scale non-cache-coherent distributed systems such as commodity clusters. It addresses the problem of performing an efficient distribution of the transactional memory across the nodes. This is done by assigning a home node for each data item (a scalar or an array element) which maintains an authoritative version of the item and its metadata. Home nodes are also responsible for synchronizing the access of conflicting remote transactions to the items they shelter.

Cluster-STM does not allow dynamic creation of execution contexts. The system is bootstrapped with a fixed number of tasks, which cannot be greater than the number of processors available, and no other tasks are created or destroyed during the life of the program. Cluster-STM does not provide any local concurrency control scheme and, therefore, each task can only execute one transaction at a time. However, tasks can request the execution of transactional code on other processor in order to explore data locality, thus avoiding slow data transfers.

Processors are treated as a flat set, i.e., there is no distinction between processors within a node and processors across nodes. Cluster-STM does not feature any replication scheme nor does it provide a coherent cache of transactional remote items. Programmers in need of such features have to implement their own schemes at the application-level.

Discussion By treating processes as a flat set, Cluster-STM discards the opportunity to take advantage of shared memory among processes within the same node and thus avoid the overhead imposed by message passing communication. This problem could have been minimized by exploring local concurrency, which is unfortunately not supported. The non-exploitation of multiversioning can lead to a significant amount of aborts of read-only transactions. Leaving replication and caching mechanisms for programmers is heavy burden for those that develop applications which demand any of these features.

3.3.5 DiSTM

DiSTM [41] is a DSTM built of two core components: a transactional execution engine, an extended version of the DSTM2 [45] and a remote communication system that is based on the ProActive framework [46]. The implementation relies on a central master node, which is responsible for coordinating the execution of the cluster.

Being based on DSTM2, DiSTM executes all transactions optimistically. When a transaction updates an object, instead of directly modifying the actual object, a cloned version of the object is used, which is kept private until the transaction commits.

The commit phase is preceded by a validation phase where conflicts are detected and resolved. The validation phase also has to guarantee the transactional coherence of the cluster. If a transaction passes the validation phase, it can commit safely, making its changes public. Each node of the cluster maintains a cached version of the transactional data-set that is used by its local transactions. DiSTM is responsible for keeping each cached data-sets coherent.

The key concept of the ProActive framework is the notion of active object. Each active object has its own execution context and can be distributed over the network, supporting both mobility and remote method invocation. Nodes communicate among themselves through calls to active objects.

In the next paragraphs we briefly describe three distributed transactional memory coherence protocols: a decentralized one called Transactional Coherence and Consistency (TCC) [21] and two centralized, based on the concept of leases.

Transactional Coherence and Consistency TCC works in decentralized configurations, where each node proposes transactions to the cluster.

When a transaction wishes to commit, (i) it broadcasts to all other nodes its read-set and write-set which goes tagged with a previously acquired unique system-wide (monotonically increasing) identifier from the master node, that represents the order of the transaction within the global serialization order (this phase is called pre-commit), (ii) it waits for the result of the remote validation that occurs in all other nodes and finally, (iii) based on the result of the remote validation phase, it either commits or aborts and is rescheduled. Upon remote validation, the transaction's read and write-sets are compared against the read and write-sets of the transactions executed on a remote node (local transactions), resulting in one of three possible scenarios:

1. **There is no conflict:** No transaction is aborted. The remote node reports to the committing node that, from its side, the transaction can commit safely.
2. **There is a conflict with a "younger" local transaction:** As validation is done in a serial fashion against all local transactions in execution, the identifier of the "younger" conflicting transaction, i.e., one that has a greater global serialization number, is saved in a list and the validation phase continues. If later the validation phase of the remote transaction detects a conflict with an "older" local transaction then, (i) its validation immediately fails,

(ii) the failure is reported to the committing node and finally, (iii) the conflict list is discarded and all the "younger" transactions in it can proceed safely. Otherwise, if the remote transaction does not conflict with any "older" local transaction then, (i) its validation succeeds, (ii) the success is reported to the committing node and finally, (iii) all the "younger" transactions present in the conflict list are aborted and rescheduled. This scheme avoids unnecessary aborts of younger transactions.

3. **There is a conflict with an "older" local transaction:** (i) The remote transaction's validation immediately fails, (ii) the conflict list is discarded and the committing node is informed of the failure.

A transaction that has passed the validation phase successfully must make its changes visible cluster-wide. TM coherence is assured by a master-centric, eager approach. After making its changes visible locally, the transaction updates the global data-set that is kept at the master node. In turn, the master eagerly updates all the changed data-sets on the rest of the nodes of the cluster. Upon updating the cached data-sets, a new validation phase occurs, which discovers and aborts the transactions that have not read the most up-to-date values from the cached data-set.

Serialization Lease The role of the lease is to serialize the transactions' commits in the cluster and, therefore, to avoid the expensive broadcast of transactions' read and write-sets for validation purposes. Each transaction that passes the local validation phase requests the unique cluster-wide lease from the master node and blocks until getting it. If no other transaction possesses the lease, the master node gives it to the requesting transaction, otherwise it places the request in a queue. When the transaction of the lease owner commits, it (i) updates the global data-set kept at the master node, which eagerly updates all the changed data-sets on the rest of the nodes of the cluster, and (ii) releases the lease. Upon updating the cached data-sets, a validation phase occurs, which discovers and aborts the transactions that have not read the most up-to-date values from the cached data-set. After the lease is effectively released, the master node gives it to the next, non-aborted transaction that is waiting in the queue.

Multiple Leases Unlike the previous scheme, in this multiple leases are assigned for transactions that attempt to commit, so multiple transactions may commit in parallel, but only if they do not conflict with each-other. Every transaction that passes the local validation phase requests a lease from the master node, which will validate the committing transaction locally (i.e., at the master node), against all other transactions that own a lease at the given time. If a conflict is discovered, the transaction aborts and restarts, otherwise a lease is conceded to the transaction. When a transaction that owns a lease commits, it (i) updates the global data-set kept at the master node, which eagerly updates all the changed data-sets on the rest of the nodes of the cluster, and (ii) releases the lease. Upon updating the cached data-sets, a validation phase occurs, which discovers and aborts the transactions that have not read the most up-to-date values from the cached data-set.

Discussion As DiSTM does not exploit multiversioning, the abort rate for read-only transactions, which typically dominate realistic workloads, can be significant. The lack of fault-tolerance guarantees represents additional complexity for programmers that develop applications which demand them. All three distributed TM coherence protocols have scalability issues. The TCC protocol imposes two broadcasts along the critical path of the commit phase, which translates in long commit phases. The serialization lease protocol minimizes the number of broadcasts needed in the TCC protocol, but forces transactions to acquire a unique cluster-wide lease from the master node, making it the bottleneck for acquiring and releasing the lease. Furthermore, committing transactions have to block until they acquire the lease and the master node may attempt to assign it to a transaction that has aborted due to a conflict with the last committing transaction. The multiple leases protocol effectively reduces the time that a transaction takes to acquire a lease, but since every transaction has to be validated by the master node, it again becomes the bottleneck. Moreover, in the presence of conflict intensive workloads, the benefits of the multiple leases scheme diminishes, since fewer transactions are allowed to commit in parallel.

3.3.6 D^{2}STM

Distributed Dependable Software Transactional Memory (D^{2}STM) [42] addresses the problems of how to boost performance and enhance fault-tolerance on DSTMs. It is built on top of the JVSTM [27] transactional engine. D^{2}STM includes a replication manager that relies on the properties of the AB primitive for group communication. All nodes in the cluster maintain a full copy of the transactional memory.

D^{2}STM provides a conventional STM interface that transparently ensures non-blocking and strong consistency guarantees (i.e., one-copy serializability) even in the presence of failures. It provides all the properties of the JVSTM (described in Section 3.2.4).

The replica synchronization scheme employed, called Bloom Filter Certification (BFC), can be classified as a non-voting certification scheme that exploits a bloom filter based encoding [47] of the transactions' read-set. In the following paragraphs we briefly describe how BFC works.

Bloom Filter Certification A bloom filter is a probabilistic data structure with a tunable size that is used to test whether an element is member of a set. Queries to the bloom filter can result in false positives (i.e., a query for a certain item may indicate that it is member of the set when it is not), but never in false negatives. The more elements that are added to a set, the larger the probability of false positives. Conversely, the greater the size of the bloom filter, the lower the probability of false positives occur for the same number of elements.

A committing transaction with a non-empty write-set is first locally validated by the JVSTM. If it passes the local validation phase it is submitted for distributed certification. The replication manager begins by encoding the transaction's read-set in a bloom filter. After that, it AB-casts the transaction's

write-set (which is not encoded in a bloom filter) and read-set (encoded as a bloom filter) to all other nodes in the cluster. Upon the AB-delivery, the transaction is validated at all nodes. This cluster-wide validation phase consists in checking if the items in the write-sets of all the transactions that were committed after the committing transaction has started, are in the committing transaction's read-set. If no match is found, the transaction can commit safely. Otherwise the committing transaction is aborted and restarted.

Given that the validation phase requires the availability of the write-sets of all concurrent transactions previously committed that may invalidate any of the active transactions, the replication manager maintains a set containing those transactions. The set of transactions required is known because each node broadcasts (as a piggyback to the AB-cast transaction validation message) the identifier of the lowest version of the transactional memory that is still being used by a locally active update transaction.

Discussion By providing full replication of the transactional set, D^{2}STM presents itself as a suitable platform for the development of systems with strong consistency and high availability requisites.

Bloom filter based encoding effectively help to reduce the amount of information that is broadcast in the cluster. This is a determinant factor for the performance of the AB primitive and leads to a significant reduction of the overhead associated with the transaction certification phase. Providing abort and wait-free read-only transactions is a very important feature, since most realistic transaction-processing workloads are read-dominated. However, D^{2}STM does not overlap communication and processing phases, as it does not exploit early indications provided by an OAB primitive. Another weakness is that the write-sets are applied in a serial fashion, which can result in unused computational resources.

The architectural design of D^{2}STM is clean and simple. Its major bottlenecks are the AB-cast service and the validation phase, as they both represent a significant overhead.

3.3.7 AGGRO

AGGRO [43] is an active replication protocol that aims at maximizing the overlap between communication and processing through an AGGRessively Optimistic concurrency control scheme.

The key idea underlying AGGRO is to propagate modifications across uncommitted transactions, according to a serialization order that is compliant with the optimistic message delivery order provided by an OAB primitive, through which replicas exchange information. While most concurrency control schemes that make use of optimistic deliveries block conflicting transactions until their final order is known, AGGRO propagates the updates of yet uncommitted (but complete) transactions to the succeeding transactions. This optimistic scheme allows the processing of any transaction while its final order is yet unknown and is being defined in background.

AGGRO supports dynamic transactions, performs early conflict detection and uses a deferred update strategy. It assumes that the transactions are *snapshot deterministic* [48], i.e., when a given transaction is re-executed (due to a previous abort) over a given version of the transactional memory, and therefore always reads the same values, then it behaves deterministically by always executing the same set of read and write operations.

The Protocol A transaction can be in one of four possible states: *active* (i.e., executing), *complete* (i.e., committing), *committed* or *aborted*. The definition of these states made in Section 3.1.3 holds for the AGGRO protocol.

Each transactional item X maintained by the transaction manager is associated with a set of versions $\{X^1, ..., X^n\}$. Only a single version of a transactional item is committed at any time. However, a version that has been written by an uncommitted transaction can be in one of two possible states: *work-in-progress* (WIP) or *complete*.

- **WIP:** the version was created by a currently active transaction. Its value is only visible for the transaction that has created it.
- **Complete:** the version was created by a transaction that is in the complete state. Its value is available for the transactions that succeed the transaction that wrote it.

When the transaction manager receives a request from the overlying application requesting the execution of a transaction, it OAB-casts the transaction itself and blocks until the message containing the transaction is OAB-delivered. Upon the OAB-delivery, the transaction is activated.

All read and write operations to transactional items are trapped by the transaction manager, in order to perform a controlled propagation of complete versions across transactions and also do conflict detection.

When the transaction is AB-delivered, the transactional manager waits until the transaction completes. After that, it checks if there exist any conflicts due to a mismatch between the OAB-delivery order and the AB-delivery order. In case the AB-delivered transaction has read a value written by a transaction that does not precede it, the AB-delivered transaction has to abort. Aborting a transaction clears all its uncommitted versions and spawns a new thread to re-execute it.

Discussion Being a pure active replication protocol, AGGRO leads to a great amount of redundant computation. By sticking to the OAB-delivery order for the optimistic serialization order, AGGRO may not perform as well with conflict-intensive workloads in a network environment with a very high percentage of out-of-order messages, since it will not be able to effectively overlap computation with communication. Moreover, it will impose a greater computational overhead due to the execution of cascading rollbacks.

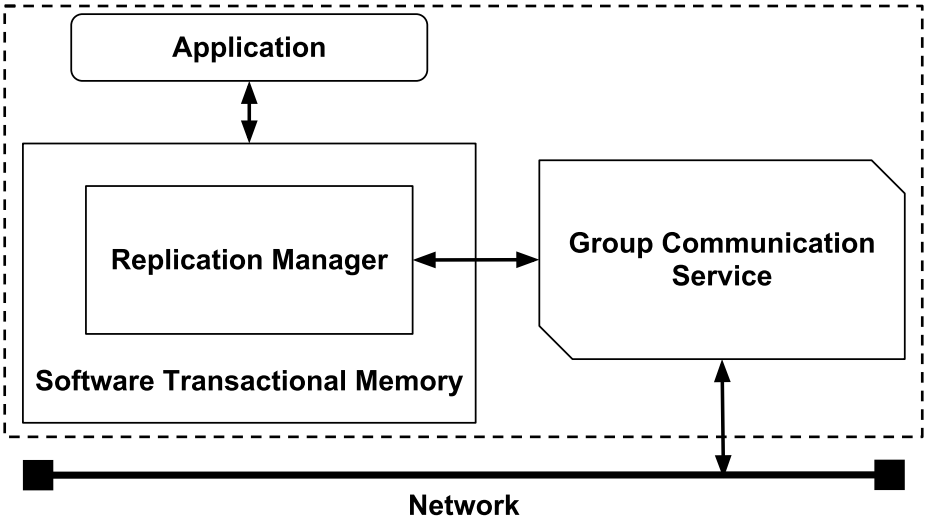

Fig. 1. Components of the typical architecture of a DSTM-based application

4 Architecture

4.1 Overview

Figure 1 illustrates the typical architecture that we want to target with our speculative replication protocol. Each application-server of the cluster is composed by the following components:

- **Application:** represents the logic implemented by the programmer. It requests the execution of transactional code to the software transactional memory.
- **Software Transactional Memory:** responsible for the processing of the incoming requests from the application-logic.
- **Replication Manager:** responsible for replicating and maintaining the transactional data-set consistent across the cluster.
- **Group Communication Service:** responsible for the group communication support and for ensuring the atomic broadcast properties.

Although this is a generic architecture, we will consider the concrete case where we are targeting the Fénix framework [4]. Therefore, the software transactional memory is the JVSTM [27], the STM replication manager is under development at the Distributed Systems Group and the Software Engineering Group of INESC-ID, and the group communication service is the Appia framework [6].

A transaction is started whenever the application requests it to the STM. When the application requests the commit of a transaction, the STM replication manager has to do it globally. The replication manager is responsible for ensuring

the consistency of the transactional memory among all replicas, which is achieved through a certification protocol. The transaction is first submitted to a local validation process and if it succeeds, the transaction's write-set or data-set is broadcast, depending on whether the certification protocol is a voting or a non-voting scheme, respectively. The transaction is then validated at all replicas, in a deterministic process that outputs the decision of either to abort or commit the transaction. Although both voting and non-voting schemes can be used, in this report we assume the use of a non-voting scheme.

We want to provide the ability to perform the global commit of a transaction in background, allowing the application to continue its execution. For this, we propose the use of a speculative replication protocol that ensures the consistency of the transactional memory in the presence of speculative modifications, i.e., although the transactions may execute over an inconsistent state, the transaction manager ensures that invalid changes are never seen outside the system. The implementation of this strategy faces at least three big challenges, as we discuss next.

4.2 Challenges

Firstly, this scheme is only effective in presence of sequences of transactions, i.e., a execution flow that is composed by a series of transactions, which does not represent the majority of the code written by programmers for applications. Typically, an STM-based application is composed by transactional code interleaved with non-transactional code. To tackle this problem we will transactify non-transactional code in an automatic fashion. This can be done at either the source code level, or the byte-code level. A possible way is to adapt the work of Anjo in [49] to our needs, i.e., we only want to transactify portions of non-transactional code that reside in the middle of already existent transactions and not the whole application, nor we need automatic parallelization. Although transactional code executes slower than its non-transactional counterpart, we expect that, in the long run, the cost will be compensated.

A transaction that results from the automatic transactification process may, or may not update the replicated transactional state. If it does not, then there is nothing to commit globally, so the transaction is strictly local. However, if it does, it must be committed globally. Therefore, the replication manager has to be able to distinguish between items of the replicated transactional data-set and items from the local transactional data-set.

The second challenge is the need to propagate the modifications made by yet uncommitted transactions to its succeeding transactions. This will be done in a process similar to what is employed in AGGRO (described in Section 3.3.7).

We assume that transactions are snapshot deterministic (described in Section 3.3.7) since otherwise, i.e., if the writes made by a transaction do not uniquely depend on the values that it reads, then we cannot consider legal the writes made by transactions that, although were successfully validated globally, may have been executed over an inconsistent state. This happens because we cannot

guarantee that a transaction will produce the same result if it is re-executed over a state that is guaranteed to be consistent.

Transactions can be in one of four possible states: *active* (i.e., executing), *complete* (i.e., committing), *committed* or *aborted*. The definition of these states presented in Section 3.1.3 holds for our protocol. Uncommitted versions of transactional data items can be in one of two possible states: *work-in-progress* (WIP) or *complete*. The definition of these states is presented in Section 3.3.7.

When a transaction starts at its local node, it is placed in a set of transactions in the active state. Immediately upon an active transaction modifies an item, it creates/modifies a version for that item. This version is marked as WIP and can only be accessed by the transaction that created it. When a transaction reads from an item and accesses a version in the complete state, the read operation establishes a dependency between the reading transaction and the transaction that wrote the uncommitted value. After the application requests that the transaction be committed, it is first locally validated. If the local validation fails, the transaction has to be aborted. Otherwise, if the local validation succeeds, the transaction is removed from the set of active transactions, is placed in a set for transactions in the complete state and all the versions in its write-set change from WIP to the complete state. Subsequently, the certification phase begins, leading to the broadcast of the transaction's data for global validation.

If the global validation succeeds, the transaction may commit at all nodes. Beyond applying the transaction's updates, the node that executed the transaction has to perform further actions. It removes the transaction from the set of completed transactions, all the items in the transaction's write-set change from the complete state to the Committed state, and all speculative dependencies among the committed transaction and other transactions are removed. After that, the replication manager tries to globally commit the next completed transaction that has no speculative read dependency.

If the global validation fails, the transaction has to be aborted. This leads to a cascading rollback process, because the transaction manager has to abort all the transactions that have read from versions created by the aborted transaction, in order to ensure the consistency of the transactional memory.

The third challenge is related with the second. A transaction in the complete state may have been executed over an inconsistent state, since it may have read from versions created by other transactions in the complete state. Therefore, without further work, its global commit cannot be requested until all the versions in its read-set are in the committed state. Besides this, the commit order has to respect the serial execution order. A solution for this problem may include an immediate commit of each transaction after its execution completes, and plugging additional information to its validation message that tells from which transactions it depends. We will be looking for more ways to overcome this problem.

5 Evaluation

The performance of the proposed replication protocol will be assessed with well established benchmark suites for transactional systems, such as:

- **TPC-W**: [50] is a transactional web benchmark. The workload is performed in a controlled internet commerce environment that simulates the activities of a business oriented transactional web server. The workload exercises a breadth of system components associated with multiple environments.
- **Lee-TM:** [51] is a non-trivial benchmark suite for TM systems based on the well known Lee's routing algorithm used in circuit routing. Lee's routing algorithm has many of the desirable properties of a non-trivial TM benchmark such as large amounts of parallelism, complex contention characteristics, and a wide range of transaction durations and lengths.
- **STMBench7:** [52] consists of a set of graphs and indexes intended to be suggestive of many complex applications, e.g., CAD/CAM. A collection of operations is supported to model a wide range of workloads and concurrency patterns.

These benchmarks will have to be adapted for the JVSTM programming model. We will use an already developed implementation of a protocol with similar characteristics (i.e., certification-based, non-voting) that does not perform any speculative execution. Its results will serve as base to assess the efficiency of our speculative replication protocol.

The performance metrics that will be analyzed are:

- Throughput (transactions per second)
- Response time (microseconds)
- Transactions' abort rate (percentage)

These metrics will allow us to evaluate the protocol scalability and its behavior under various workloads with distinct characteristics.

6 Scheduling of Future Work

Future work is scheduled as follows:

- January 7 - March 29, 2011: Detailed design and implementation of the proposed architecture, including preliminary tests.
- March 30 - May 3, 2011: Perform the complete experimental evaluation of the results.
- May 4 - May 23, 2011: Write a paper describing the project.
- May 24 - June 14, 2011: Finish the writing of the dissertation.
- June 14, 2011: Deliver the MSc dissertation.

7 Conclusions

In this report we surveyed a number of subjects related to software transactional memory replication. We started with the description of fundamental knowledge of replication, such as models, group communication and database replication. We proceeded to STM concepts, properties, and briefly described two proposals. The survey ends with the description of challenges for efficient STM replication and DSTM proposals.

The survey of the related work and the resulting analysis motivated the proposal of the architecture here presented, for a novel replication protocol for transactional systems that employs a speculative scheme, in order to boost the system performance. We identified some challenges in the implementation of the proposed architecture.

The report concludes with a description of the evaluation methodology to be applied to our solution and a scheduling for the future work.

Acknowledgments

I am grateful to my advisor, professor Luís Rodrigues, to all the Distributed Systems Group team, namely to Nuno Carvalho, Paolo Romano, Maria Couceiro, Pedro Ruivo, Xavier Vilaça, Nuno Machado and João Paiva, and also to the Software Engineering Group team, namely to Ivo Anjo, for the fruitful discussions, comments and support during the preparation of this report. This work was partially supported by project "ARISTOS", under the FCT grant (PTDC/EIA-EIA/102496/2008).

References

1. Moore, G.E.: Cramming More Components Onto Integrated Circuits. Proceedings of the IEEE **86**(1) (1998) 82–85
2. Cachopo, J.: Development of Rich Domain Models with Atomic Actions. PhD thesis, Technical University of Lisbon (2007)
3. Romano, P., Carvalho, N., Rodrigues, L.: Towards distributed software transactional memory systems. In: Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008), Watson Research Labs, Yorktown Heights (NY), USA (September 2008) (invited paper).
4. IST: The Fénix Framework website. https://fenix-ashes.ist.utl.pt/trac/fenix-framework
5. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms: Taxonomy and survey. ACM Comput. Surv. **36** (December 2004) 372–421
6. Miranda, H., Pinto, A., Rodrigues, L.: Appia: A flexible protocol kernel supporting multiple coordinated channels. Distributed Computing Systems, International Conference on **0** (2001) 707–710
7. Vitenberg, R., Keidar, I., Chockler, G.V., Dolev, D.: Group communication specifications: A comprehensive study. ACM Computing Surveys **33** (1999) 2001
8. Powell, D.: Group communication. Commun. ACM **39** (April 1996) 50–53

9. Birman, K.P., Joseph, T.A.: Reliable communication in the presence of failures. ACM Trans. Comput. Syst. **5** (January 1987) 47–76
10. Lamport, L.: Ti clocks, and the ordering of events in a distributed system. Commun. ACM **21** (July 1978) 558–565
11. Kemme, B., Pedone, F., Alonso, G., Schiper, A., Wiesmann, M.: Using optimistic atomic broadcast in transaction processing systems. Knowledge and Data Engineering, IEEE Transactions on **15**(4) (jul. 2003) 1018 – 1032
12. Haerder, T., Reuter, A.: Principles of transaction-oriented database recovery. ACM Comput. Surv. **15** (December 1983) 287–317
13. Pedone, F., Guerraoui, R., Schiper, A.: The database state machine approach. Distrib. Parallel Databases **14**(1) (2003) 71–98
14. Schneider, F.B., Schneider, F.B.: Implementing fault-tolerant services using the state machine approach: A tutorial. ACM Computing Surveys **22** (1990) 299–319
15. Bernstein, P.A., Hadzilacos, V., Goodman, N.: Concurrency Control and Recovery in Database Systems. Addison-Wesley (1987)
16. Harris, T., Marlow, S., Peyton-Jones, S., Herlihy, M.: Composable memory transactions. In: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP '05, New York, NY, USA, ACM (2005) 48–60
17. Moravan, M.J., Bobba, J., Moore, K.E., Yen, L., Hill, M.D., Liblit, B., Swift, M.M., Wood, D.A.: Supporting nested transactional memory in logTM. SIGOPS Oper. Syst. Rev. **40** (October 2006) 359–370
18. Martin, M., Blundell, C., Lewis, E.: Subtleties of transactional memory atomicity semantics. IEEE Comput. Archit. Lett. **5** (July 2006) 17–
19. Guerraoui, R., Kapalka, M.: On the correctness of transactional memory. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP '08, New York, NY, USA, ACM (2008) 175–184
20. Ananian, C.S., Asanovic, K., Kuszmaul, B.C., Leiserson, C.E., Lie, S.: Unbounded transactional memory. In: Proceedings of the Eleventh International Symposium on High-Performance Computer Architecture. (Feb 2005) 316–327
21. Hammond, L., Wong, V., Chen, M., Carlstrom, B.D., Davis, J.D., Hertzberg, B., Prabhu, M.K., Wijaya, H., Kozyrakis, C., Olukotun, K.: Transactional memory coherence and consistency. SIGARCH Comput. Archit. News **32** (March 2004) 102–
22. Herlihy, M., Moss, J.E.B.: Transactional memory: Architectural support for lock-free data structures. In: Proceedings of the 20th Annual International Symposium on Computer Architecture. (May 1993) 289–300
23. Tomic, S., Perfumo, C., Kulkarni, C., Armejach, A., Cristal, A., Unsal, O., Harris, T., Valero, M.: EazyHTM: Eager-lazy hardware transactional memory. In: MICRO '09: Proceedings of the 2009 42nd IEEE/ACM International Symposium on Microarchitecture. (2009)
24. Bobba, J., Goyal, N., Hill, M.D., Swift, M.M., Wood, D.A.: TokenTM: Efficient execution of large transactions with hardware transactional memory. In: Proceedings of the 35th Annual International Symposium on Computer Architecture. (Jun 2008)
25. Shavit, N., Touitou, D.: Software transactional memory. In: Proceedings of the 14th ACM Symposium on Principles of Distributed Computing. (Aug 1995) 204–213
26. Saha, B., Adl-Tabatabai, A.R., Hudson, R.L., Cao Minh, C., Hertzberg, B.: McRT-STM: a high performance software transactional memory system for a multi-core runtime. In: Proc. 11th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPoPP '06). (Mar 2006) 187–197

27. Cachopo, J., Rito-Silva, A.: Versioned boxes as the basis for memory transactions. Sci. Comput. Program. **63**(2) (2006) 172–185
28. Harris, T., Fraser, K.: Language support for lightweight transactions. In: Object-Oriented Programming, Systems, Languages, and Applications. (Oct 2003) 388–402
29. Dice, D., Shalev, O., Shavit, N.: Transactional locking II. In: DISC '06: Proc. 20th International Symposium on Distributed Computing. (Sep 2006) 194–208 Springer-Verlag Lecture Notes in Computer Science volume 4167.
30. Tabba, F., Wang, C., Goodman, J.R., Moir, M.: NZTM: Nonblocking, zero-indirection transactional memory. In: Workshop on Transactional Computing (TRANSACT). (2007)
31. Damron, P., Fedorova, A., Lev, Y., Luchangco, V., Moir, M., Nussbaum, D.: Hybrid transactional memory. In: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS). (2006) 336–346
32. Kumar, S., Chu, M., J. Hughes, C., Kundu, P., Nguyen, A.: Hybrid transactional memory. In: Proceedings of Symposium on Principles and Practice of Parallel Programming. (Mar 2006)
33. Dice, D., Shavit, N.: Understanding tradeoffs in software transactional memory. In: CGO '07: Proceedings of the International Symposium on Code Generation and Optimization. (mar 2007) 21–33
34. Ennals, R.: Software transactional memory should not be obstruction-free. Technical Report IRC-TR-06-052, Intel Research Cambridge Tech Report (Jan 2006)
35. Marathe, V.J., Iii, W.N.S., Scott, M.L.: Design tradeoffs in modern software transactional memory systems. In: In Proceedings of the 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems, ACM Press (2004) 1–7
36. Marathe, V.J., Iii, W.N.S., Scott, M.L.: Adaptive software transactional memory. In: In Proc. of the 19th Intl. Symp. on Distributed Computing. (2005) 354–368
37. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)
38. Moss, J.E.B., Hosking, A.L.: Nested transactional memory: model and architecture sketches. Sci. Comput. Program. **63** (December 2006) 186–201
39. Manassiev, K., Mihailescu, M., Amza, C.: Exploiting distributed version concurrency in a transactional memory cluster. In: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming. PPoPP '06, New York, NY, USA, ACM (2006) 198–208
40. Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. PPoPP '08, New York, NY, USA, ACM (2008) 247–258
41. Kotselidis, C., Ansari, M., Jarvis, K., Luján, M., Kirkham, C., Watson, I.: Distm: A software transactional memory framework for clusters. In: Proceedings of the 2008 37th International Conference on Parallel Processing. ICPP '08, Washington, DC, USA, IEEE Computer Society (2008) 51–58
42. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D2stm: Dependable distributed software transactional memory. Pacific Rim International Symposium on Dependable Computing, IEEE **0** (2009) 307–313
43. Palmieri, R., Quaglia, F., Romano, P.: AGGRO: Boosting STM replication via aggressively optimistic transaction processing. Network Computing and Applications, IEEE International Symposium on **0** (2010) 20–27

44. Palmieri, R., Quaglia, F., Romano, P., Carvalho, N.: Evaluating database-oriented replication schemes in software transactional memory systems. In: Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on. (april 2010) 1 –8
45. Herlihy, M., Luchangco, V., Moir, M.: A flexible framework for implementing software transactional memory. SIGPLAN Not. **41** (October 2006) 253–262
46. Baduel, L., Baude, F., Caromel, D., Contes, A., Huet, F., Morel, M., Quilici, R.: Programming, Composing, Deploying for the Grid. In Cunha, Jose C.; Rana, O.F., ed.: Grid Computing: Software Environments and Tools. Springer (2006) 205 – 229
47. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM **13** (July 1970) 422–426
48. Romano, P., Palmieri, R., Quaglia, F., Carvalho, N., Rodrigues, L.: Brief announcement: on speculative replication of transactional systems. In: Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures. SPAA '10, New York, NY, USA, ACM (2010) 69–71
49. Anjo, I.F.S.D.: JaSPEx: Speculative Parallelization on the Java Platform (november 2009)
50. García, D.F., García, J.: TPC-W E-Commerce Benchmark Evaluation. Computer **36** (February 2003) 42–48
51. Ansari, M., Kotselidis, C., Watson, I., Kirkham, C., Luján, M., Jarvis, K.: Lee-tm: A non-trivial benchmark suite for transactional memory. In: Proceedings of the 8th international conference on Algorithms and Architectures for Parallel Processing. ICA3PP '08, Berlin, Heidelberg, Springer-Verlag (2008) 196–207
52. Guerraoui, R., Kapalka, M., Vitek, J.: Stmbench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. **41** (March 2007) 315–324