SPECULA: Speculative Replication of Software Transactional Memory

Sebastiano Peluso*†, João Fernandes*, Paolo Romano*, Francesco Quaglia† and Luís Rodrigues*
*INESC-ID/IST, Lisbon, Portugal †Sapienza, University of Rome, Italy

***Abstract*—This paper introduces SPECULA, a novel replication protocol for Software Transactional Memory systems that seeks maximum overlapping between the phases transaction processing and replica synchronization via speculative processing techniques.**

By removing the execution of the replica synchronization phase from the critical path of execution of transactions, SPECULA allows threads for pipelining the execution of speculatively executed transactional and/or non-transactional code. The core of SPECULA is a multi-version concurrency control algorithm that supports speculative transaction processing while ensuring the strong consistency criteria (analogous to opacity) that are desirable in non sand-boxed environments like STMs.

Via an extensive experimental study, based on a fully-fledged prototype and on both synthetic and standard STM benchmarks, we demonstrated that SPECULA can achieve speed-ups of up to one order of magnitude with respect to state-of-the-art non-speculative replication techniques.

I. Introduction

The advent of the multi-core era has fostered, over the last decade, an intense research on paradigms for parallel programming that can stand as simpler, more scalable and composable alternatives to conventional lock-based synchronization schemes. Encapsulating the complexity of concurrency control within the familiar abstraction of atomic transaction, Transactional Memory (TM) [1] have garnered significant interest both in the academic and industrial research community. Recently, the maturing of TM research has led to a hardware TM implementation for a commodity high-performance microprocessor and to the inclusion of TM support for the world's leading open source C/C++ compiler.

Distributed TMs (DTM) represent a natural evolution of TMs that aims at introducing a novel programming paradigm combining the simplicity of TMs with the scalability and failure resiliency achievable by exploiting the resource redundancy of distributed platforms. These features make the DTM model particularly attractive for inherently distributed application domains such as Cloud computing or High Performance Computing (HPC). In the HPC domain, several specialized programming languages (such as X10 [2] or Fortress [3]) already provide programmatic support for the DTM abstraction. In the context of Cloud computing, several recent NoSQL data-grids platforms [4], [5] expose transactional APIs and rely on in-memory storage of data to achieve higher performance and elasticity level.

In these in-memory platforms replication plays a role of paramount importance for fault-tolerance (as well as scalability) purposes, given that it represents the key means to ensure data durability in face of failures. On the other hand, when compared to other transactional systems such as classic DBMSs, in DTMs the cost of replication is particularly exacerbated. In TMs, there are in fact a number of factors that contribute to reducing the (local) execution time of transactions, such as the avoidance of costs due to persistent storage access or to parsing/optimization of SQL queries [6], or the possibility to rely on dedicated hardware supports [7] to minimize the costs associated with conflict detection and transactional logging.

As a direct consequence, state of the art transactional replication mechanisms designed for classic database systems have typically a dramatic impact on performance when employed in TM contexts. This phenomenon is clearly highlighted by the experimental data reported in Figure 1, which was gathered using a Java-based DTM platform [8] that integrates a state of the art database replication protocol [9], [10], often referred to as non-voting certification protocol. In this protocol transactions are executed locally in an optimistic fashion and consistency is ensured at commit-time, via a distributed certification phase that uses Atomic Broadcast (AB) to enforce agreement on a common transaction serialization order.

The DTM platform was evaluated using a standard benchmark for STM, namely STMBench7 [11], which was deployed on a private cluster of up to 8 nodes interconnected via a Gigabit Ethernet and equipped with 2 quad-core Xeon processors at 2.13GhZ and 8GB of RAM (which represents the reference experimental testbed used in the remainder of this paper). The plots show that, even in rich benchmarks like STMBench7, entailing complex, long-running transactions, the duration of the transaction commit phase (which entails the latency of the AB-based replica synchronization phase) accounts on average for *90%-95%* of the total transaction execution time.

In this paper we present SPECULA, a novel transactional replication protocol that exploits speculative techniques in order to achieve complete overlapping between replica synchronization and transaction processing activities. In SPECULA each transaction is executed only on a single node avoiding any form of synchronization till is enters its commit phase. Unlike conventional transactional replication protocols, in SPECULA the commit phase is executed in a non-blocking fashion: rather than waiting till the completion of the replica-wide synchronization phase, in SPECULA the results (i.e. the writeset) generated by a transaction that successfully passes a local validation phase are speculatively committed in a local multi-versioned STM, making them immediately visible to future transactions generated on the same node (either by the same or by a different thread).

By removing the execution of the replica synchronization phase from the critical path of execution of transactions, SPECULA allows threads for pipelining the execution of speculatively executed transactional and/or non-transactional

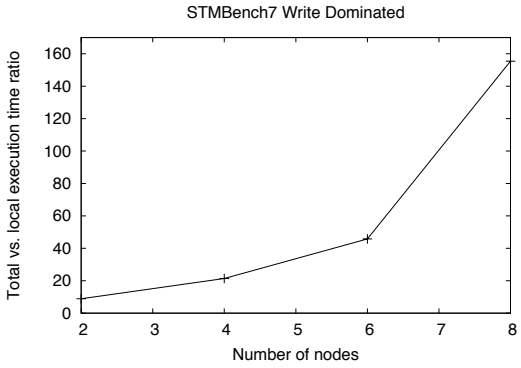


Fig. 1. Total vs. local execution time ratio in a replicated STM

code. If the outcome of the AB-based transaction certification scheme is positive, namely in absence of conflicts with concurrent remote transactions, SPECULA attains complete overlapping between processing and communication, with obvious benefits in terms of performance. In presence of misspeculations, SPECULA's run-time transparently detects data and flow dependencies, squashing any affected speculative execution in a completely transparent fashion. In order to support rolling-back multiple, speculatively-commuted transactions, as well as to revert the effects of speculatively executed non-transactional code, SPECULA relies on three main mechanisms: i) *continuations* [12], namely an abstraction that reifies the program control state that is used to save/restore the process' stack and point in computation; ii) a speculative multi-versioned STM (called *SVSTM*, Speculative Versioned STM) that avoids blocking on read/write operations and guarantees opacity and one copy serializability; iii) undo-logging techniques to restore the state of non-transactional heap.

We implemented a fully fledged prototype of the SPECULA in JAVA, and in order to achieve full transparency for applications we rely on automatic, class-loading time code instrumentation to inject code necessary to generate undo-logs for the update of objects residing in the non-transactional heap, as well as to orchestrate the usage of continuations. In addition, SPECULA's runtime automatically detects non-revertible operations (such as calls to native code), transparently injecting forced synchronization points that block speculation in order to guarantee the correct execution of such operations.

We evaluate SPECULA using both micro-benchmarks, as well as standard STM benchmarks. Our experimental study shows that SPECULA allows achieving up to one order of magnitude speed-ups when compared with a baseline non-speculative transactional replication protocol.

The remainder of this paper is structured as follows. Section II discusses related work while Section III introduces the system model. The SPECULA protocol is presented in Section V. Section VI reports the results of an experimental analysis aimed to quantify the performance of SPECULA.

II. Related Work

The issue of replicating transactional sites in order to achieve fault tolerance has been intensively studied in literature, especially for what concerns replication of database systems. The outcomes in this area include protocol specification (see, e.g., [13], [9]) middleware based replication architectures (see, e.g., [14], [10], [15]), as well as solutions requiring internal supports at the DBMS level (see, e.g., [16]). A key result confirmed by several works in this area (see, e.g., [17]) is that AB-based replication protocols, such as [9], can avoid the well-known scalability limitations [13] of classic two-phase commit [18] based replication protocols and ensure robust performance even in scenarios of non-minimal contention probability.

More recently, the advent of in-memory transactional data platforms (such as STMs) has raised renewed interest in the design of highly efficient transactional replication schemes. The solutions in [8], [19], [20] represent an example of replication protocols specifically targeted towards STM systems. These works introduced a number of optimizations aimed at minimizing the cost of AB-based replication protocols, such as bloom-filter based encoding schemes [8], lease abstractions permitting to minimize the usage of AB [19], and self-tuning mechanisms based on machine-learning techniques [20]. These optimizations are orthogonal to the key innovative idea introduced by SPECULA, namely preventing threads from blocking till the completion of the replica coordination phase. In fact, these optimizations could, in principle, be integrated with SPECULA, with the goal of maximizing the efficiency of the distributed transaction certification phase (even though we opted not to do so, in this work, for the sake of simplicity).

In some of our recent works [21], [22], [23], [24] we have introduced replication techniques that exploit the, so called, optimistic delivery order (an early, albeit potentially erroneous indication of the final total delivery order established by the AB algorithm) to overlap, in an optimistic fashion, the transaction execution with the (atomic broadcast-based) replica coordination phase with the purpose of minimizing the replication overhead. The approaches in [21], [22], [23] have targeted active replication, where each transactional site is requested to process each incoming update transaction, and investigated the idea of exploring multiple serialization orders for the same transaction in order to maximize robustness against mismatches between the optimistic and final delivery order. SCert [24], conversely, targets the scenario of certification-based (a.k.a. deferred update) [9] replication. Unlike active replication, this scheme, in which each transaction is executed at a single site and is only validated at remote sites, has been shown [10] to lead to a more effective resource exploitation in presence of write-intensive workload. Like SCERT, also the solution presented in this paper is built as an extension of the transaction certification replication protocol.

While these approaches and SPECULA share the common goal of minimizing replication overheads by overlapping processing and communication, they differ in one fundamental aspect from SPECULA. All the aforementioned protocols block a thread that has speculatively processed a transaction T, preventing it from executing further transactional or non-transactional code, until T's outcome has been finalized. SPECULA, on the other hand, optimistically assumes that locally valid transactions will also successfully pass the global validation phase (i.e. not incur in any conflict with remote transactions), and allow threads to speculatively pipeline a se-

quence of transactions possibly interleaved with the execution of non-transactional code. As shown in Figure 2, this allows SPECULA to achieve a way more effective overlapping between processing and communication, especially in scenarios where the ratio between the duration of these two phases is very low (e.g. in workloads with very short transactions). Also, those literature results only tailor speculation at the level of the transactional data-layer, while SPECULA takes a cross-layer approach in which both transactional and non-transactional code blocks can be (speculatively) alternated, thus entailing a synergic mix of state recoverability techniques at the levels of both the data-layer and the application layer.

Given that our proposal copes with the very general case where non-transactional (application level) code blocks can speculatively observe the output of not yet finalized transactions via interleaved executions along the same thread, our work has also relations with recent proposals in the context of partial rollback via checkpoints and continuations [12]. The base difference is that we exploit these notions within the context of a replicated transactional systems, thus combining them with distributed approaches for transaction management and validation at different sites.

As for the systematic use of speculative processing scheme, our work is also related to a set of results in the area of automatic speculative-parallelization in general purpose applications [25]. Here we consider the issue of fault-tolerant data replication, while those works mostly target performance and latency reduction via distributed executions on, e.g., commodity clusters.

III. System Model

We consider a classical distributed system composed by a set of STM processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate through a message passing interface and that can fail according to the fail-stop model [18]. The system is supposed asynchronous and enhanced with an unreliable failure detector encapsulating the synchrony assumptions required for implementing an *Atomic Broadcast* (AB) service.

AB is defined by the primitives *AB-broadcast(m)*, which is used to broadcast messages, and *AB-deliver(m)*, which delivers messages to the upper layer providing system-wide total order guarantees. In particular it ensures the following properties: **Validity:** If a correct process p broadcasts a message m, then p eventually delivers m; **Integrity:** No message is delivered more than once and if a process delivers a message m with sender p, then m was previously broadcast by process p; **Uniform Agreement:** If a message m is delivered by some process then m is eventually delivered by every correct process; **Total order.** Let m_1 and m_2 be any two messages and suppose p and q are any two correct processes that deliver m_1 and m_2. If p delivers m_1 before m_2 , then q delivers m_1 before m_2; **Causal Order:** If a process p delivers m and m' such that m causally precedes m', according to Lamport's causal order [26] (denoted $m \rightarrow m'$), then p delivers m before m'.

Each process in Π runs an instance of the SPECULA protocol and maintains a copy of a Software Transactional Memory (STM) [27], which allows application threads to combine sets of concurrent operations in a sequence of atomic *transactions*. As in typical STMs, we assume that threads can interleave the execution of transactional and non-transactional code.

IV. Correctness Criteria

Our global consistency criterion for SPECULA is 1-Copy Serializability (1CS) [28], which guarantees that the history of the the (finally) committed transactions executed by any process in Π is equivalent to a serial history executed on a single process.

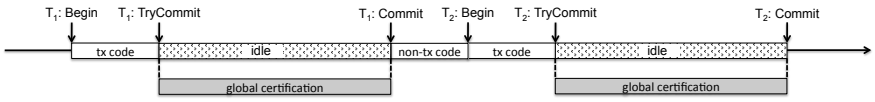
At the level of single (non-replicated), SPECULA guarantees a correctness criterion analogous to the well-known opacity criterion [29] (which, having been specified assuming a non-speculative transactional execution model, cannot be straightforwardly applied to SPECULA). Specifically SPECULA ensures that the snapshots observable by any transaction T is equivalent to that generated by a sequential history of transactions, independently from whether T is eventually aborted or (speculatively/finally) committed. This criterion prevents, just like opacity, any anomaly that may arise by observing system states that could never be generated by any serializable transaction history, and that could lead, in non-sand-boxed environments like STMs, to arbitrary behaviours, such as infinite loops or crashes.

V. The SPECULA protocol

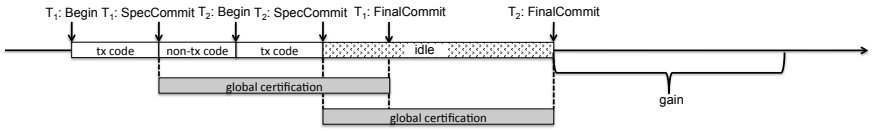
A. Protocol Overview

Analogously to classical certification-based replication schemes, e.g. [9], [10], in SPECULA transactions are processed locally on their origin nodes without relying on inter-replica synchronization facilities till they enter the commit phase. However, unlike conventional schemes, if a transaction requests a commit and it passes a local validation stage (that verifies the absence of conflicts with any concurrent transaction committed so far), it is locally committed in a *speculative* fashion and the global AB-based transaction certification stage is executed in parallel, in an asynchronous fashion. This allows a thread th that issued a commit for a transaction T to avoid blocking till the outcome of T's global certification is determined. Conversely, T's writeset is speculatively committed in the local STM, making it visible to subsequently starting local transactions, and th is allowed to execute immediately any subsequent non-transactional and transactional code block.

The above described speculative processing mechanism can lead to the development of two types of causal dependencies with respect to speculatively committed transactions: *speculative flow dependencies* and *speculative data dependencies*. A speculative flow dependency is developed whenever a thread executes code (either of transactional or non-transactional nature) after having previously speculatively (but not yet finally) committed some transaction. A speculative data dependency arises whenever the read operation of a transaction returns a value created by a speculatively committed transaction. In the following we say that a thread is executing speculatively if it has developed either a speculative data dependency or a speculative flow dependency. A speculative dependency from a speculatively transaction T is removed whenever the final outcome (commit/abort) for T is determined.



(a) Execution example using CERT



(b) Execution example using SPECULA

Fig. 2. Execution example of CERT and SPECULA

In presence of an abort event for a speculatively committed transaction, the above dependencies can lead to the propagation of a cascading abort of speculatively executed transactions, as well as require the restoration of non-transactional state (i.e. heap, stack, and the thread control state) updated during the speculative execution of non-transactional code. On the other hand, in case speculatively committed transactions are compatible with the final serialization order established by the AB-based certification scheme, SPECULA achieves an effective overlapping between the processing and distributed transaction certification phases. The diagram in Figure 2 illustrates the advantages achievable by SPECULA with respect to a conventional certification-based replication protocol, hereafter named CERT, via an example execution of a sequence of two transactions T_1 and T_2, interleaved with a non-transactional code block. In this example we are assuming that both T_1 and T_2 complete their execution without incurring in aborts due to conflicts with other transactions in the system. Both with CERT and SPECULA, T_1 is executed locally and, when it enters the commit phase, a *global certification* phase is started. At this point the two schemes diverge. With CERT, the application level thread is blocked until T_1's global certification finishes. With SPECULA, instead, T_1 is speculatively committed and the application level thread can immediately start processing the subsequent non-transactional code and execute transaction T_2. By pipelining the execution of transactional and non-transactional code blocks, and overlapping it with the transaction certification phase, SPECULA achieves, in the above example, a reduction of the overall execution time, in the considered example, equal to the duration of the T_1's global certification.

B. High level software architecture

The diagram in Figure 3 provides a high level overview of the software architecture of a SPECULA replica. A concurrent application, i.e. *TM Application* layer, starts a transaction by triggering a *Begin* event on the *Speculative Versioned Software Transactional Memory*, hereafter referred as *SVSTM*, which registers the transaction in the *SPECULA* environment and handles every subsequent *Read* and *Write* operation executed within the context of that transaction.

The *SVSTM* layer provides isolation guarantees during transactions' execution and ensures the atomicity of the state alterations associated with abort, speculatively commit and final commit events. These properties are guaranteed by a

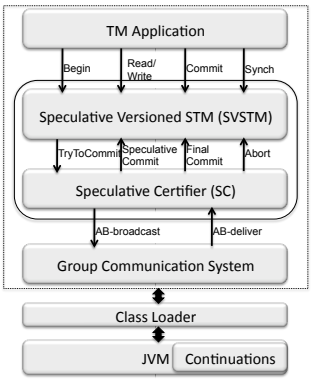


Fig. 3. Software architecture of a SPECULA instance

multi-versioning concurrency control scheme that ensures that read operations performed by a transaction T can be always executed in a non-blocking manner by properly selecting versions belonging to the most recent consistent snapshot already committed before the beginning of T. Further, unlike classical multi-versioning schemes, *SVSTM* allows a speculatively committed transaction T to:

1) externalize T's writeset to other locally executing transactions before T's final outcome is established by its AB-based certification phase.
2) support the atomic removal of T's writeset, in case the certification phase of T determines that it needs to be aborted. In this case, the SVSTM supports also the cascading abort of transactions having speculative (data and/or flow) dependencies from T.

Additionally, with SPECULA we need to cope with scenarios in which a non-revocable operation (e.g. an I/O operation) is issued by a thread that is executing in speculative mode, i.e. that depends causally on a speculatively committed state. To tackle this case, *SVSTM* offers a simple programming interface, called *Synch*, that can be used by user-level applications to force a synchronization point and blocks speculation on the caller thread until all the transactions speculatively committed so far by that thread are final committed, or at least one of them is aborted (in which case it is needed to squash this speculative execution).

When the application layer requests a commit (*Commit* event) for a transaction T, *SVSTM* triggers the validation of

T's execution to the *Speculative Certifier*, hereafter referred as *SC*, which tries to commit T in two steps. If, in the first step, *SC* successfully speculative validates T then it triggers a *Speculative Commit* event to the upper layer and it enters the second step; otherwise an *Abort* event is triggered, because T cannot be speculatively serialized correctly. In the second validation step SC tries to globally certify T's execution by relying on the total ordering service provided by the *GCS* layer. Depending on whether or not T can be correctly serialized with the final committed transactions in the cluster, a *Final Commit* or an *Abort* event is triggered to the upper layer.

In order to safely support the rollback of chains of speculatively committed transactions (possibly interleaved by non-transactional code), SPECULA relies on the *continuations* abstraction, and on the automatic transactification of non-transactional code, via undo-logging techniques. A continuation reifies the control state of a computational thread control state, and allows resuming the execution at the point in which it was created. To ensure total transparency for the user level applications, SPECULA relies on a customized class loader that instruments automatically the application code (at the bytecode level) in order to automatically capture continuations out of the transactions' boundaries and transparently create all the needed data structures used for reversing non-transactional executions.

C. Speculative execution of transactions

For space constraints we have to report the pseudo-code formalizing the behavior of the algorithm used to manage the speculative transaction processing in appendix. Nevertheless, in the following we describe in detail the key mechanisms at the basis of the SVSTM, providing several insights and arguments on its correctness..

Management of speculative/final snapshots. As in classical multi-versioned STMs, e.g. [30], also *SVSTM* maintains multiple committed versions of data, and uses a timestamp based mechanism to associate transactions with a snapshot that is used during their execution to determine version visibility. Unlike existing multi-version concurrency control algorithms, however, in SVSTM we need to allow the addition and removal of speculatively committed versions. These manipulations of the object versions need to take place without endangering the correctness (i.e. serializability) of the snapshots observed by currently executing transactions.

To this end, in SPECULA each transactional object is wrapped in a, so called, *Transactional Container* (*TC*), an abstraction that provides two main functionalities: i) allowing cluster-wide object identification, by transparently associating unique IDs to transactional objects, and ii) maintaining and managing (speculative/final) committed versions of a transactional object. The speculative and final versions maintained by the *TC* are organized in two separate single-linked lists. As shown in Figure 4(a)), the *TC* maintains the pointers to the most recent speculatively and finally committed versions of an object, denoted, respectively, as *lastSpec* and *lastFinal*. A (speculatively/finally) committed version created by a transaction T stores (i) a reference to the previously (speculatively/final) committed version in the list (if any), (ii) the associated value, (iii) a scalar timestamp, called version number, vn identifying the (speculatively/finally) committed snapshot generated by T, (iv) a reference to a *Transaction* object identifying T.

The most recent final committed snapshot is tracked by the SVSTM using a single scalar timestamp, which we call *UpperFinID*. As in typical conventional (i.e. non-speculative) multi-versioned STMs, whenever a transaction T is final committed the *UpperFinID* is incremented and the new versions created by T are written back tagging them with the current value of *UpperFinID*.

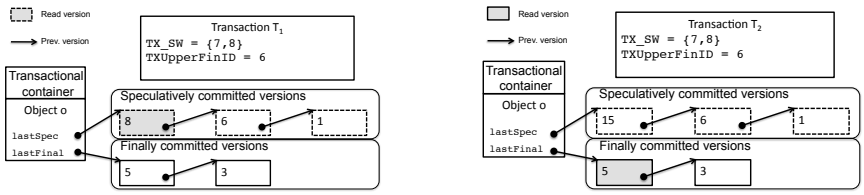
The management of speculatively committed snapshots is instead regulated via a, so called, *Speculative Window* (SW), which is a linked list maintaining a node for each speculatively committed transaction by a given SPECULA replica.

The advancement of speculative commits is regulated by a second timestamp, called *UpperSpecID*, which identifies the most recent transaction in the speculative window (i.e. the last transaction that has been speculatively committed). Whenever a transaction is speculatively committed, *UpperSpecID* is incremented and, for each data item in its writeset, a new speculative version is added in its *TC* and tagged with the new value of *UpperSpecID*. Further a node is added to the head of the SW, causing its widening. The speculative window is instead narrowed, whenever a final outcome (commit/abort) is determined for one of the transactions in SW, causing the removal of the corresponding node.

As we will see, the most recent speculatively committed snapshot of a SVSTM's replica is equivalent to the one obtained by serializing the speculatively committed (and not aborted) transactions contained in the speculative window after the entire sequence of finally committed transactions up to the one having (final committed) snapshot id equal to *UpperFinID*.

Transaction Activation. Let us analyze the data structures associated with a transaction and how they are initialized upon the begin of transaction: i) the transaction's readset and writeset; ii) a $state$ variable that can assume values in the domain $\{executing,\ aborted,\ speculatively\ committed,\ finally\ committed\}$; iii) a scalar timestamp, called $TxUpperFinID$, which determines the most recent final committed snapshot visible by that transaction and is set, upon the beginning of the transaction, to the current value of the SVSTM's $UpperFinID$; iv) a linked list, called TX_SW, that is initialized, at transaction's activation time, by creating a copy of the SVSTM's SW; v) a reference to a continuation, called $resumePoint$, which is initialized to `null`, and is updated upon the speculatively commit of the transaction to refer to the thread's execution context at the end of the execution of this transaction. vi) an $undo - log$ that, as it will be discussed in Section V-D, is used to rollback any update of the non-transactional heap performed by the non-transactional code blocks executed after the speculative commit of the transaction.

Tracking speculative flow dependencies In order to track flow dependencies among speculative transactions executed by the same thread an additional timestamping mechanism is



(a) Read of a speculative version (b) Read of a final version

Fig. 4. Versioning in *SVSTM* and examples of *read* operations

used. Each thread th is associated with a scalar timestamp called $epoch$, which is initialized to zero and incremented upon the abort of a transaction speculatively committed by th. Further, each replica maintains a table, called $EpochTable$ that stores the current epoch of each thread issuing transactions in any of the processes in Π. As it will become clearer in the following, this simple mechanism allows detecting whether an AB-delivered commit request (possibly associated with a remotely originated transaction) should be discarded due to a speculative flow dependency from an aborted speculative transaction.

Read and write operations. Write operations are managed in a very simple manner: the new written value is buffered in a transactional local collection, i.e. the transaction's writeset. As we will see, the post-images created by a transaction T are only made visible to other transactions upon T's (speculative or final) commit.

Let us now analyze how it is managed a read operation issued by a transaction T on object o. If T has already written on the object for which it is issuing a read operation, it has to observe the value that it previously wrote. Thus, it if first necessary to check whether o is already in T's writeset and, in the positive case, return the o's value stored there.

Otherwise, it is first checked if o's speculative versions' chain contains any version speculatively committed by a transaction present in T's speculative window. If this is true, the most recent of such versions, say v^* is the one that should be observed by T, based on the speculative serialization order that was attributed to it upon its start. Before returning v^* it is however first necessary to check whether the transaction that created v^* has been aborted in the meanwhile (recall each version stores a pointer to the creating version). In such a case T needs to be aborted as well, as it is attempting to read an invalid speculative snapshot.

Finally, if no visible speculative version of o exists, the visibility rule of conventional multi-version concurrency control algorithms is used, namely it is returned the most recent final committed version of o having a timestamp less or equal than *UpperFinID*.

To better clarify the read mechanism we show in Figure 4(a) two example executions in which a read operation is issued by a transaction T having $TXUpperFinID$ equal to 6 and having in its SW two speculatively committed transactions with timestamps 7 and 8. In the example on the left side, T observes the speculative version having timestamp 8, since transaction 8 is in T's SW. In the example to the right, none of the available speculatively committed versions is observable by T, which is forced to read the most recent finally committed version having timestamp 5.

Commit requests. The above described visibility rules ensure that a transaction T observes an opaque snapshot [29] equivalent to the one generated by the sequential history that includes the (globally validated) final committed transactions up to T's *TXUpperFinID* followed by the speculatively committed transactions in T's *TX_SW*. On the other hand, in SPECULA read-only transactions may have to be aborted due to the detection of speculative flow/data dependencies, possibly well after they have completed their execution. The serialization order of the speculatively committed local transactions from which a read-only transaction has a speculative dependency may in be fact be found to be incompatible with the one determined by the AB-based certification mechanism.

As the speculative snapshot observed by a transaction is opaque, SPECULA can spare read-only transactions from the cost of a dedicated (and very expensive) AB-based validation phase. Conversely, SPECULA adopts a lazy validation mechanism that delegates the validation of read-only transactions (that have observed some speculatively committed value) to the first update transactions subsequently executed by the same thread[1].

By this mechanism, whenever a thread th requests the commit for an update transaction T_{up}, it does not only validate T_{up} but also any speculative read only transaction T_{ro} that th executed before T_{up} and after the last update transaction preceding T_up. Let us denote this set of read-only transactions using the notation $RO(T_{up})$.

More in detail, as an update transaction T_{up} requests to commit, it is first locally validated to check for conflicts with already (both speculatively and final) committed transactions. This validation simply entails iterating over T_{up}'s readset and checking whether T_{up} would still observe the same versions if its execution were to start in this moment.

If the local validation phase is successful, the commit request of T_{up} is sent via AB, along with the following information: the identifiers of the objects in T_{up}'s readset, its writeset, its speculative window and the identifiers of the set of speculative transactions from which and $RO(T_{up})$ have developed a read-from dependency. We denote this latter sets of transactions, as $SRF(RO(T_{up}))$. Finally, in order to allow remote nodes to track speculative flow dependencies, it is sent

[1]Unless, as we will see, the maximum bound on the number of speculatively committed transaction or the end of the thread's code is reached. In which case the thread is forced to block until the speculative snapshots observed by the read-only transactions has been either finally committed or aborted.

via AB also the unique identifier of the thread executing T_{up} and its current $epoch$ value.

While the commit request of AB is being delivered, in a non-blocking fashion, T_{up} is speculatively committed, allowing the thread that has executed T_{up} to progress.

Speculative commit. The speculative commit of a transaction T by a thread th implies two main operations. First, a checkpoint of the current's thread execution context is generated by creating a continuation and is stored in T's $resumePoint$ variable. This continuation will be used in case it is later detected that the speculative transaction executed by th after T needs to be rolled back.

If T is an update transaction, it is also necessary to write-back the speculative versions that T updated/created. This is done by atomically i) increasing $UpperSpecID$, ii) adding the new speculatively committed versions of the objects in T's writeset at the head of the speculative chain version of the corresponding *TCs* and (iii) adding a new node associated with T to the head of the SVSTM's SW.

Global validation and final commit. When at node n_i an AB is delivered requesting the commit of a transaction T_{up} originated by thread th running on node n_j, T_{up} is validated to detect whether it is possible to final commit it.

First, it is checked in which epoch T_up has been originated by th. If the T_up's epoch is less than the current value stored in $EpochTable$ for th, it means that T_up has a speculative flow dependency from an aborted transaction and can be discarded. If T_up is tagged with a larger epoch value than the one currently associated with th at node n_i, say e', the corresponding entry in $EpochTable$ is updated.

It is then checked whether the read-only transactions $RO(T_{up})$ have read a valid snapshot. To this end it is checked whether the transactions in $SRF(RO(T_{up}))$ have been, in the meanwhile, final committed at n_i. This is achieved by looking up a table, called $Spec2FinIDs$, that maintains a mapping between the identifier of each transaction T that is both speculatively committed and final committed, and the *UpperFinID* snapshot that was attributed to T when it was final committed.

By the causal ordering property of the AB channel, at the time in which T_{up} is final delivered, it follows that also the update transactions $SRF(RO(T_{up}))$ from which T_{up} (via $RO(T_{up})$ depends indirectly must have already been final delivered. Therefore, if any transaction T^* in $SRF(RO(T_{up}))$ cannot be found in $Spec2FinIDs$, it means that T^* must have been already aborted at n_i.

Before final committing a read-only transaction $\tilde{\mathrm{T}}$ in $RO(T_{up})$, the following additional check is performed. The maximum final commit timestamp, denoted as $maxFinID$, is computed across all the speculatively committed transactions from which $\tilde{\mathrm{T}}$ depends (namely, $SRF(\tilde{\mathrm{T}})$). Then, for each object in $\tilde{\mathrm{T}}$'s readset, it is determined which was the most recently finally committed version at the time in which the SVSTM had $UpperFinID$ equal to $maxFinID$. If the final commit timestamp of any of these versions is larger than $\tilde{\mathrm{T}}$'s $TXUpperFinID$, and the creating transaction, say $\hat{\mathrm{T}}$, is not present in $\tilde{\mathrm{T}}$'s SW, then it means that $\hat{\mathrm{T}}$ has been serialized before $\tilde{\mathrm{T}}$ in the final delivery order, and that $\tilde{\mathrm{T}}$ has missed $\hat{\mathrm{T}}$'s snapshot. In order to guarantee 1CS, $\tilde{\mathrm{T}}$ needs to be aborted. More precisely, this control is aimed at ensure the so-called snapshot monotonicity property [31], i.e. to avoid scenarios in which two read-only transactions executing at different replicas can serialize in different (and mutually incompatible) orders the final commit events of non-conflicting update transactions.

Clearly, if any of the transactions in $RO(T_{up})$ is aborted, also T_up has to be aborted. Next T_{up}'s readset is validated. To this end it is first verified whether, for each object o in T_{up}'s readset, o's most recent final committed version, say $lastFinal(o)$, has version number less or equal than T_up's *TxUpperFinID*. This is certainly a safe read, given that T_{up} has read a non-speculative version of o, which has not been in the meanwhile overwritten by more recent versions, and is hence valid. If this is not the case, it is checked whether the transaction that created $lastFinal(o)$, say $T^{\dagger}$, is in the speculative window of T_up. Also in this case the $Spec2FinIDs$ table is used to determine a mapping between the identifiers of (possibly remotely originated) speculatively committed transactions and their final commit timestamp (if any). If this is false, it means that T_up needs to be aborted as it has read a speculative version that has been either aborted, or overwritten by a most recent (final committed) version.

If T_{up} is a remote transaction, an additional test is performed before applying its writeset, in order to determine whether T_{up}'s commit invalidates any local speculatively committed transaction. To this end it is checked, for each transaction T' in the node's SVSTM whether its readset intersects with the writeset of T_{up}. In this case, T' is aborted, triggering the cascading abort of any other speculative transaction having speculative flow/data dependencies with T'.

At this point the node's $UpperFinID$ is increased, and the writeset of T_{up} is applied by creating new finally committed version in the corresponding *TCs* and tagging them with the new $UpperFinID$ value.

Finally, if T_{up} had been previously speculatively committed on node n_j (i.e. $n_i = n_j$), it is removed from the SVSTM's SW

Abort. If a transaction T is aborted while it is still executing, its state is simply marked as $Aborted$. The state of a transaction is checked by the SVSTM's layer any time that a read/write/commit operation is issued for that transaction, which guarantees that T will be aborted before it can speculatively commit and externalize its writeset.

Additional care needs to be taken, instead, if the transactions T that is being aborted has already been speculatively committed. In this case, in fact, there may be locally running transactions that include T in their speculative windows, and which may access some of the objects for which T has produced a speculative version while T's abort procedure is being concurrently executed. In order to avoid anomalies, it is also required that the abort of T is performed atomically with the abort of any transaction in SW that has (possibly transitive) speculative dependencies from T (otherwise, concurrently executing transactions may miss the snapshot of T and observe the snapshot created by transactions that

depend on T). Let us denote with th the thread that executed a transaction T that is being aborted. The abort procedure recursively aborts all the transactions that were speculatively committed by th after T in reverse chronological order (from the most recent to the the oldest). Next, the abort procedure is called on any other transaction in SW that has read from T. Finally, the state of T is set to $aborted$, which allows to prevent in an atomic way that future reads can observe any of T's speculatively committed version.

Garbage Collection In order to allow the garbage collection of speculative and final committed versions, whenever a transaction is final committed or aborted SPECULA evaluates what are the oldest final and speculative snapshots visible by any locally running transaction. These snapshots are equal to the minimum value, across all the active transactions T, of T's $TXUpperFinID$, and, respectively, of the minimum speculative snapshot identifier in T's TX_SW.

D. Speculative execution of non-transactional code

In case of misspeculation SPECULA may have to revert the execution of a sequence of transactional and non-transactional code blocks. As already discussed, continuations are used to allow resuming the execution flow at the end of the most recent valid transaction committed by each thread.

In order to rollback the alterations performed on the heap by non-transactional code SPECULA relies on undo logs. Undo logs are built in a totally transparent fashion for the application by intercepting 8 different JAVA bytecode instructions issuing modifications of fields and arrays. Automatic bytecode instrumentation is achieved by replacing the JAVA default class-loader with a custom class-loader that performs dynamic bytecode rewriting in the moment in which any JAVA class is loaded by the JVM.

When anyone of these bytecode instructions is executed by a thread th, the pre-image of the target address is saved in the undo-log before being overwritten with the new value. An undo log keeps just one value per memory address, namely the oldest one, and is associated with the last speculatively committed transaction T that was executed by th. If later, the squashing of a speculative execution causes the abort of th, the undo log is used to restore the heap state to the snapshot at the moment in which the continuation that is going to be resumed was captured (that is at the end of transaction T).

Another issue that is tackled by SPECULA's automatic bytecode instrumentation framework is the transparent addition of forced synchronization blocks preceding any non-revocable operation (e.g. I/O operations). More in detail, SPECULA leverages on the JaSPEx [32] library to detect, at the bytecode level, calls to native code via JNI and other not revertible operations (e.g. invocations of java.io.* package), and automatically insert bytecode that forces the thread to wait for the validation of any previously speculatively committed transaction.

E. Correctness Arguments

In this section we provide some informal arguments on the correctness of SPECULA with respect to the criteria mentioned in Section IV.

The proof of 1CS stems directly from the fact that every final committed transaction, is validated deterministically and in the same final order by all the replicas in the system. Specifically, if a transaction T_i has developed any speculative dependency from a speculatively committed transaction T_j, SPECULA validates T_i committed only after having final committed, or aborted, T_j. In the latter case, T_i will be aborted either when T_j is delivered, or during the validation of T_i that takes place when the commit request of T_i is AB-delivered. Now assume that all the transactions T_j from which T_i has developed a speculative dependency have been committed at the time in which T_i is validated. In this case, the validation procedure of T_i detects whether T_i would observe the same snapshot if it were executed using a non-speculative multi-versioning scheme starting on a committed snapshot that includes all the transactions committed before T_i according to the AB-based serialization order.

Let us now analyze how SPECULA ensures the serializability of the snapshots observed by transactions (including those that are eventually aborted) throughout their execution. When a transaction T is activated at a node n_i, SPECULA speculatively serializes T after the totally ordered history of (update) transactions, say H, composed by i) the prefix of final committed transactions, followed by ii) the sequence of transactions speculatively committed ordered according to n_i's SW. The latter ones have been locally validated before being speculatively committed. This guarantees the serializability of H. During the execution of T, new transactions may commit, either speculatively or finally. In both cases, however, the new versions created by these transactions cannot be observed by T due the visibility conditions regulating the execution of read operations. Also, the abort of any of the transactions in T's TX_SW, say T^* does not compromise the serializability of the snapshot observable by T. In fact, the versions speculatively committed by T^* are only removed by the underlying SVSTM after T's execution is completed (see the "Garbage collection" paragraph in Section V-C.

VI. Evaluation

In order to assess the performance gains achievable by SPECULA we use as baseline for our experimental analysis a non-speculative multi-versioned replicated STM that, analogously to SPECULA, relies on an AB-based distributed certification phase taking place whenever an update transactions enters it commit phase [9],[10]. We use as local STM for the baseline JVSTM [30], a state of the art multi-versioned STM that never block or aborts read-only transactions. The AB layer is based on the LCR [33] algorithm, a recent ring-based AB algorithm that is known for its robust performance especially at high throughput.

In addition to the mechanisms described in Section V, in our SPECULA prototype we introduced a simple throttling mechanism that limits the maximum number of speculatively committed transactions pending at any node. By treating the speculativity level as an independent parameter, our study aims to assess the impact of SPECULA's speculative processing technique on various performance indicators, and in particular latency of the AB layer and transactional abort rate.

The first considered workload is a synthetic benchmark, called Bank, that was selected to assess the maximum gains

achievable by SPECULA: bank contains exclusively update transactions that simulate the transfers among bank deposits, and was configured not to generate any conflict. These are clearly ideal conditions for SPECULA, and the plots in Figure 5 confirm the achievement of striking performance gains, with speed-ups varying in the range 4x (for 8 nodes) to 13x (for 2 nodes). Despite the simplicity of this scenario, these data allow us to draw some interesting conclusions. The top plot in Figure 5 shows that increasing the speculativity level beyond 8 does not provide any additional speed-up for SPECULA. This is explainable by observing in the bottom plot that the AB-delivery latency grows very rapidly for speculativity levels above 8, highlighting that SPECULA has already hit the maximum throughput level achievable by the underlying AB implementation.

The second considered workload is the write-dominated configuration of STMBench7 [11]. STMBench7 is a complex benchmark that features a number of operations with different levels of complexity over an object-graph with millions of objects. In the write-dominated configuration, this benchmark generates around 50% of update transactions and yields, with the baseline replication protocol, a moderate contention level ranging from 5% (for 2 nodes) to around 15% (for 8 nodes). Also in this scenario, see Figure 6 SPECULA achieves remarkable speed-ups, up to a 4.4x factor in the configuration of 6 nodes and speculativity level equal to 8. Interestingly, unlike in the previously analyzed scenario, with STMBench7, an excessively high setting for the speculativity level can actually hinder SPECULA's performance. This is justifiable observing the trends of the abort rate and AB-delivery latency in the two bottom plots of Figure 6. For speculativity levels lower than 8, the AB-delivery (in log scale) remains comparable with that of the baseline, and the abort rate is even lower when using SPECULA. We argue that this decrease of the transaction abort probability is imputable to the fact that the burst of speculatively committed transactions by each replica has a high chance of being final delivered without the interleaving of messages associated with remote transactions, at least at low/moderate levels of load for the AB layer. As the speculativity level increases, on the other hand, the GCS load accordingly increase, and the chances that the burst of transactions speculatively committed by a node are final delivered without interleaving commit requests for remote transactions decrease drastically. The result is a rapid growth of the abort rate, especially as the number of nodes, and hence the concurrency in the system, grow.

Finally, in Figure 7, we analyze the read-dominated workload of STMBench7, a setting in which this benchmark generates 94% of read-only transactions. This is clearly a least favorable scenario for SPECULA, as, on average, 94% of the incoming transactions can be executed very efficiently by the baseline algorithm, i.e. locally and without the risk of incurring in aborts or blocks. Also the speculative transaction processing mechanism at the core of SPECULA is triggered only for 6% of the transactions. Nevertheless, even in this unfavourable scenario, SPECULA achieves comparable, or even higher (up to a two-fold speed-up) throughput in almost all the evaluated scales of the platform. The only exception is the case of 2 nodes where the baseline can benefit from a significantly lower

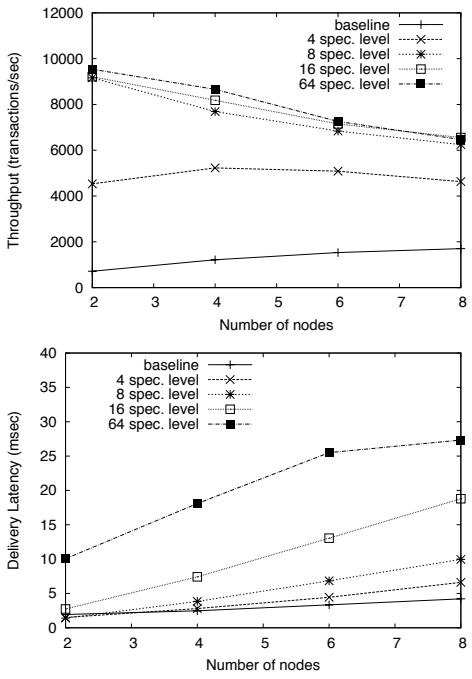


Fig. 5. Bank Benchmark

AB-delivery latency. For space constraints we do not report the abort rates for this scenario, but they remain below 5% both for the baseline and SPECULA in all the considered settings. Analogously to the bank benchmark's scenario, also in this case increasing the speculativity level beyond 4 does not pay off in SPECULA. In fact, as two update transactions are on average interleaved by a relatively high number of read-only transactions (that do not incur any replica synchronization phase in the baseline) the gains achievable in SPECULA by overlapping the phases of processing and replica coordination (for update transactions) are quite reduced.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proc. ISCA*. ACM, 1993.

[2] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, "X10: an object-oriented approach to non-uniform cluster computing," in *Proc. of OOPSLA*. ACM, 2005, pp. 519–538.

[3] E. Allen, D. Chase, J. Hallett, V. Luchangco, J.-W. Maessen, S. Ryu, G. L. S. Jr., and S. Tobin-Hochstadt, "The Fortress Language Specification," Sun Microsystems, Inc., Tech. Rep., 2007.

[4] A. Seovic, M. Falco, and P. Peralta, *Oracle Coherence 3.5*, 1st ed. Packt Publishing, 2010.

[5] "Red Hat / JBoss Infinispan." [Online]. Available: http://www.jboss.org/infinispan

[6] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proc. LADIS*. ACM, 2008, pp. 4:1–4:4.

[7] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *Proc. ASPLOS*. ACM, 2006, pp. 336–346.

[8] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2stm: Dependable distributed software transactional memory," *PRDC*, pp. 307–313, 2009.

[9] F. Pedone, R. Guerraoui, and A. Schiper, "The database state machine approach," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 71–98, 2003.

[10] M. Patiño Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso, "Scalable replication in database clusters," in *Proc. DISC '00*. Springer-Verlag, 2000, pp. 315–329.

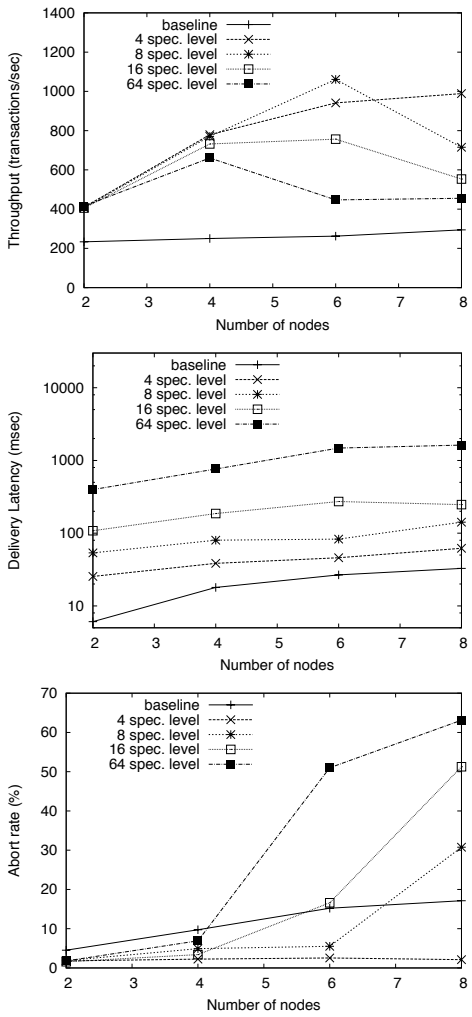


Fig. 6. STMBench7 Write Dominated

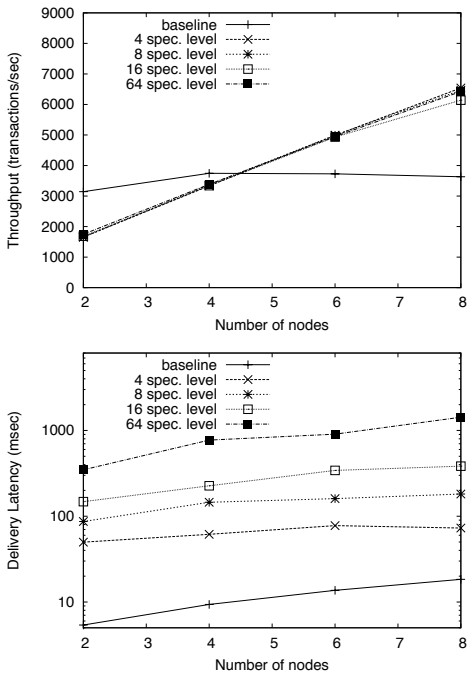


Fig. 7. STMBench7 Read Dominated

[11] R. Guerraoui, M. Kapalka, and J. Vitek, "Stmbench7: a benchmark for software transactional memory," in *Proc. EuroSys*. ACM, 2007, pp. 315–324.

[12] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *Proc. SPAA*. ACM, 2008, pp. 160–168.

[13] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proc. SIGMOD*. ACM, 1996, pp. 173–182.

[14] Y. Lin, B. Kemme, M. Patiño Martínez, and R. Jiménez-Peris, "Middleware based data replication providing snapshot isolation," in *Proc. SIGMOD*. ACM, 2005.

[15] F. Pedone and S. Frølund, "Pronto: High availability for standard off-the-shelf databases," *J. Parallel Distrib. Comput.*, vol. 68, no. 2, pp. 150–164, Feb. 2008.

[16] B. Kemme and G. Alonso, "Don't be lazy, be consistent: Postgres-r, a new way to implement database replication," in *Proc. VLDB*. Morgan Kaufmann Publishers Inc., 2000.

[17] F. Pedone and A. Schiper, "Optimistic atomic broadcast: a pragmatic viewpoint," *Theor. Comput. Sci.*, vol. 291, no. 1, pp. 79–101, 2003.

[18] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to Reliable and Secure Distributed Programming (2. ed.)*. Springer, 2011.

[19] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, ser. Middleware '10. Springer-Verlag, 2010, pp. 376–396.

[20] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: Polymorphic self-optimizing replication for in-memory transactional grids," in *Middleware 2011*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011, vol. 7049, pp. 309–328.

[21] R. Palmieri, F. Quaglia, and P. Romano, "AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing," in *Proc. of NCA*. IEEE Computer Society, 2010, pp. 20–27.

[22] R. Palmieri, F. Quaglia, and Romano, "OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems," in *Proc. of SRDS*. IEEE Computer Society, 2011.

[23] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, "Brief announcement: on speculative replication of transactional systems," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, ser. SPAA '10. ACM, 2010, pp. 69–71.

[24] N. Carvalho, P. Romano, and L. Rodrigues, "Scert: Speculative certification in replicated software transactional memories," in *Proceedings of the 4th Annual International Conference on Systems and Storage*, ser. SYSTOR '11. ACM, 2011, pp. 10:1–10:13.

[25] H. Kim, A. Raman, F. Liu, J. W. Lee, and D. I. August, "Scalable speculative parallelization on commodity clusters," in *Proc. ISCA*. IEEE Computer Society, 2010, pp. 3–14.

[26] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[27] N. Shavit and D. Touitou, "Software transactional memory," in *Proc. PODC*. ACM, 1995, pp. 204–213.

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

[29] R. Guerraoui and M. Kapalka, "On the correctness of transactional memory," in *Proc. PPoPP*. ACM, 2008, pp. 175–184.

[30] J. a. Cachopo and A. Rito-Silva, "Versioned boxes as the basis for memory transactions," *Sci. Comput. Program.*, vol. 63, no. 2, pp. 172–185, Dec. 2006.

[31] A. Adya, "Weak Consistency: A Generalized Theory and Optimistic Implementations for Distributed Transactions," PhD Thesis, Massachusetts Institute of Technology, Tech. Rep., 1999.

[32] I. F. S. D. Anjo and J. Cachopo, "Jaspex: Speculative parallel execution of java applications," *1 INFORUM*, 2006.

[33] R. Guerraoui, R. R. Levy, B. Pochon, and V. Quéma, "Throughput optimal total order broadcast for cluster environments," *ACM Trans. Comput. Syst.*, vol. 28, no. 2, pp. 5:1–5:32, 2010.

APPENDIX

Algorithm A.1 Transaction activation

```
1: void Begin(Transaction tx, ThreadContext tc)
2:     tx.state← executing;
3:     tx.TxUpperFinID← UpperFinID;
4:     tx.TX_SW← copy(SW);
5:
```

Algorithm A.2 Read and Write operations

```
1: void Write(Transaction tx, Object o, Value val)
2:     if tx.state = aborted then
3:         throw ABORT;
4:     end if
5:     put(tx.writeSet, o, val);
6:
7: Value Read(Transaction tx, Object o)
8:     if tx.state = aborted then
9:         throw ABORT;
10:    end if
11:    Value val←get(tx.writeSet, o);
12:    if val ≠ null then
13:        return val;
14:    end if
15:    Version v*←mostRececentInSpecWindow(o.lastSpec,
   tx.TX_SW);
16:    if v* ≠ null then
17:        if v*.commitTx.state = aborted then
18:            throw ABORT;
19:        end if
20:    else
21:        v*←mostRecentLessThan(o.lastFinal,
   tx.TxUpperFinID);
22:    end if
23:    return v*.value;
24:
```

Algorithm A.3 Commit request

```
1: void Commit(Transaction tx)
2:     if tx.state = aborted then
3:         throw ABORT;
4:     end if
5:     if validateLocal(tx) then
6:         if !tx.isReadOnly() then
7:             AB-broadcast(tx);
8:         end if
9:         specCommit(tx);
10:    else
11:        throw ABORT;
12:    end if
13:
```

Algorithm A.4 Delivering a commit broadcast message on node n_i

```
1: AB-Deliver(Transaction tx)
2:     Integer epoch←get(EpochTable, tx.getThread.getID);
3:     if tx.getThread.getEpoch() < epoch then
4:         tx.state ← aborted;
5:         if tx.isLocal() then
6:             tx.abort();
7:         end if
8:         return ;
9:     end if
10:    for all stx ∈ SRF(RO(tx)) do
11:        if stx ∉ Spec2FinIDs then
12:            tx.state ← aborted;
13:            if tx.isLocal() then
14:                tx.abort();
15:            end if
16:            return ;
17:        end if
18:    end for
19:    for all o ∈ tx.readSet do
20:        if o.lastFinal.version.vn > tx.TxUpperFinID ∧
   ¬contained(o.lastFinal.version.commitTx, Tx.TX_SW)
   then State tx.state ← aborted;
21:            if tx.isLocal() then
22:                tx.abort();
23:            end if
24:            return ;
25:        end if
26:    end for
27:    if tx.isRemote() then
28:        for all stx ∈ localSpecTxs do
29:            if (stx.readSet ∩ tx.writeSet) ≠ ∅ then
30:                stx.state← aborted;
31:                stx.abort();
32:            end if
33:        end for
34:    end if
35:    finalCommit(tx);
36:
```

Algorithm A.5 Local validation

```
1: boolean validateLocal(Transaction tx)
2:     if !tx.isReadOnly() then
3:         for all o ∈ tx.getReadSet do
4:             if o.value!=o.lastFinal.value ∧
   o.value!=o.lastSpec.value then
5:                 return false;
6:             end if
7:         end for
8:     end if
9:     return true;
10:
```