SPECULA: Speculative Replication of Software Transactional Memory

Sebastiano Peluso*†, João Fernandes*, Paolo Romano*, Francesco Quaglia† and Luís Rodrigues*
*INESC-ID, Instituto Superior Técnico, Universidade Técnica de Lisboa, Portugal
†DIS, Sapienza, University of Rome, Italy

***Abstract*—This paper introduces SPECULA, a novel replication protocol for Software Transactional Memory (STM) systems that seeks maximum overlap between transaction execution and replica synchronization phases via speculative processing techniques. By removing the replica synchronization phase from the critical path of execution of transactions, SPECULA allows threads to speculatively pipeline the execution of both 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 that are desirable in non-sandboxed environments like STMs. Via an experimental study, based on a fully-fledged prototype and on both synthetic and standard STM benchmarks, we demonstrate that SPECULA can achieve speedups 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 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] has garnered considerable interest in both academic and industrial research communities. Recently, the maturing of TM research has led to a TM implementation in hardware for a commodity high-performance microprocessor and also to the inclusion of TM supports for the world's leading open source C/C++ compiler.

Distributed TMs (DTMs) represent a natural evolution of TMs, which aims at combining the simplicity of TMs with the scalability and failure resiliency achievable by exploiting resource redundancy within distributed platforms. This makes 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 (e.g. X10 [2]) provide programmatic support for the DTM abstraction. As for Cloud computing, several recent NoSQL data-grid platforms, such as Oracle© Coherence or Red Hat© Infinispan, expose transactional APIs and rely on in-memory storage to achieve higher performance and elasticity levels.

This work has been partially supported by national funds through FCT - Fundação para a Ciência e a Tecnologia, under projects PTDC/EIA-EIA/102496/2008 and PEst-OE/EEI/LA0021/2011, by the Cloud-TM project (co-financed by the European Commission through the contract no. 57784) and by COST Action IC1001 EuroTM.

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, the cost of replication in DTMs is particularly exacerbated [3]. As a direct consequence, state of the art transactional replication mechanisms would introduce a significant overhead when employed in TM contexts. This phenomenon is clearly highlighted by the experimental data reported in Figure 1, which ware gathered using D^2STM, a state of the art Java-based DTM platform [4] that integrates an Atomic Broadcast (AB) based replication protocol [5], [6]. This DTM platform was evaluated using a standard benchmark for Software TM (STM) systems, namely STMBench7 [7], which was deployed on a cluster of up to 8 nodes[1]. The plot shows that, even in complex benchmarks comprising long-running transactions, like STMBench7, the duration of the transaction commit phase (which is dominated by the latency of the distributed replica synchronization scheme) is normally 10x-150x longer than the local transaction execution time.

In this paper we aim at addressing this issue by introducing 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 on a single node, thus avoiding any form of synchronization till it 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, 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 replica synchronization phase from the critical path of transactions' execution, SPECULA allows threads to pipeline the computation of sequences of transactional and/or non-transactional code fragments, which are processed speculatively. If the outcome of the AB-

[1]The cluster, which represents the reference experimental test-bed used in the remainder of this paper, is composed by machines equipped with 2 quad-core Xeon processors at 2.13GHz and 8GB of RAM, interconnected via a Gigabit Ethernet.

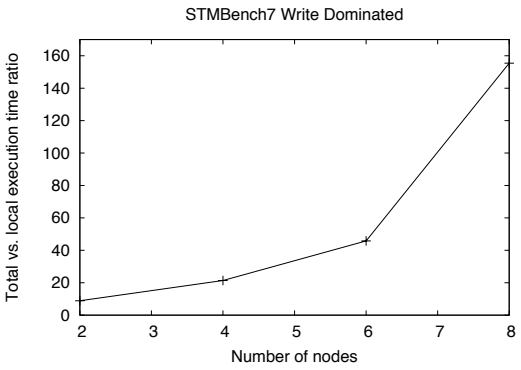


Figure 1. Total vs. local execution time ratio in a replicated STM

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 detects data and flow dependencies, squashing any affected speculative execution in a completely transparent fashion. In order to support the rolling-back of multiple speculatively committed transactions, and to revert the effects of speculatively executed non-transactional code, SPECULA relies on three main mechanisms: i) *continuations* [8], i.e. an abstraction that allows to save/restore threads' stack and point in computation; ii) a speculative multi-versioned STM (called *SVSTM*) that avoids blocking on read/write operations and guarantees 1-Copy Serializability (1CS) [9] for committed transactions, as well as serializability of the snapshots observed by all the transactions (even those that are eventually aborted to preserve 1CS); iii) undo-logging techniques to restore the state of the non-transactional heap.

We implemented a fully fledged prototype of SPECULA in Java, and, in order to achieve full transparency for applications, we relied 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, and to orchestrate the usage of continuations. In addition, SPECULA's runtime automatically detects non-reversible 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 evaluated SPECULA using both micro-benchmarks and standard STM benchmarks. Our experimental study shows that SPECULA allows achieving up to one order of magnitude speedups when compared with a baseline non-speculative transactional replication protocol.

The remainder of this paper is structured as follows. Section II discusses related work. Section III introduces the system model. The consistency criterion guaranteed by SPECULA is defined in Section IV. The SPECULA protocol is presented in Section V. Section VI reports the results of the experimental analysis. Section VII concludes the paper.

II. Related Work

Transactional data replication for fault tolerance has been intensively studied in literature, especially for what concerns replication of database systems. The results in this area include protocol specifications (see, e.g., [10], [5]), middleware based replication architectures (see, e.g., [11], [6]), as well as internal supports at the DBMS level (see, e.g., [12]). A key insight confirmed by several works in this area (see, e.g., [5]) is that AB-based replication protocols can avoid the well-known scalability limitations [10] of classic two-phase commit [13] based replication protocols and ensure robust performance even in scenarios of non-minimal contention.

More recently, the works in [4], [14], [15] have provided replication protocols specifically targeted at STM systems, by introducing a number of optimizations aimed at minimizing the cost of AB-based replication, such as bloom-filter based encoding schemes [4], lease abstractions permitting to minimize the usage of AB [14], and self-tuning mechanisms based on machine-learning techniques [15]. These optimizations are orthogonal to the key innovative idea behind SPECULA, namely preventing threads from blocking till the completion of the replica coordination phase. In fact, these optimizations could be integrated with SPECULA, with the goal of maximizing the efficiency of the distributed transaction certification phase.

In some of our recent works [16], [17], [18], [19] we have introduced replication techniques that exploit the, so called, optimistic delivery order (an early, albeit potentially erroneous indication of the final delivery order established by the AB service) to overlap, in an optimistic fashion, transaction execution and replica coordination phases. In these solutions, any transaction is executed on every replicated site, while SPECULA relies on a certification based scheme, to be executed after a transaction has been processed at a single site. Also, the above works 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 mix of state recoverability techniques at both data-layer and application levels.

Our work is also related to a set of results in the area of automatic speculative-parallelization in general purpose applications [20]. 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 [13]. We assume that the system is asynchronous and is augmented 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, AB 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 [21] (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 an STM [22], which allows application threads to combine sets of concurrent operations in a sequence of atomic *transactions*. We assume that threads can interleave the execution of transactional and non-transactional code.

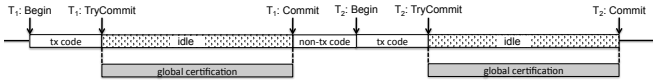
IV. Correctness Criteria

The global consistency criterion for SPECULA is 1-Copy Serializability (1CS) [9], which guarantees that the history of the finally committed transactions executed by any process in Π is equivalent to a serial history as it was executed on a single process. At the level of each single (non-replicated) process, SPECULA guarantees a correctness criterion analogous to the opacity criterion [23] (which, having been specified assuming a non-speculative transaction execution model, cannot be straightforwardly applied to SPECULA). Specifically, SPECULA ensures that the snapshot observable by any transaction T is equivalent to that generated by a sequential history of transactions, independently of whether T is eventually aborted or committed. Just like opacity, this criterion prevents any anomaly that may arise by observing system states that could never be generated in any serializable transaction history, and which could lead, in non-sandboxed environments like STMs, to arbitrary behaviors (such as infinite loops or crashes).

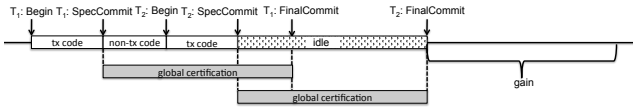
V. The SPECULA protocol

A. Protocol Overview

Analogously to classical certification-based replication schemes, e.g. [5], [6], 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 invokes commit and passes the local validation stage (that verifies the absence of conflicts with any concurrent transaction committed so far), it is locally committed in a *speculative* manner and the global AB-based transaction certification stage is executed in an asynchronous fashion.



(a) Execution example using CERT



(b) Execution example using SPECULA

Figure 2. Execution examples

This avoids blocking a thread th that issued a commit for a transaction T untill 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.

This 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 speculatively (but not yet finally) committed some transaction. A speculative data dependency arises whenever a 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 committed transaction T is removed whenever the final outcome (commit/abort) for T is determined.

In case of an abort event for a speculatively committed transaction, the above dependencies can lead to cascading abort of speculatively executed transactions, and require the restoration of non-transactional state (i.e. heap, stack, and 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 local processing and distributed 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. With both the protocols, T_1 is executed locally and, when it enters the commit phase, a *global certification* is started. At this point the two protocols diverge. With CERT, the application level thread is blocked

until T_1's global certification finishes. Instead, with SPECULA, T_1 is speculatively committed and the application level thread can immediately start processing the subsequent non-transactional code. Then it executes transaction T_2. Hence, in this scenario, SPECULA achieves a reduction of the overall execution time equal to the duration of T_1's certification.

B. High Level Software Architecture

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

The SVSTM layer provides isolation guarantees during transaction execution and ensures the atomicity of the state alterations associated with *Abort*, *Speculative Commit* and *Final Commit* events. These properties are guaranteed by a multi-version 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 snapshot that has been committed, possibly in a speculative fashion, before starting T. Further, unlike classical multi-version schemes, SVSTM allows a speculatively committed transaction T to externalize its writeset to other locally executing transactions before its final outcome is established. Also, SVSTM atomically removes T's writeset if the certification phase of T determines that it needs to be aborted, in which case cascading abort of transactions having speculative (data and/or flow) dependencies from T is triggered.

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 causally depends on a speculatively committed state. To tackle this case, SVSTM offers a simple programming interface, called *Synch*, that can be used by the applications to force a synchronization point, which blocks speculation on the caller thread until all the transactions speculatively committed so far by that thread are actually committed, or at least one of them is aborted (in which case this speculative execution needs to be squashed).

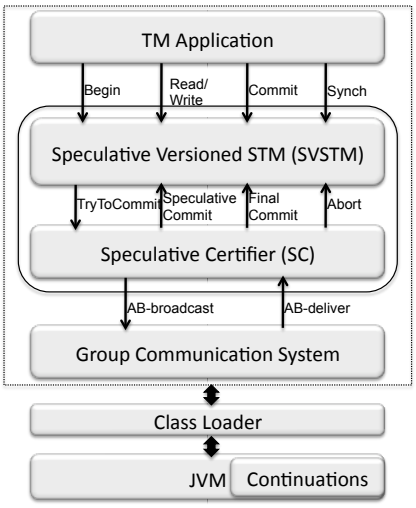


Figure 3. Software architecture of a SPECULA instance

When the application layer invokes commit (*Commit* event) for a transaction T, SVSTM requests T's validation to the *Speculative Certifier* (*SC*), which first attempts to validate T speculatively. If the validation fails, an *Abort* event is triggered. Otherwise, *SC* triggers a *Speculative Commit* event to the upper layer, unblocking the application level thread, and activates a second validation phase aimed at certifying T's consistency on a global basis by relying on the total ordering service provided by the *G*roup Communication System (GCS). If this second certification phase is also successful, a *Final Commit* event is triggered. Otherwise, an *Abort* event is generated.

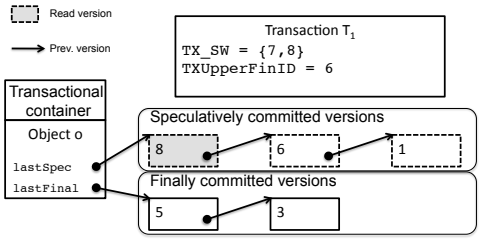
In order to support the rollback of chains of speculatively committed transactions, possibly interleaved by non-transactional code, SPECULA relies on the *continuation* abstraction, and on the automatic management of non-transactional code via undo-logging techniques. A continuation reifies the control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

C. Speculative Execution of Transactions

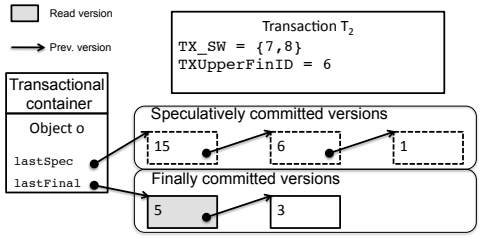
Due to space constraints, we cannot include the pseudo-code of the algorithm used by SVSTM to manage the speculative execution of transactions (which can be found in [24]). Nevertheless, in the following we describe in detail the key mechanisms used by SVSTM.

Management of Speculative/Final Snapshots. As in classical multi-versioned STMs, e.g. [25], SVSTM maintains multiple committed versions of data and uses a timestamp based mechanism to associate each transaction *T* with a snapshot that is used during *T*'s 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 *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 (speculatively/finally) committed versions of



(a) Read of a speculative version



(b) Read of a final version

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

each transactional object. The speculative and final versions maintained by *TC* are organized in two separate single-linked lists. As shown in Figure 4(a), *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/finally) committed version in the list (if any), (ii) the associated value, (iii) a scalar timestamp vn, called version number, 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 using a single scalar timestamp, which we call *UpperFinID*. As in 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 *Speculative Window* (*SW*), which is a linked list containing an element for each speculatively committed transaction by the local SPECULA replica. The advancement of speculative commits is regulated by an additional 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 the transaction writeset, a new speculative version is added in the associated *TC* and tagged with the new value of *UpperSpecID*. Further, a node is added to the head of SW, causing its widening. The speculative window is 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 by an instance of SVSTM is equivalent to the one obtained by serializing the speculatively committed (and not aborted) transactions contained in the speculative window after the sequence of finally committed transactions up to the one having (final committed) snapshot id equal to *UpperFinID*.

Transaction Activation. Each transaction is associated with the following data structures: i) the read-set and writeset, maintaining, respectively, the set of items read and written by the transaction; 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 which is set, upon starting the transaction, to the current value of *UpperFinID*; iv) a linked list, called TX_SW, that is initialized at transaction's activation time, by creating a copy of the local SVSTM's *SW*; v) a reference to a continuation, called $resumePoint$, which is initialized to `null` and is updated upon the speculative commit of the transaction to capture the thread's execution context at the end of the transaction; vi) an $undoLog$ that, as it will be discussed in Section V-D, is used to rollback any update of the non-transactional heap performed by 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 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 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 by simply buffering the new written values in the transaction's writeset. Let us now analyze how a read operation issued by a transaction T on object o is managed. In order to guarantee that transactions observe the values that they previously wrote, it is first necessary to check whether o is already in T's writeset and, in the positive case, the read operation returns that o's value. Otherwise, it is 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. However, before returning v^*, it is necessary to check whether the transaction that created

v^* has been aborted in the meanwhile (recall each version stores a pointer to the creating transaction). In such a case T needs to be aborted as well.

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

To better clarify the read mechanism we show in Figure 4 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 of Figure 4(a), T observes the speculative version having timestamp 8, since transaction 8 is in T's SW. In the example of Figure 4(b), 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 every transaction T always observes a snapshot 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 dependencies.

As the speculative snapshot observed by a transaction is guaranteed to be serializable, 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 transaction subsequently executed by the same thread. 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 as $RO(T_{up})$.

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 finally) 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 started in that 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 speculative transactions from which there is a read-from dependency (wrt T_{up} and the read-only transactions in $RO(T_{up})$). We denote the latter set of transactions as $SRF(RO(T_{up}))$. Finally, in order to allow remote nodes to track speculative flow dependencies, the unique identifier of the thread executing T_{up} and its current $epoch$ value are also sent via AB.

Speculative Commit. The speculative commit of a transaction T by a thread th implies two main operations. First, a checkpoint of the current thread execution context is generated by creating a continuation and storing it in T's $resumePoint$ variable. This continuation will be used in case it is later necessary to rollback the execution of (transactional/non-transactional) code that is processed by th in a speculative fashion, following the speculative commit of T. 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 *SW* of the local SVSTM.

Global Validation and Final Commit. When at node n_i an AB message 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. To this end, it is first checked in which epoch T_{up} has been originated. If 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 greater epoch value than the one currently associated with th at node n_i, say e', the corresponding entry in $EpochTable$ is updated.

Next, it is checked whether the read-only transactions $RO(T_{up})$ have read a valid snapshot. To this end it is checked if the transactions in $SRF(RO(T_{up}))$ have been, in the meanwhile, finally 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 finally committed, and the *UpperFinID* snapshot that was attributed to T when it was finally 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 finally committing a read-only transaction $\tilde{\text{T}}$ 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 $\tilde{\text{T}}$ depends (namely, $SRF(\tilde{\text{T}})$). Then, for each object in $\tilde{\text{T}}$'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 greater than $\tilde{\text{T}}$'s *TXUpperFinID*, and the creating transaction, say $\hat{\text{T}}$, is not present in $\tilde{\text{T}}$'s SW, then it means that $\hat{\text{T}}$ has been serialized before $\tilde{\text{T}}$ in the final delivery order, and that $\tilde{\text{T}}$ has missed $\hat{\text{T}}$'s snapshot. In order to guarantee 1CS, $\tilde{\text{T}}$ needs to be aborted.

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 finally committed version, say *lastFinal*(o), has version number less than or equal to 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 more recent (finally 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 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 versions 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 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 transaction 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 correspond to the minimum value, across any active transaction 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 at the moment in which any Java class is loaded by the JVM.

When any of these bytecode instructions is executed by a thread th, the pre-image of the target address is saved in the *undoLog* 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 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 [26] library to detect, at the bytecode level, calls to native code via JNI and other non-reversible operations (e.g. invocations of the java.io.* package), and automatically insert bytecode that forces the thread to wait for the validation of any 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 assurance 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 only after having finally 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 of T_i detects whether T_i would observe the same snapshot as if it were executed via a non-speculative multi-version 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 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 finally committed transactions, followed by ii) the sequence of speculatively committed transactions ordered according to n_i's SW. The latter ones have been locally validated before being speculatively committed. This guarantees 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 to 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 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 used as baseline 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 reaches the commit stage [5],[6]. The local STM exploited for the baseline is JVSTM [25], a state of the art multi-versioned STM that never blocks or aborts read-only transactions. The AB layer relies on the LCR algorithm [27], a recent ring-based AB algorithm that is known for its robust performance especially at high throughput. The results reported in this section were obtained using the same experimental platform employed to produce the plot reported in Figure 1.

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 speculation level as an independent parameter, we aim at assessing the impact of SPECULA on various performance indicators, in particular the latency of the AB layer and the transaction abort rate.

The first considered workload is a synthetic benchmark, called Bank, that was selected to assess the maximum gains achievable by SPECULA. Bank exclusively entails update transactions that simulate the transfers among bank deposits, and was configured not to generate any conflict. These are

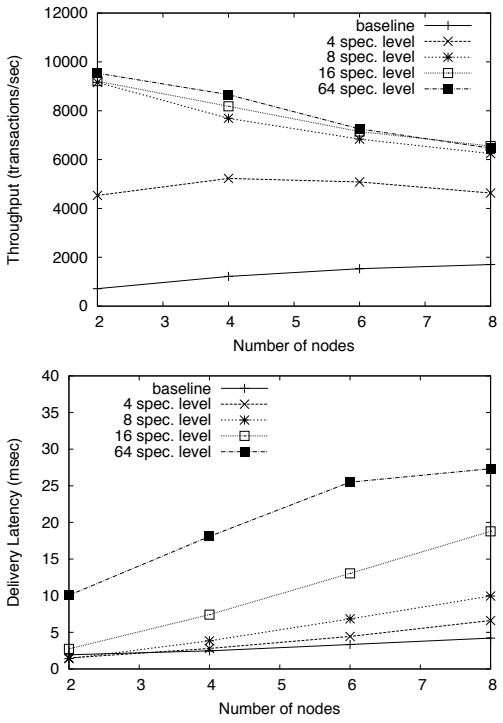


Figure 5. Bank benchmark

clearly ideal conditions for SPECULA, and the plots in Figure 5 confirm the achievement of striking performance gains, with speedups 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 speculation level beyond 8 does not provide any additional speedup for SPECULA. This is explainable by observing in the bottom plot that the AB-delivery latency grows very rapidly for speculation levels above 8, highlighting that SPECULA has already hit the maximum throughput of the underlying AB implementation.

The second considered workload is the write-dominated configuration of STMBench7 [7]. 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 causing an abort rate ranging from 5% (for 2 nodes) to around 15% (for 8 nodes). Also in this scenario (see Figure 6) SPECULA achieves remarkable speedups, up to a 4.4x factor in the configuration with 6 nodes and speculation level equal to 8. Interestingly, unlike in the previously analyzed scenario, with STMBench7, an excessively high setting for the speculation 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 speculation levels lower than 8,

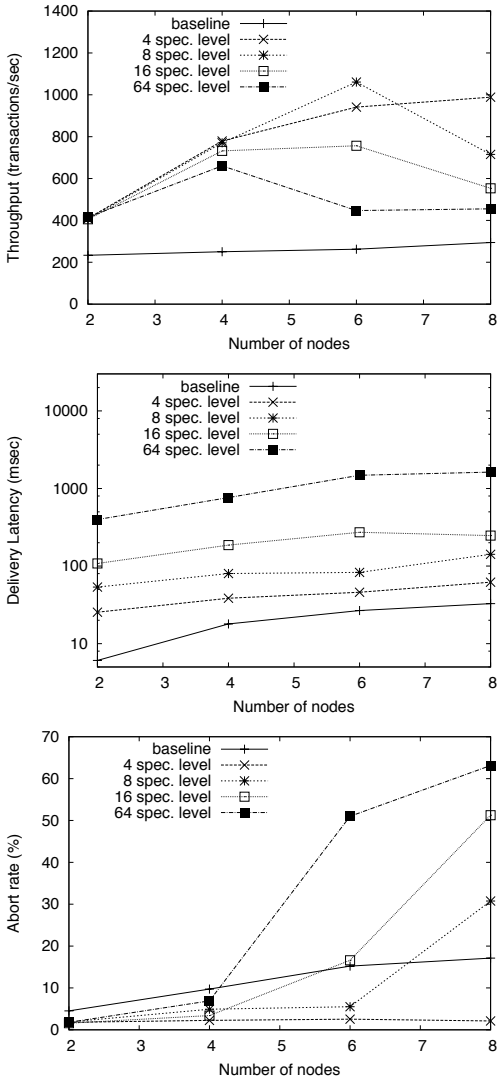


Figure 6. STMBench7 write dominated

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 speculation level increases, on the other hand, the GCS load accordingly increases, 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, increases.

Finally, in Figure 7, we analyze the read-dominated workload of STMBench7, a setting in which this benchmark

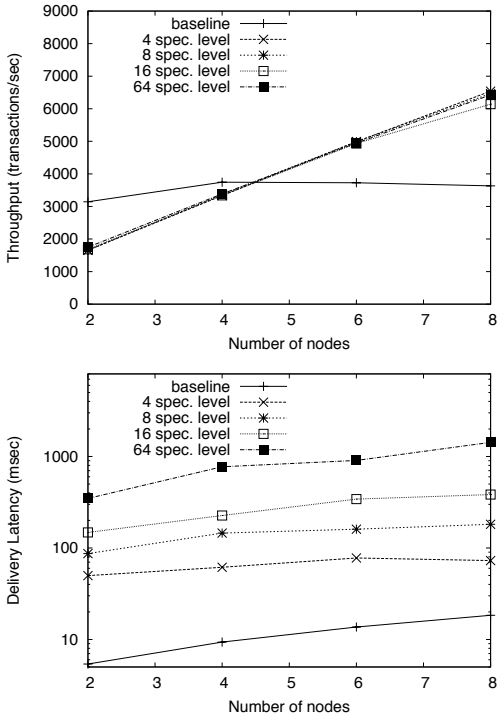


Figure 7. STMBench7 read dominated

generates 94% of read-only transactions. This is clearly a least favorable scenario for SPECULA since, 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 unfavorable scenario, SPECULA achieves comparable, or even higher 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 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 for SPECULA in all the considered settings. Analogously to the Bank benchmark's scenario, also in this case increasing the speculation 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 in 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.

VII. Conclusions

In this paper we have introduced SPECULA, a novel transactional replication protocol that aims at minimizing the overhead of the replica coordination phase by speculating on its eventual success, and letting application level

threads optimistically pipeline the execution of subsequent transactional/non-transactional code blocks.

SPECULA relies on three key building blocks: an innovative multi-version concurrency control, which manages the coexistence of speculatively and finally committed data versions while ensuring serializability of the snapshots observed by transactions throughout their execution; a novel distributed certification protocol, which ensures that the history of finally committed transactions is 1-copy serializable even in scenarios entailing speculatively executed transactional/non-transactional code; a set of mechanisms (including continuations, undo-logging of updates on non-transactional heap variables, automatic detection of non-revocable operations) aimed at ensuring total transparency of the management of speculative executions (e.g. squashing of speculatively executed code blocks) for the user level application.

We have developed a Java prototype that uses SPECULA to replicate an STM system, and shown that SPECULA can achieve speedups of up to one order of magnitude with respect to state of the art non-speculative solutions.

REFERENCES

[1] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proc. of 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] P. Romano, N. Carvalho, and L. Rodrigues, "Towards distributed software transactional memory systems," in *Proc. of LADIS*. ACM, 2008.

[4] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D^{2}STM: Dependable distributed software transactional memory," in *Proc. of PRDC*. IEEE Computer Society, 2009, pp. 307–313.

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

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

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

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

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

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

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

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

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

[14] N. Carvalho, P. Romano, and L. Rodrigues, "Asynchronous lease-based replication of software transactional memory," in *Proc. of Middleware*. Springer-Verlag, 2010, pp. 376–396.

[15] M. Couceiro, P. Romano, and L. Rodrigues, "Polycert: Polymorphic self-optimizing replication for in-memory transactional grids," in *Proc. of Middleware*. Springer-Verlag, 2011, pp. 309–328.

[16] 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.

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

[18] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues, "Brief announcement: on speculative replication of transactional systems," in *Proc. of SPAA*. ACM, 2010, pp. 69–71.

[19] N. Carvalho, P. Romano, and L. Rodrigues, "Scert: Speculative certification in replicated software transactional memories," in *Proc. of SYSTOR*. ACM, 2011, pp. 10:1–10:13.

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

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

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

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

[24] S. Peluso, J. Fernandes, P. Romano, F. Quaglia, and L. Rodrigues, "SPECULA: Speculative replication of software transactional memory," INESC-ID, Tech. Rep. 12, April 2012.

[25] J. 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.

[26] I. Anjo and J. Cachopo, "Jaspex: Speculative parallel execution of java applications," in *Proc. of INFORUM*, 2006.

[27] R. Guerraoui, 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.