SCert: Speculative Certification in Replicated Software Transactional Memories*

Nuno Carvalho
INESC-ID/IST
Rua Alves Redol, 9
Lisboa, Portugal
nonius@gsd.inesc-id.pt

Paolo Romano
INESC-ID
Rua Alves Redol, 9
Lisboa, Portugal
romano@inesc-id.pt

Luís Rodrigues
INESC-ID/IST
Rua Alves Redol, 9
Lisboa, Portugal
ler@ist.utl.pt

ABSTRACT

Being much simpler to compose and verify than classical lock based synchronization schemes, Software Transactional Memories (STMs) have emerged as an attractive paradigm for supporting concurrent access to in-memory storage systems. This paper is focused on the issue of how to replicate STMs to enhance both their performance and dependability. This is an extremely challenging problem, since the communication/processing ratio in STMs is typically several orders of magnitude higher than in conventional database systems, thus amplifying the relative cost of replication.

We present SCert (Speculative Certification), a novel replication protocol for STMs that exploits early knowledge about message ordering in the underlying atomic broadcast layer to propagate, in a speculative fashion, the updates of transactions before there is an agreement on the final serialization order. This speculative approach brings the two following key benefits. On one hand, it lowers the chances that transactions access stale snapshots, thus minimizing the probability of later incurring in an abort. On the other hand, it provides early conflict detection, thus reducing the amount of wasted computation and/or waiting time from transactions doomed to abort. An experimental study of SCert, based on a fully fledged distributed STM prototype and heterogeneous benchmarks, has shown performance gains of up to 4.5x when compared with previous certification based schemes.

Categories and Subject Descriptors

D.4.2 [**Storage Management**]: Distributed memories; D.4.7 [**Organization and Design**]: Distributed systems

General Terms

Algorithms, Performance, Reliability

1. INTRODUCTION

Building on the abstraction of atomic transactions, and freeing the programmer from the complexity of conventional lock-based synchronization schemes, Software Transactional Memory (STM) provides the support for building concurrent in-memory data stores, improving the system reliability and boosting productivity [6]. Over the last years, a wide body of literature has been developed in the area of STMs and the first enterprise-class STM-based applications have started to be deployed in production systems [18]. A fundamental challenge in this context is to design efficient schemes capable of fulfilling the scalability and reliability requirements of real-world applications.

At current date, only a handful of replication protocols explicitly targeted at STMs have been proposed [23, 4, 26, 11, 9]. On the other hand, since STMs and databases share the common abstraction of atomic transaction, the wide body of literature on replicated and distributed databases represents a natural reference point to architect STM replication protocols. Among the vast range of transactional replication schemes targeting database systems, approaches based on Atomic Broadcast (AB) [16] and distributed certification procedures [32, 21, 31] appear to be particularly suited for implementing STMs and have inspired several recent STM replication protocols [11, 9].

Unlike classic eager replication schemes (based on fine-grained distributed locking and atomic commit), which suffer from large communication overheads and distributed deadlocks [14], AB based certification schemes do not require any replica coordination during the transaction execution phase. Instead, transactions are executed locally in an optimistic fashion and consistency (typically, 1-Copy serializability [3]) is ensured at commit-time, via a distributed certification phase that uses AB to enforce agreement on a common transaction serialization order. This provides the following benefits: it avoids distributed deadlocks, which are known to significantly limit scalability of eagerly replicated transactional systems [14]; it offers non-blocking guarantees in the presence of failures; it allows for a modular implementation, where the complexity associated with failure handling is encapsulated by the AB and group management system layers (typically providing View-Synchrous communication [16]).

Further, certification schemes are more scalable and easy to deploy in practice than classical active replication schemes [22, 35, 29]. Active replication requires write transactions

*This work has been partially supported by the project "Cloud-TM" (co-financed by the European Commission through the contract no. 257784), the FCT project ARISTOS (PTDC/EIA- EIA/102496/2008) and by FCT (INESC-ID multiannual funding) through the PIDDAC Program Funds.

SYSTOR'11, May 30–June 1, 2011, Haifa, Israel.

to be executed on every replica, which limits the system scalability in presence of write intensive workloads. Also, it requires the transaction to be deterministic, which typically forces the use of complex mechanisms to filter out any source of non-determinism [31] during transaction processing. With a few exceptions [35, 29], optimizations of active replication require a priori knowledge about the data items to be accessed by transactions during their execution [16].

On the downside, certification replication schemes rely on an inherently optimistic assumption: the data snapshot provided to a transaction is unlikely to be invalidated by the stream of updates generated by concurrent transactions. In high conflict scenarios, where this assumption does not hold, certification schemes may suffer from undesirably high abort rates and be prone to trashing [9].

The replication protocol presented in this paper, named SCert (Speculative Certification), tackles precisely this issue. The key idea at the core of SCert is to reduce the time to disseminate the updates generated by committing transactions in order to achieve the following two complementary goals:

- to provide executing transactions with fresher snapshots, thus reducing the probability of abort due to reads from stale data;
- to detect conflicts earlier during transaction execution, thus reducing the amount of wasted computation and useless waiting time caused by transactions doomed to abort.

This is achieved via a speculative approach, which leverages on the service provided by an Optimistic Atomic Broadcast (OAB) layer [33]. OAB allows to propagate the post-images of committing transactions well before their final serialization order is defined by the AB service. In addition to the final, total delivery order notification, which is available only after several communication steps (typically at least three [12]), an OAB service also provides an earlier guess of the final total order. This guess, called optimistic delivery order, normally corresponds with the spontaneous network delivery order and can therefore be made available after a single communication step. Also, as discussed in [33] and confirmed by our experimental study, the probability of mismatch between optimistic and final delivery order is typically fairly low in LANs ($< 15\%$).

SCert takes advantage of this property in a twofold way. First, it propagates the updates in a speculative serialization order that corresponds to the sequence of optimistically delivered messages. Second, it allows speculatively activated transactions to further propagate the snapshots they generate across chains of speculative transactions. This provides an effective pipelining of speculative transactions that allows to maximize the gains achievable via speculation. On the other hand, speculation exposes SCert to risks of cascading abort in case of mismatches between the optimistic and final delivery order of two conflicting transactions. As we will demonstrate in our experimental study, this represents an advantageous tradeoff. In all the scenarios we tested, including those generating higher loads (and consequently mismatches between optimistic and final delivery orders) the performance penalty associated with the occurrence of mismatches between final and optimistic delivery orders is largely compensated by the benefits achievable by the aggressive propagation of speculative snapshots.

It should be highlighted that the benefits of speculatively propagating the snapshots are higher in the context of STMs than in conventional databases. In fact, unlike classical database systems, STMs incur neither in disk access latencies nor in the overheads of SQL statement parsing and plan optimization. This makes the execution time of typical STM transactions two or three orders of magnitude shorter than in database settings [34]. Since the ratio between coordination times and transaction processing times is higher in STMs, there are also more opportunities to obtain performance gains from optimistic schemes that shorten the coordination phase, such as SCert.

In order to evaluate the actual speed-ups offered by SCert we have developed a complete system prototype based on JVSTM [7] and the APPIA Group Communication System [28]. While the SCert scheme could be in principle coupled with STMs that employ different concurrency control policies, our choice to integrate SCert with a multi-versioned STM (like JVSTM), is motivated by a twofold reason. First, the multi-versioning concurrency control mechanism adopted by JVSTM allows maximizing the performance of read-only transactions, preventing them from aborting or ever blocking due to conflicts with write transactions. Further, since JVSTM already maintains and manages multiple data item versions, it lends itself naturally to be extended to support the additional, speculative data item versions exploited by SCert. Through an extensive experimental evaluation, based on both synthetic micro-benchmarks, as well as complex STM benchmarks we show that SCert achieves speed-ups of up to 4.5x when compared with competing replicated STMs [11].

The remainder of this paper is structured as follows. Section 2 discusses relationships with related work. Section 3 presents the system model and Section 4 describes the architecture. Section 5 introduces SCert and discusses the issues associated with its integration with JVSTM. Section 6 presents the results of our experimental study. Finally, Section 7 concludes the paper.

2. RELATED WORK

There are many similarities among replication of in-memory software transactional storages and database replication, given that they both implement the abstraction of atomic transactions. Modern database replication schemes [32, 31, 21] rely on Atomic Broadcast (AB) to enforce a global transaction serialization order. AB-based schemes are non-blocking and avoid the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commitment [14]. They can be classified in two main categories: optimistic (or certification based) approaches [32, 21] and conservative approaches [22].

Conservative approaches can be seen as an instance of the classical state-machine (active replication) approach [16]. They serialize transactions through AB *prior* to their actual execution, which then follows a deterministic schedule on each replica. This prevents aborts that could result from the concurrent execution of conflicting transactions in different replicas. On the other hand, they require transactions to be deterministic. This leads to several drawbacks in realistic settings. First, it demands to intercept and filter any source of non-determinism in the user level logic (such as interaction with local timers or devices), which may be costly, intrusive, and non-trivial. Second, enforcing deterministic

thread scheduling at each replica requires a careful identification of the data items to be accessed by each transaction, prior to its actual execution. This is particularly complex with STMs, since they expose a more loosely structured data model (e.g., word-based [13] or object-oriented [7] programming interfaces) than relational databases. Finally, these approaches require update transactions to be fully executed by all replicas [31], which strongly hampers their scalability in presence of write intensive workloads.

Optimistic approaches [32, 36], on the other hand, avoid all of the above problems by executing transactions at a single node without acquiring any (distributed) lock during their execution, and relying on a commit-time global certification phase to ensure consistency. Unfortunately, in these approaches local transactions are vulnerable to aborts induced by conflicting transactions that execute concurrently on remote nodes. This can lead to severe performance degradation in high conflict scenarios, where these approaches may cause high abort rates. The SCert protocol aims precisely at tackling this problem. By speculatively exposing the updates of committing transactions as soon as these are optimistically delivered, SCert maximizes the chances that new transactions access a non-obsolete data snapshot (thus sparing them from a later abort), and provides earlier conflict detection for already executing transactions (thus minimizing the amount of computational resources wasted by transactions doomed to be aborted).

The idea of exploiting the early notifications provided by the spontaneous network delivery order has been originally proposed in the context of active replication schemes [31, 22]. By activating the processing of a transaction as soon as the corresponding message is optimistically delivered, the solution in [22] aims at overlapping the transaction processing and replication phases. Unfortunately, as highlighted in [30], the effectiveness of this optimization is strongly reduced when applied to STMs, given that the transaction execution times are typically several orders of magnitude shorter than the AB latency.

The distributed STM solutions we are aware of are those described in [23, 4, 26, 11, 9, 24, 29, 35]. Here, we will not discuss the algorithms of [23, 4, 26, 24] as they do not integrate any fault-tolerance or replication features, which is the focus of this paper. We will therefore focus our discussion on analyzing the relationships between SCert and the replication mechanisms employed by [11, 9, 29, 35].

To the best of our knowledge, D^{2}STM [11] has been the first fault-tolerant replicated STM platform to be proposed. D^{2}STM adopts an optimistic certification scheme, and uses a Bloom-filter based encoding scheme to minimize the amount of information transmitted via the AB primitive, at the cost of a tunable (and typically negligible) additional abort rate. D^{2}STM's bloom filter based certification can be transparently coupled with our SCert scheme (and indeed the SCert prototype described in the following does integrate this technique). Unlike D^{2}STM, however, SCert takes a speculative approach which allows achieving up to 4.5x speed-ups in high conflict scenarios, as reported in Section 6.

The Asynchronous Lease Certification (ALC) [9] relies on AB to enforce agreement on the order of acquisition of lease requests on subsets of data in a non-blocking, deadlock-free fashion. This allows replicas that already own leases on the data accessed by their transaction to disseminate data updates using Uniform Reliable Broadcast (URB) [16], which is a group communication primitive faster than AB. Further, by holding leases when a transaction aborts, the transaction re-execution becomes sheltered from subsequent aborts as long as it accesses the same data set. The SCert protocol represents a complementary, lightweight technique, which may even be combined with ALC protocol to boost its performance. Among the STM replication mechanisms proposed up to date, the ones that are closer in spirit to SCert are those presented in [35, 29, 5], which also speculatively process transactions exploiting the optimistic delivery order notifications provided by an OAB service. The fundamental difference between SCert and these approaches is that the latter ones belong to the previously described class of conservative, active replication schemes, whereas SCert is, to the best of our knowledge, the first speculative certification replication protocol ever proposed in literature.

The large body of literature on Distributed Shared Memory (DSM) is also related to our work. Early DSM systems [25] enforced strong consistency guarantees at the granularity of a single memory access. Those systems have proved hard to implement with good performance. Due to this reason, a significant body of research was devoted to build DSM systems that aim at achieving better performance at the cost of relaxing memory consistency guarantees [20]. Unfortunately, developing software for relaxed DSM's consistency models can be challenging as programmers are required to fully understand sometimes unintuitive consistency models. Conversely, the simplicity of the atomic transaction abstraction, at the core of STM platforms, allows to increase programmers' productivity [6] with respect to both locking disciplines and relaxed memory consistency models. Further, the strong consistency guarantees provided by atomic transactions can be supported through much more efficient algorithms that, like SCert, incur only in a single synchronization phase per transaction, amortizing the communication overhead across a (possibly large) set of memory accesses.

Atomic transactions play a key role also in the Sinfonia [1] platform, where these are referred to as "mini-transactions". However, unlike in conventional (distributed) STM settings or in SCert, Sinfonia assumes transactions to be static, i.e. that their data-sets and operations are known in advance, which limits the generality of the programming paradigm provided by this platform.

3. SYSTEM MODEL

We consider a system composed of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing. We assume that a majority of processes is correct and that the remaining minority may fail according to the fail-stop (crash) model. Furthermore we assume that the system is asynchronous but augmented with an unreliable failure detector such that a primary partition view synchronous Group Communication Service (GCS) [10, 2] can be implemented. GCS integrates two complementary services: *membership* and *multicast communication*. Informally, the role of the membership service is to provide each participant in a distributed computation with information about which processes are active and which ones are failed. Such information is called a group *view*. The multicast service allows a member to send a message to the group of participants with different reliability and ordering properties.

A primary-component group membership service GCS provides a totally ordered sequence of group views to every

correct participant. Specifically, the GCS delivers to the application a *viewChange* event to notify the alteration of the (primary component) view, and an *ejected* event to notify the exclusion of the process from the primary component (typically because of a false failure suspicion). We say that a process is v_i-correct in a given view v_i if it does not fail in v_i and if v_{i+1} exists, it transits to it. We assume a GCS ensuring the following properties on the delivered views:

- **Self-inclusion:** if process p delivers view v_i, then p belongs to v_i.
- **Strong view-synchrony:** messages are delivered in the same view in which they were sent.
- **Primary component view:** the sequences of views delivered are totally ordered and for any two consecutive views v_i, v_{i+1} there always exists a v_i-correct process p that belongs to both views.
- **Non-Triviality:** when a process fails or it is partitioned from the primary view, it will be eventually excluded from the primary component view.
- **Accuracy:** a correct process that is not is partitioned from the primary view, is eventually included in every view delivered by the GCS.

The GCS provides an Optimistic Atomic Broadcast (OAB) [12] communication service, which exports three communication primitives: *OA-broadcast*(m), which is used to broadcast message m; *Opt-deliver*(m), which delivers message m without providing ordering guarantees; *TO-deliver*(m), which delivers message m in the final total order. Informally, OAB ensures total order of the *TO-deliver* events in a non-blocking fashion despite process crashes. A *TO-Deliver* event for a message m is however always preceded by a corresponding *Opt-deliver* event, which provides an early, and possibly incorrect, estimate of the final order in which m will be delivered by the OAB service. The OAB service guarantees the following properties:

- **Validity** If a v_i-correct process p *OA-broadcasts* message m in v_i, then p *Opt-delivers* and *TO-delivers* m.
- **Integrity** Any message m is *Opt-delivered* and/or *TO-delivered* by a process p at most once, and only if it had been previously *OA-broadcast*.
- **Optimistic Order** If a node p *TO-delivers* m, then node p has previously *Opt-delivered* m.
- **Uniform Agreement** If process p *TO-delivers* m in view v_i, then any v_i-correct process *TO-delivers* m in view v_i.
- **Total Order** If two processes p and q *TO-deliver* messages m and m', then they do so in the same order.

4. SCERT ARCHITECTURE

The architecture of the software deployed on each replica is illustrated in Figure 1. The top layer is a wrapper that intercepts the application level calls for transaction demarcation (i.e. to begin, commit or abort transactions), not interfering with the application (read/write) access to the transactional data items, which are managed directly by the underlying STM layer. This approach allows for transparently extending the classic STM programming model to a distributed setting.

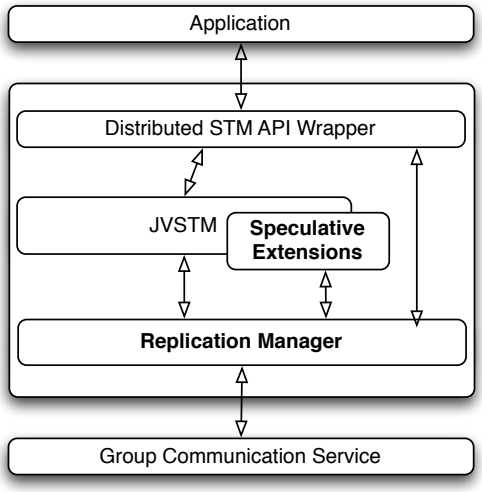


Figure 1: Architecture of a SCert replica.

The mechanisms for maintaining and managing speculative data item versions are provided by the two core components of the SCert protocol: the STM's Speculative Extensions (SE) and the Replication Manager (RM). We have opted to implement the SE for a multi-versioned STM, namely JVSTM [7]. As already mentioned in the Introduction section, JVSM maximizes the performance of read-only transactions and, since it already embodies mechanisms to maintain multiple copies of the same data, it lends itself naturally to support the additional speculative data item versions required by SCert. We will detail thse mechanisms in Section 5.2. The RM is responsible of coordinating the commit phase, implementing the speculative certification scheme by leveraging on the services provided by the STM, the SE and the GCS.

Finally, the bottom layer is the Group Communication Service [10], which provides the view synchronous OAB services. All the experiments described in this paper have been performed using the Appia GCS [28]. However, to maximize portability, our implementation uses a generic GCS, namely JGCS [8], which supports several other group communication implementations.

5. THE SCERT PROTOCOL

Before delving in the detailed description of the SCert protocol, we provide a brief, informal overview of its key mechanisms. As in conventional certification protocols, e.g. [11], in SCert transactions are run locally, without incurring in any replica coordination during their execution. Once a transaction reaches its commit phase, it is first locally validated and then its read-set and write-set are disseminated to all replicas by means of an (optimistic) atomic broadcast. Unlike conventional certification protocols, however, SCert does not wait until the final delivery order of the atomic broadcast is known to certify the transaction. Instead, SCert speculatively certifies a transaction as soon as the broadcast is optimistically delivered. If the validation succeeds,

the transaction is *speculatively committed*.

Note that the application call to commit a transaction does not return if the transaction is only *speculatively committed*. Therefore, user-level code is not affected by mispeculations that may result from a mismatch between the optimistic and final delivery orders. Still, the post-images (i.e. the values of the write-set) of a speculatively committed transaction are applied (added) to the STM and marked as *speculative*. A speculatively committed transaction will eventually be *finally committed*, its updates marked as *committed* and the user-level code allowed to return from the invocation of the commit method. Speculative values only become committed values if there is no mismatch between the optimistic and the final order of the OAB or, when a mismatch occurs, if the transaction can be safely re-ordered. Roughly speaking, the latter case corresponds to scenarios in which the transaction did not develop any read-from dependency from transactions that were speculatively committed in a serialization order not conciliable with the final delivery order.

Speculatively committed versions of data items (simply named speculative versions) are immediately made available to new transactions. Therefore, new transactions are tentatively serialized after the last speculatively committed transaction, thus improving their chances to observe a non-stale snapshot. In the following, transactions that are activated while the local STM maintains speculative versions will be denoted as *speculative transactions*.

In SCert, speculative transactions that enter their commit phase can also atomically broadcast, in their turn, a certification request. Upon the optimistic delivery of a speculative transaction T, T is validated to detect conflicts not only against committed transactions, but also against speculatively committed transactions that were optimistically delivered before T. This allows to generate a chain of speculatively committed transactions that are serialized in an order compliant with the sequence of optimistic deliveries. In other words, during the time window that starts with the optimistic delivery of a transaction T and ending with its final delivery, SCert strives to serialize any concurrently executing $T^1, \ldots, T^n$ according to their optimistic delivery order, achieving an overlap between communication and processing that is not possible with a conventional (non-speculative) certification scheme.

Furthermore, SCert also exploits speculative versions to implement early conflict detection. As soon as a transaction T is speculatively committed, any other local transaction that i) was serialized before T, and that ii) has read, or reads, a data item updated by T is immediately aborted. Whenever the optimistic order matches the final delivery order, this early abort mechanism prevents the waste of time/computational resources with respect to conventional certification schemes, where conflicts are only detected upon the final AB delivery.

The remainder of this section is structured as follows. In Section 5.1, we start by providing an overview of the key mechanisms of JVSTM, followed by a discussion, in Section 5.2, on how JVSTM has been extended to maintain and manage speculative versions. Next, in Section 5.3, we describe how the Replication Manager orchestrates the execution of transactions across the distributed STM platform. In Section 5.4 we highlight the performance benefits of SCert, by illustrating some of its execution sketches. Finally, in Section 5.5 we provide some informal arguments on its correctness.

5.1 Overview of JVSTM's internals

JVSTM implements a multi-version scheme which is based on the abstraction of a *versioned box* (VBox). A VBox is a container that keeps a tagged sequence of values - the history of the versioned box. Each of the history's values corresponds to a change made to the box by a successfully committed transaction and is tagged with the timestamp of the corresponding transaction. The versions of VBox are arranged into a linked list, whose head maintains the version created by the last transaction that committed (and issued a write on the VBox).

To keep track of the serialization order of transactions, JVSTM maintains a global integer timestamp, *commitTimestamp*, which is incremented whenever a transaction commits. Each transaction stores its timestamp in a local *snapshotID* variable, which is initialized at the time of the transaction activation with the current value of *commitTimestamp*. This information is used both during transaction execution, to identify the appropriate values to be read from the VBoxes, and, at commit time, during the validation phase, to determine the set of concurrent transactions to check against possible conflicts.

More in detail, when a transaction T having *snapshotID*=s issues a read operation on a VBox X, JVSTM returns the version stored in X associated with the largest timestamp smaller or equal to s. In other words, it returns the version created by the last transaction that i) has issued a write on X and ii) was serialized before T. For what concerns write operations, JVSTM stores the values written by a transaction in a private buffer, and applies them to the corresponding VBoxes only at commit time, provided that the transaction passes a validation phase.

The validation is performed by checking whether any of the VBoxes read by a transaction T has been updated by some committed transaction T' with a larger timestamp. In this case T is aborted. Otherwise, T is committed by atomically executing (within a critical section) the following operation. The *commitTimestamp* variable is increased, and the transaction's *snapshotID* is set to the new value of *commitTimestamp*. Finally the new values of all the VBoxes written by the transaction are appended to the linked list of versions tagged with the current value of *commitTimestamp*.

As a final note, JVSTM integrates a garbage collection mechanism that detects if there are versions stored within some VBox that are no longer visible by any currently active transaction. The interested reader may refer to [7] for a detailed description of this mechanism.

5.2 JVSTM Extensions for Speculative Transactions

In order to maintain and manage speculative versions, the following extensions have been integrated in JVSTM. In addition to *commitTimestamp*, JVSTM now maintains an additional global timestamp called *speculativeTimestamp* that is incremented whenever a transaction is speculatively committed. Note that since a transaction is only committed after it is final delivered, and given that a final delivery for a message is always preceded by its optimistic delivery, it follows that *speculativeTimestamp* $\geq$ *commitTimestamp*.

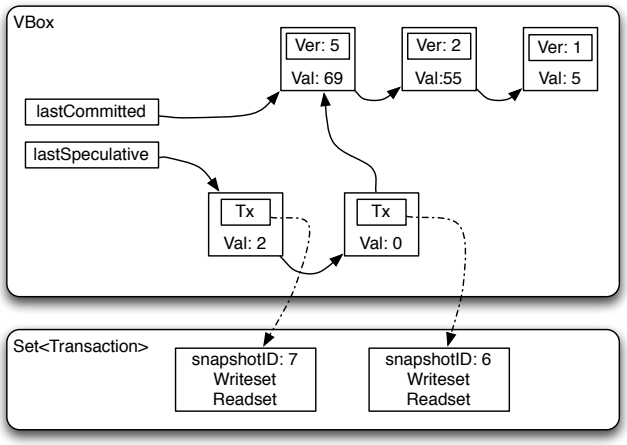


Figure 2: VBox with speculative versions.

Also, *speculativeTimestamp* = *commitTimestamp* only if currently there are no speculatively committed transactions (and consequently speculative data items' versions). Whenever a transaction is activated, its *snapshotID* variable is assigned the current value of *speculativeTimestamp*, thus serializing it after the last speculatively committed transaction.

To distinguish between speculative and non-speculative data item versions, we extended the original JVSTM VBox data structure as shown in Figure 2. The VBox stores both committed and speculative versions into a single linked list, maintaining one reference to the most recent committed version (*lastCommitted*) and one to the most recent speculative version (*lastSpeculative*). Unlike final committed versions, versions created by speculatively committed transactions are not associated with a version timestamp; instead, they store a reference to a data structure that keeps the *snapshotID*, the read-set, and the write-set of the (local or remote) transaction that created them. As we will see in Section 5.3, this indirection mechanism allows to manage efficiently the case in which, due to mismatches between optimistic and final delivery orders, speculative transactions need to be final committed in an order different from that in which they had been originally speculatively committed.

These additional data structures are used to implement early conflict detection. In JVSTM, if during a read operation a transaction T realizes that a version tagged with a timestamp larger than its own *snapshotID* is already stored within the VBox, T is not aborted right away. Conversely, T navigates the linked list of versions to retrieve the version generated by the last transaction that committed before T started. This prevents read-only transactions from aborting (by serializing them in the past). Unfortunately, for update transactions, even though at this stage T is already doomed to abort, JVSTM will only abort it during the validation phase taking place during T's commit phase. In our speculative version of JVSTM, we patched this suboptimal behavior by immediately aborting an update transaction T that reads a VBox for which a (committed or speculatively committed) version exists with a timestamp larger than T's *snapshotID*.

Furthermore, in order to allow the RM to orchestrate the SCert replication protocol, the JVSTM API has been extended with the following primitives:

specCommit**(Transaction T)** This method speculatively commits a transaction T. To this end, it increases *speculativeTimestamp*, assigns the new value to T's *snapshotID*, and updates the VBoxes of all items in T's write set. The VBoxes are updated as follows: i) a new speculative version is added to the head of the versions' list tagged with the current value of *speculativeTimestamp*, and ii) the *lastSpeculative* pointer is set to this new version.

specAbort**(Transaction T)** This method is used to abort a previously speculatively committed transaction. To this end, it eliminates the corresponding speculative versions of data items in T's write-set from their VBoxes (updating the *lastSpeculative* pointer accordingly).

specValidate**(Transaction T)** This method validates T by iterating over its read-set and returning true only if T has read the most recent *speculatively or finally* committed version of each data item.

validate**(Transaction T)** This method validates T by iterating over its read-set and returning true only if T has read the most recent *finally* committed version of each data item.

commit**(Transaction T)** This method finalizes the commit of a transaction T that is currently the oldest of the speculatively committed transactions. To this end, it increases the *commitTimestamp* and updates the VBoxes of all items in T's write-set. The VBoxes are updated as follows: i) the speculative version previously created by T is replaced by a non-speculative version tagged with the current value of *commitTimestamp*, and ii) the *lastCommitted* pointer is set to this new version.

abort**(Transaction T)** This method aborts the transaction T. Since the transaction will not be applied to memory, all the data of this transaction is discarded, including the read-set and the write-set.

specOutOfOrderCommit**(Transaction T)** This method is used to speculatively commit a transaction without adding its write-set to the head of the linked list of versions. To this end, it first increases *speculativeTimestamp* and assigns it to T's *snapshotID*. Next, for each data item in T's write-set, it inserts into the corresponding VBoxes a speculative version after the speculative version created by the transaction with the largest not null *snapshotID*. If no such transaction exists, T's version is just inserted after the last committed version. Finally, the *lastSpeculative* pointer is set to refer to the version created by T.

outOfOrderCommit**(Transaction T)** This method is used to commit a transaction T that is not currently the oldest of the speculatively committed transaction. As we will see this is possible either because T was not previously speculatively committed (upon its optimistic delivery), or because it was speculatively committed in a different order. In both cases the *commitTimestamp* is increased and the *lastCommitted* pointer is set to refer

to a new non-speculative data item version that is inserted between the last finally committed version and the first speculatively committed version (if any). If T had previously been speculatively committed, however, any speculative version it had previously stored in JVSTM is also erased.

5.3 Replication Manager

The pseudocode describing the behavior of the RM is shown in Algorithm 1 and Algorithm 2. As outlined before, transactions execute in a single machine, accessing the most recent speculatively committed snapshot available at the time they were activated. The RM is activated whenever a local transaction requests to commit. At this point, the transaction undergoes first a local validation. Conflicts with concurrent transactions that have already locally (speculatively or finally) committed are detected at this stage. If this validation fails, the transaction is immediately aborted. Otherwise, its read-set, write-set, and *snapshotID* are sent to all replicas using the OA-broadcast primitive (described in the Section 3). At this point, the user call becomes blocked until the transaction outcome is defined.

A transaction is received by all nodes twice. The first time, it is received by the Opt-deliver primitive, which provides an early estimate of the final delivery order. As already discussed, SCert leverages on the observation that in a local network, the spontaneous order of delivery of the messages from the network coincides, with high probability [22], with the final total delivery eventually determined by the OAB service.

Optimistic Delivery

When the transaction is optimistically delivered, it is validated to detect possible conflicts with the transactions that committed so far, either finally or speculatively. This phase is called *speculative validation*. If it successfully passes this phase, the transaction is speculatively committed and the transaction is appended to the *specComm* set. Otherwise, the transaction is added to the *specAborted* set. Note that at this stage the transaction is not aborted yet. The transaction may in fact be still committed if, upon its final delivery, a mismatch between the optimistic and final delivery orders is detected, and if the serialization order determined by the final delivery order results to be equivalent to the one in which the transaction was originally processed. In both cases, the transaction is added to the *optDel* queue. As we will see, this queue will be later used to detect possible mismatches between the optimistic and final delivery orders.

Final Delivery of Aborted Transactions

Upon TO-delivery of a transaction T, it is first checked (via the isFinalAborted method) if the transaction has already been aborted. As we will see, this can happen in case T had observed the speculative snapshot generated by a transaction T' that was later on aborted, generating the cascading abort of T. In this case, T is simply removed from the *optDel* queue.

Final Delivery with "Matching-Order"

If the outcome of T has still to be determined, it is checked whether T is at the head of the *optDel* queue. If it is true, this means that the final delivery order matches the optimistic delivery order. In this case, the transaction's outcome (abort or commit) can be easily determined by checking whether the transaction has been placed in the *specComm* set or in the *specAborted* set. If the transaction had executed locally and was speculatively aborted, the local instance of JVSTM is notified. This last step is not necessary if the transaction has been executed remotely, as the local instance of JVSTM has no knowledge of the transaction. If the transaction is committed, the COMMIT() method is called to update the VBoxes as detailed in Section 5.2.

Algorithm 1: Replication Protocol (Part I).

```
FIFOQueue<Transaction> optDel = ∅
Set<Transaction> specComm = ∅
Set<Transaction> specAborted = ∅

void commit (Transaction T)
  if ( ¬ JVSTM.specValidate (T) ) then
     JVSTM.abort (T)
  else
     trigger OA-broadcast [T]
     wait until ( isTransactionFinished (T) ∨ ejected )
     if ( ejected ) then
        JVSTM.abort (T)

boolean isTransactionFinished (Transaction T)
  return ( JVSTM.isAborted (T) ∨ JVSTM.isCommitted (T) )

upon event Opt-deliver ([Transaction T]) atomically do
  optDel.add (T)
  if ( ¬JVSTM.validate (T) ) then
     JVSTM.abort (T)
  else
     if ( ¬JVSTM.specValidate (T) ) then
        specAborted.add (T)
     else
        specComm.add (T)
        JVSTM.specCommit (T)

upon event TO-deliver ([Transaction T]) atomically do
  if ( JVSTM.isFinalAborted (T) ) then
     optDel.remove (T)
  else
     if ( optDel.getFirst () ≠ T ) then
        handleOutOfOrder (T)
     else
        optDel.removeFirst ()
        if ( specAborted.contains (T) ) then
          specAborted.remove (T)
          JVSTM.abort (T)
        else
          specComm.remove (T)
          JVSTM.commit (T)
```

Final Delivery with "Mismatching-Order"

On the other hand, T is not at the head of the *optDel* queue, then a mismatch between the optimistically delivery and final delivery has occurred. Naturally, this is the most complex scenario that has to be managed by SCert. The pseudocode for this case is depicted in the handleOutOfOrder() method (see Algorithm 2).

After removing T from the *optDel* queue, T is validated to detect whether, despite the misalignment between the optimistic and final delivery orders, it can still be serialized immediately after the last finally committed transaction. If this validation fails and T had not been previously speculatively committed, T can be aborted right away, since no other transaction may have ever observed its snapshot.

Additional care is needed in the following two cases:

Algorithm 2: Replication Protocol (Part II).

```
void handleOutOfOrder (Transaction T)
 optDel.remove (T)
 boolean outcome = JVSTM.validate (T)
 if ( ¬ outcome ∧ specAborted.contains (T)) then
  // avoid revalidate other txs
  specAborted.remove (T)
  JVSTM.abort (T)
 else
  temporarily block activation of new transactions
  abort local transactions not yet in their commit phase
  if ( ¬ outcome ) then
    specComm.remove (T)
    JVSTM.abort (T)
  else // tx out of order, but still committable
    if ( specAborted.contains (T) ) then specAborted.remove (T)
    if ( specComm.contains (T) ) then specComm.remove (T)
    JVSTM.outOfOrderCommit (T)
  revalidateOptDelTxs ()
  unblock activation of new transactions

void revalidateOptDelTxs ()
 JVSTM.lastSpeculativeTimestamp =
  JVSTM.lastCommittedTimestamp
 foreach Transaction T ∈ optDel ∧¬ JVSTM.isFinalAborted (T) do
  // reset snapshotIDs before re-assigning them
  T.snapshotID = null
 foreach Transaction T ∈ optDel ∧¬ JVSTM.isFinalAborted (T) do
  if ( ¬JVSTM.validate (T) ) then
    JVSTM.abort(T)
  else
    if ( ¬JVSTM.specValidate (T) ) then
      if ( specComm.contains (T) ) then
        // Tx prev. speculatively committed
        specComm.remove (T)
        specAborted.add (T)
        specAbort (T)
    else // Tx passed speculative validation
      if ( specAborted.contains (T) ) then
        // Tx prev. speculatively aborted
        specAborted.remove (T)
        specComm.add (T)
        JVSTM.specOutOfOrderCommit (T)
      else // Tx already spec. committed, update its snapshotID
        T.snapshotID = ++JVSTM.lastSpeculativeTimestamp
```

Algorithm 3: Replication Manager at process p_i - Dealing with View Changes.

```
View currentView={p1,...,pi,...,pn}
boolean inPrimaryComponent=true

upon event ViewChange(View newView) do
 if (¬inPrimaryComponent ∨ pi is joining for the first time) then
   perform state transfer
   inPrimaryComponent=true
 else
   ∀pj s.t. (pj ∈ currentView ∧ pj ∉ newView) do
     ∀ T ∈ specComm s.t. T.proc = pj do
       specComm.remove (T)
     ∀ T ∈ specAborted s.t. T.proc = pj do
       specAborted.remove (T)
     ∀ T ∈ optDel s.t. T.proc = pj do
       optDel.remove (T)
 currentView = newView

upon event ejected do
 inPrimaryComponent=false
```

- T had previously been speculatively committed, but it needs to abort. In this case, in fact, T's snapshot may have already been observed by other transactions, that may possibly be still executing (i.e. not yet in their commit phase).
- T may be (finally) committed. In this case, either T had been previously speculatively aborted, or had been speculatively committed in a different serialization order. Either way, this can impact both the speculative decision (commit/abort) already taken for the remaining optimistically delivered transactions, and the snapshots observed by currently executing transactions.

In order to avoid currently executing transactions from accessing inconsistent snapshots and suffering of anomalies due to the loss of opacity [15], the required readjustments of the speculative snapshots are done only after having blocked new transactions from starting and after having aborted any ongoing transaction. Also, only after having concluded readjusting the speculative snapshots, the activation of new transactions will be allowed again.

The snapshot realignment consists of the following steps. First, the outcome of transaction T is finalized either via the ABORT() or the OUTOFORDERCOMMIT(), depending on the output of its validation phase. At this point, in the REVALIDATEOPTDELTXS() method, the remaining optimistically delivered transactions are revalidated to take into account the unexpected order in which T was committed. This also includes reassigning the *snapshotID* timestamps to every transaction which were found to be speculatively committable. To achieve this result, SCert starts by setting the *lastSpeculativeTimestamp* to *lastCommitTimestamp* and resetting the *snapshotID*s of all the optimistically delivered transactions. This has the effect of resetting the STM to the state it had before having speculatively committed any of the optimistically delivered transactions. Next, SCert iterates over these transactions following their (updated) order of optimistic delivery. Each of them is first validated against the already committed transactions and, in case the first validation succeeds, against those that have already been speculatively committed. If the speculative validation fails, the transaction is simply speculatively aborted. If it succeeds, however, it is checked whether the transaction had previously been speculatively committed or aborted. In the former case, it means that its snapshot is already present in memory. Thus it suffices to increase the *lastSpeculative* timestamp and assign its updated value to the transaction's *snapshotID*. If the transaction was previously speculatively aborted, instead, its write-set must be applied, in the right order, in the linked list of versions maintained by the corresponding VBoxes. This is done using the SPECOUTOFORDERCOMMIT() primitive (see Section 5.2).

Dynamic Membership

It remains to discuss the replicas' behavior in the presence of view changes and ejections from the primary component view, which is shown in the pseudo-code in Algorithm 3. Upon delivery of a new view event, if the replica re-joins the primary component or is joining the group of replicas for the first time, it triggers a state transfer procedure that realigns the content of the local replica of the STM, as well as of the state variables of the replication protocol. The state transfer procedure is a complex task that can be solved using several existing mechanisms (e.g. [19]), and will not be further detailed in this paper.

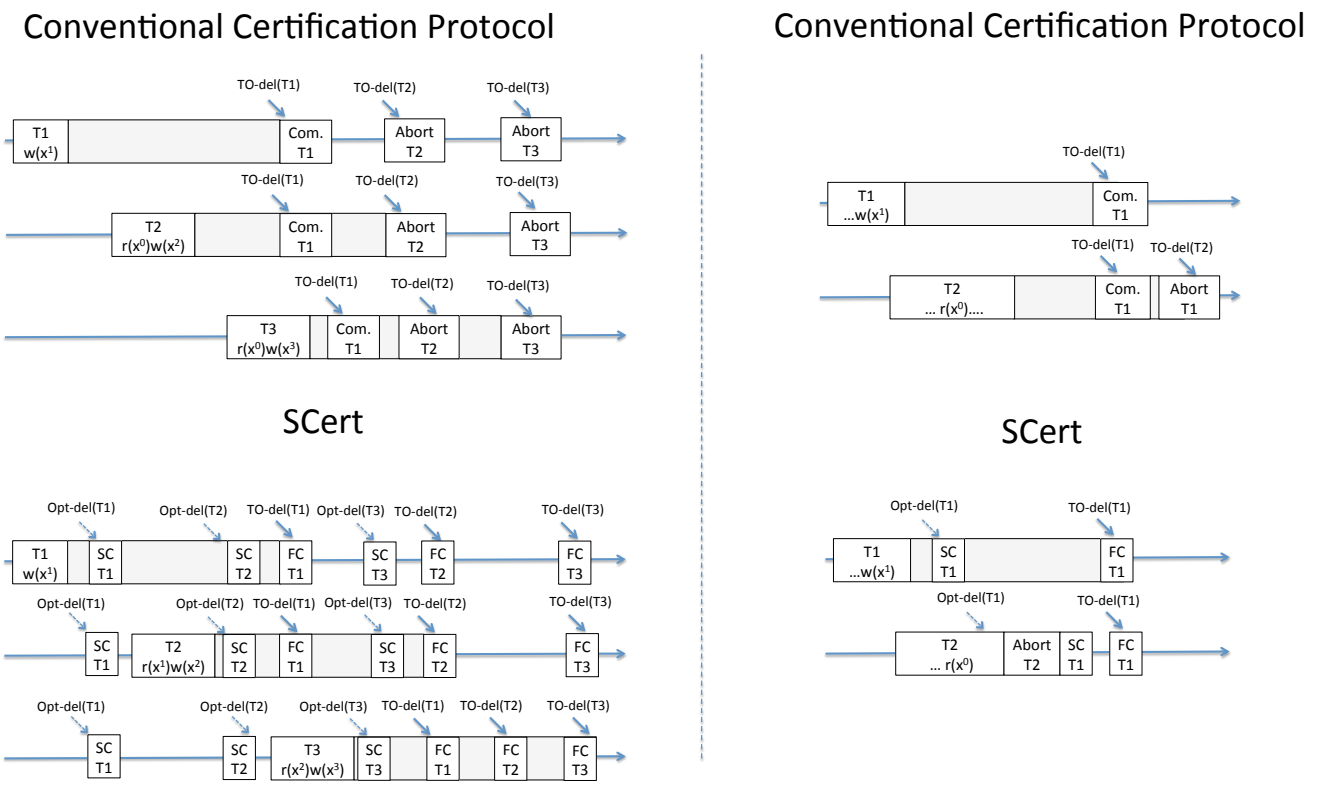


Figure 3: Comparing SCert with a classic certification protocol. SC and FC stand for, respectively, Speculative Commit and Final Commit.

5.4 Example execution sketches

To clarify the dynamics of the SCert protocol, in the diagram of Figure 3 we illustrate two sketches of its execution, contrasting it with analogous execution sketches for a conventional certification protocol, such as [11, 32].

On the left side of the diagram, it is shown the execution of three conflicting concurrent transactions, T1, T2 and T3, which all issue a read and write operation on a data item X. Let us assume that the transactions are executed on different replicas, even though the same considerations drawn in the following would apply in case the transactions were all executing in the same machine. Note that the execution times of transactions and atomic broadcast are not in scale as, in typical STM applications, the average transaction execution time is normally several orders of magnitude smaller than the completion time of atomic broadcast.

In non speculative certification schemes, the post-images of the data updated by transactions are propagated only after the corresponding message is final delivered. As the level of concurrency among transactions grows, the chances that transactions miss the snapshots generated by previously completed transactions increase significantly, leading to a corresponding increase of the abort rate. Due to this, in the example reported in Figure 3, both transactions T2 and T3 would need to abort, as both have read an obsolete version of X. In SCert, conversely, transactions T2 and T3 can benefit from the early propagation (via optimistic delivery) of speculatively committed snapshots and can be successfully committed if, as considered in this example, there is no mismatch between optimistic and final delivery orders.

Note that, in this example, the speculative propagation of snapshots takes place through a chain of transactions, as T2 reads the version of X generated by T1, and T3 observes the version of X written by T2. This brings two main benefits: i) it reduces the abort rate of concurrent transactions by exposing fresh data to the system sooner, ii) it allows overlapping the processing of transactions with the commit process. The execution sketch shown on the right side of Figure 3 illustrates the benefits deriving from the early abort notification scheme provided by SCert. Even in scenarios where it is not possible to propagate the snapshots of a concurrent transaction in time, as in the case of T2 that has already issued a read operation on X before T1 is optimistically delivered, SCert exploits speculation to abort immediately transactions that will certainly abort once that they will be final delivered in absence of mismatches between the optimistic and final delivery orders. Clearly, the effectiveness of SCert depends significantly on the probability that the optimistic order matches the final (total) order and, consequently, it results particularly attractive in Local Area Networks, where the probability that the network spontaneous order will match the final order is high. When considering, for instance, the scenario on the left side of Figure 3, had the final delivery order been {T3,T2,T1}, SCert would have induced the abort of T3 and (the cascading abort of) T2, committing only T1. It is noteworthy to highlight, however, that, also this worst case scenario for SCert, SCert would not have been outperformed by a conventional certification protocol; a non-speculative protocol would also have committed only one transaction (namely T3).

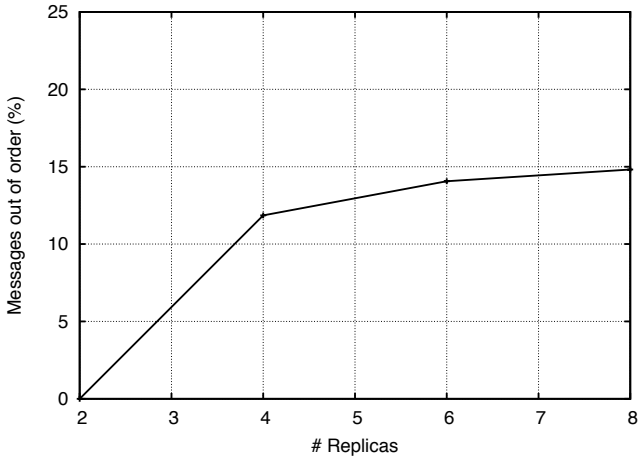
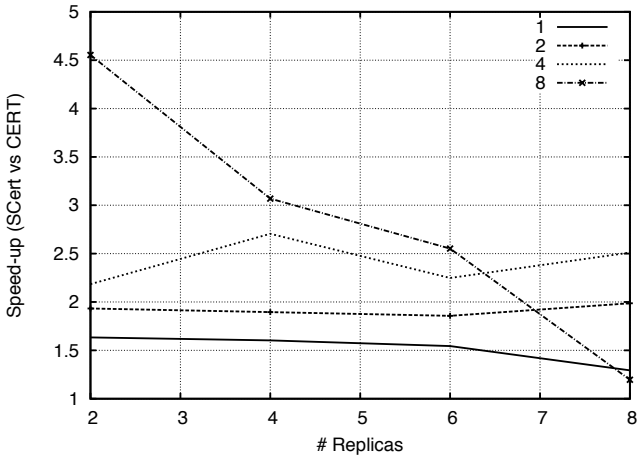


Figure 4: Mismatches between optimistic and final deliveries (Bank benchmark).

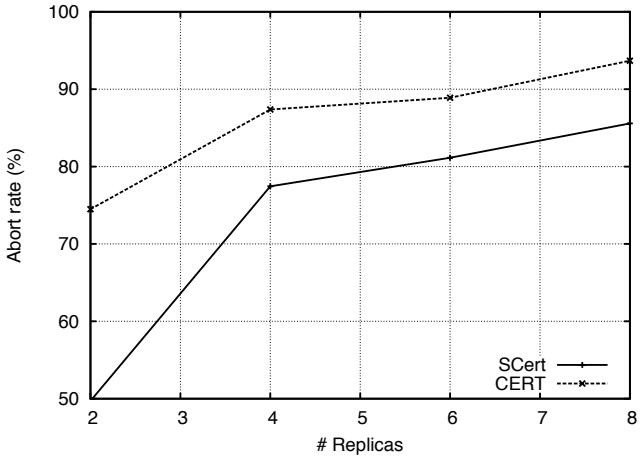
5.5 Correctness arguments

Our target consistency criterion for replication is 1-copy serializability [3], which ensures that the execution history of committed transactions across the whole set of replicas is equivalent to a serial transaction execution history on a not replicated STM. In SCert a transaction returns from an application's commit request only if the OAB service has established its final delivery order for which group-wide consensus is ensured. Further, a replica final commits a transaction T only if it passes a deterministic validation phase that ensures that T has been serialized in an order compliant with the OAB's final delivery order. To this end, SCert performs a first speculative validation upon the optimistic delivery of transactions. At this stage, however, no irreversible decision on the transaction's outcome is taken, or is externalized to user level applications. This only occurs upon final delivery of transaction. If at this point, it is found out that the optimistic and final delivery orders coincided, SCert avoids re-validating the transaction (as this would yield the same result of the speculative validation), and simply confirms the outcome of the speculative validation, final committing or aborting the transaction. If, on the other hand, upon the final delivery of transaction T a replica detects that the optimistic delivery order has been contradicted by the final delivery order, a corrective action is taken which re-validates both T and every optimistically (but not yet finally) delivered transaction. This ensures that the final decision taken on T's outcome is identical at each replica. Also, it guarantees that the outcome of the speculative validation for optimistically delivered transactions is consistent with the updated optimistic and final delivery orders. This allows to safely avoid further validations in the future, if optimistic and final delivery orders were to no longer diverge.

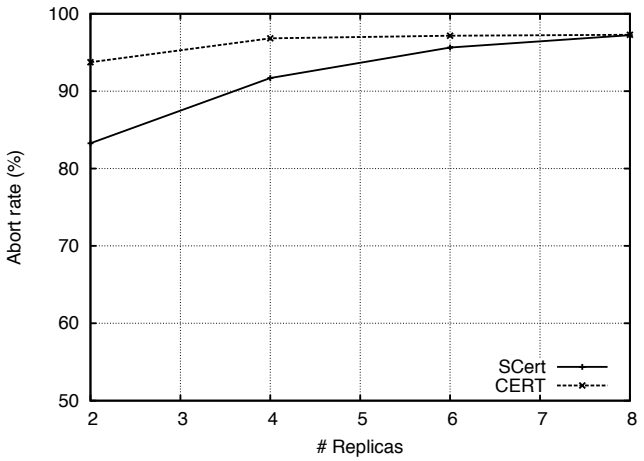
SCert preserves the strong atomicity [27] and opacity [15] properties provided by JVSTM. The former property avoids conflicts among transactional and non-transactional memory accesses. Opacity [15], on the other hand, can be informally viewed as an extension of the classical database serializability property with the additional requirement that also non-committed transactions are prevented from observing inconsistent states, namely snapshots that could not be



(a) Speed-up.

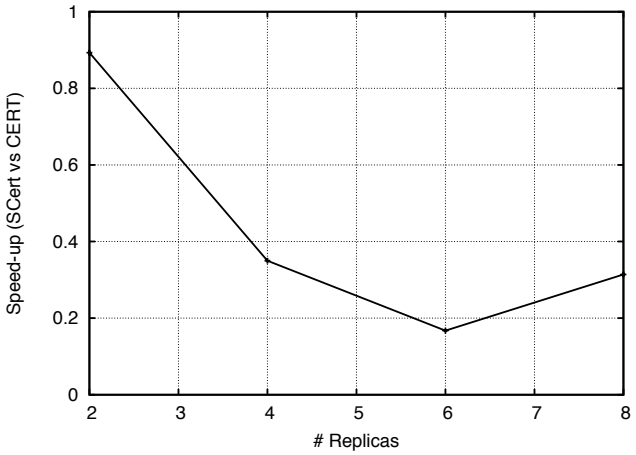


(b) Abort Rate (1 Thread).

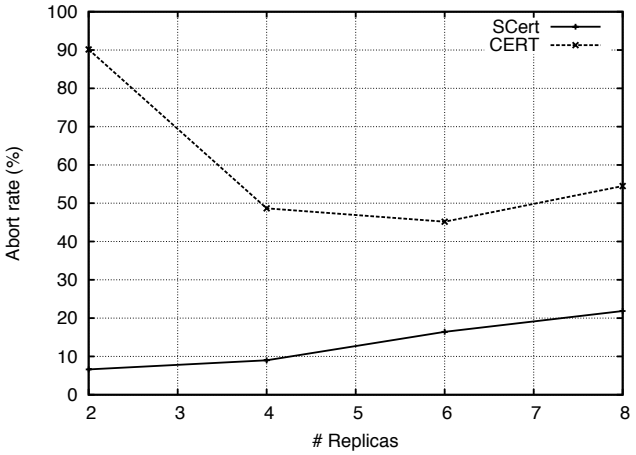


(c) Abort Rate (8 Threads).

Figure 5: Bank Benchmark.



(a) Speed-up.



(b) Abort Rate.

Figure 6: STMBench7.

generated in any sequential transaction execution history.

Strong atomicity is ensured by JVSTM at the language level, via the VBox abstraction, which prevents the possibility for any non-transactional manipulation of its state. Layering on top of JVSTM, and sharing the reliance on the VBox abstraction, SCert simply inherits this property.

Opacity in SCert is guaranteed since every transaction is forced, *since its start*, to observe a consistent snapshot obtained by sequentially executing transactions according to the order defined by the delivery of messages of the OAB service. More in detail, SCert forces the underlying STM to serialize all transactions according to the optimistic delivery order. The validation phase performed as transactions are optimistically delivered, in fact, allows speculatively committing a transaction only if it can be serialized after the last speculatively committed transaction. In absence of mismatches between the optimistic and final delivery orders, this serialization order is then simply confirmed as the OAB establishes the final delivery order. Were the two message delivery orders differ, SCert atomically aborts any transaction that was speculatively serialized in an order that is not conciliable with the final delivery order and reassigns timestamps to the remaining speculatively transactions. Before doing this, however, SCert aborts every ongoing transaction and prevents new transactions to start until this reconciliation phase is completed. This clearly rules out the possibility that ongoing transactions can observe any inconsistent state during this phase.

6. PERFORMANCE EVALUATION

In this section we report results from an experimental study aimed at quantifying the performance gains achievable by SCert when compared with non-speculative certification based protocols. To this purpose, we have used as a baseline the D^{2}STM [11] protocol. D^{2}STM is an Atomic Broadcast based certification scheme that embodies a number of optimizations to maximize performance by minimizing the size of the messages disseminated during the certification phase via Atomic Broadcast. In the remaining of the text we refer to this protocol solely as "CERT".

Our testbed platform consists of a cluster of 8 nodes, each one equipped with two Intel Quad-Core XEON at 2.0 GHz, 8 GB of RAM, running Linux 2.6.32-26-server and interconnected via a private Gigabit Ethernet. The results shown in this section are an average of 10 runs, each lasting between 10 and 30 minutes.

6.1 Bank Benchmark

We start by considering a synthetic workload, obtained by adapting the Bank Benchmark originally used in [17]. In this benchmark, each transaction transfers an amount between variables representing distinct bank accounts. This is a simple benchmark that has the advantage of providing a fine control on the conflict rate. We have initialized the benchmark such that all nodes replicate an array of accounts of size $numMachines \cdot numThreads \cdot 2$ items. Depending on the accounts accessed by each transaction, we can have from 0% to 100% conflicts among concurrent transactions.

We performed experiments to evaluate the performance of SCert in a scenario where all transactions touch the same accounts, i.e., where we have 100% conflicts. The number of replicas varied between 2 and 8, and we fixed the number of threads on each replica. Depending on the test, we configured the number of threads between 1 or 8. Since all nodes are continuously processing very small transactions and sending OAB messages, this quickly saturates the group communication service and generates a significant amount of contention in the network.

Figure 4 shows the number of messages that were delivered out of order by the Opt-delivery primitive when 8 threads are used. As it is shown in the figure, even in a high network contention scenario the number of messages optimistically delivered out of order never goes over 15%.

Figure 5 reports the speed-ups achieved by SCert with regard to CERT, as well as the observed abort-rate for both SCert and CERT. It can be observed that SCert is able to improve the system performance up to a striking 4.5x factor (see Figure 5(a)) when compared with CERT, even for small reductions in the abort rate. SCert provides the best results when the conflict rate is high but the network is not saturated (this is achieved by using 8 threads on just 2 replicas). When the network load increases (more replicas are used),

the performance advantages of SCert decrease but, in most cases, SCert is still able to reduce significantly the abort rate in the system (as shown in Figures 5(b) and 5(c)).

6.2 STMBench7

We now show results using STMBench7, a richer benchmark featuring a number of operations with different levels of complexity over an object-graph with millions of objects. Figure 6 depicts the performance of both protocols using the "write dominated" workload. As before, each plot shows the speed-up of SCert over CERT and the abort rate of both protocols (SCert and CERT). The number of replicas varies between 2 and 8 and we fixed the number of threads to 2. Unsurprisingly, the speed-ups achieved by SCert (Figure 6(a)) are higher in the scenarios where CERT suffers from higher abort rates (Figure 6(b)). This shows that also with realistic, complex applications, like STMBench7, the speculation mechanism employed by SCert succeeds in significantly reducing the abort rate, boosting throughput, on average, by about 45%.

7. CONCLUSIONS

This paper has introduced a new Speculative Certification protocol, named SCert, to implement distributed replicated STMs. SCert leverages on Optimistc Atomic Broadcast (OAB) protocols to speed-up the propagation of write-sets, reducing the number of transactions that read stale data and allowing early detection of conflicts among transactions. This novel manner of exploiting OAB is much more suited for STM implementations than previous strategies designed for database replication, that were based on active replication. By aggressively using speculation, SCert is able to achieve performance gains of up to 4.5x when compared to non-speculative certification schemes. We also plan to apply the key idea underlying the SCert protocol, namely speculatively propagating information on the data accessed by transactions prior to the completion of the group communication primitive used to disseminate this information, to the Asynchronous Lease proptocol described in [9].

8. REFERENCES

[1] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *Proc. of the 21st ACM SIGOPS Symp. on Operating Systems Principles (SOSP)*, pages 159–174. ACM, 2007.

[2] A. Bartoli and O. Babaoglu. Selecting a "primary partition" in partitionable asynchronous distributed systems. In *IEEE Symp. on Reliable Distributed Systems (SRDS)*, page 138, Los Alamitos, CA, USA, 1997.

[3] P. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[4] R. Bocchino, V. Adve, and B. Chamberlain. Software transactional memory for large scale clusters. In *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 247–258. ACM, 2008.

[5] A. Brito, C. Fetzer, and P. Felber. Multithreading-enabled active replication for event stream processing operators. In *Proc. of the 2009 28th IEEE Int'l Symp. on Reliable Distributed Systems (SRDS)*, pages 22–31, Washington, DC, USA, 2009.

[6] J. Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Technical University of Lisbon, 2007.

[7] J. Cachopo and A. Rito-Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In *Proc. of the 6th ACM Int'l Conf. on Web engineering*, ICWE '06, pages 297–304, New York, NY, USA, 2006.

[8] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In *Proc. of the Int'l Symp. on Distributed Objects and Applications (DOA)*, 2006.

[9] N. Carvalho, P. Romano, and L. Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proc. of the ACM/IFIP/USENIX 11th Middleware Conf.*, Bangalore, India, Nov. 2010.

[10] G. V. Chockler, I. Keidar, and R. Vitenberg. Group communication specifications: a comprehensive study. *ACM Computing Surveys (CSUR)*, 33(4):427–469, 2001.

[11] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D^{2}STM: Dependable Distributed Software Transactional Memory. In *Proc. of the 15th Pacific Rim Int. Symp. on Dependable Computing (PRDC)*, Shanghai, China, Nov. 2009.

[12] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.

[13] D. Dice, O. Shalev, and N. Shavit. Transactional locking II. In *Proc. of the Int'l Symp. on Distributed Computing (DISC)*, pages 194–208, 2006.

[14] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the Conf. on the Management of Data (SIGMOD)*, pages 173–182. ACM, 1996.

[15] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of the 13th ACM SIGPLAN Symp. on Principles and practice of parallel programming (PPoPP)*, pages 175–184, 2008.

[16] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.

[17] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

[18] IST. Fenixedu. http://fenixedu.sourceforge.net, 2009.

[19] R. Jiménez-Peris, M. Patiño Martínez, and G. Alonso. Non-intrusive, parallel recovery of replicated data. In *Proc. of the 21st IEEE Symp. on Reliable Distributed Systems (SRDS)*, page 150, Washington, DC, USA, 2002.

[20] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *Proc of the 19th Int. Symp. on Computer Architecture (ISCA)*, pages 13–21, New York, NY, USA, 1992. ACM.

[21] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the IEEE Int'l Conf. on*

Distributed Computing Systems (ICDCS), page 156, 1998.

[22] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the 19th IEEE Int'l Conf. on Distributed Computing Systems (ICDCS)*, page 424, 1999.

[23] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *Proc. of the Int'l Conf. on Parallel Processing (ICPP)*, pages 51–58, 2008.

[24] C. Kotselidis, M. Lujan, M. Ansari, K. Malakasis, B. Kahn, C. Kirkham, and I. Watson. Clustering JVMs with software transactional memory support. In *Int'l Symp. on Parallel Distributed Processing (IPDPS)*, pages 1 –12, April 2010.

[25] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proc. of the Symp. on Principles of Distributed Computing*, pages 229–239. ACM, 1986.

[26] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proc. of the Symp. on Principles and Practice of Parallel Programming (PPOPP)*, pages 198–208. ACM, 2006.

[27] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Computer Architecture Letters*, 5(2):17, 2006.

[28] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. Int'l Conf. on Distributed Computing Systems (ICDCS)*, pages 707–710. IEEE, 2001.

[29] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting STM replication via aggressively optimistic transaction processing. In *9th IEEE Int'l Symp. on Network Computing and Applications (NCA)*, pages 20 –27, July 2010.

[30] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *In Proc. of the IEEE Int'l Symp. on Parallel Distributed Processing, Workshops and PhD Forum (IPDPSW)*, pages 1 –8, April 2010.

[31] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the Int'l Conf. on Distributed Computing (DISC)*, pages 315–329. Springer-Verlag, 2000.

[32] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.

[33] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science - Special issue: Distributed computing*, 291:79–101, January 2003.

[34] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *Proc. of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS)*, 2008.

[35] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues. An optimal speculative transactional replication protocol. In *Int'l Symp. on Parallel and Distributed Processing with Applications (ISPA)*, pages 449 –457, September 2010.

[36] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *29th IEEE Symp. on Reliable Distributed Systems (SRDS)*, pages 214 –224, 31 2010-nov. 3 2010.