OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems

Roberto Palmieri, Francesco Quaglia
Dipartimento di Informatica e Sistemistica
Sapienza Rome University, Italy
{palmieri,quaglia}@dis.uniroma1.it

Paolo Romano
Distributed Systems Group
INESC-ID, Lisbon, Portugal
romano@inesc-id.pt

Abstract—In this article we focus on actively replicated transactional systems built on top of an Optimistic Atomic Broadcast (OAB) service. We propose a transactional replication protocol named OSARE, which opportunistically processes transactions in multiple, speculative serialization orders. This is done to increase the likelihood that the final message ordering established by the OAB service matches one of the already speculated serialization orders. OSARE attempts to serialize incoming transactions speculatively, in an order compliant with the one of the optimistic deliveries. At high load, when both the likelihood of mismatches between the optimistic and final delivery orders and the degree of concurrency among transactions increase, the likelihood also increases that a transaction misses the write on some data item issued by concurrently executing transactions that precede it along the speculated serialization order. In these scenarios, OSARE opportunistically maintains the generated snapshot, by keeping on executing transactions in a serialization order that is unaligned with the optimistic deliveries. It also re-triggers the speculative materialization of the original serialization order along which the snapshot-miss event occurred. This biases the speculative exploration towards serialization orders “close” to the order of the optimistic deliveries, which is indeed used as a initial reference starting from which any other order to be speculated is determined. Via a simulation study in the context of Software Transactional Memory (STM) systems, which has been based on traces collected by running the APPIA group-communication tool and common STM benchmarks, we show how OSARE provides up to 160% response-time speedup compared to recent proposals that entail speculative transaction processing, but that materialize speculation exclusively along the optimistic delivery sequence.

I. INTRODUCTION

Active replication, a.k.a. state machine replication [32], is a classical means for providing fault-tolerance and high availability. This scheme is based on the enforcement of consensus among the replicas on a common total order according to which the incoming requests must be processed. The problem of establishing, in a non-blocking fashion, the agreed upon total order is typically encapsulated by a so called Atomic Broadcast (AB) group communication primitive, namely a convenient abstraction of consensus for which a wide range of alternative implementations have been proposed in literature [10]. The issue of enforcing request processing in an order that deterministically complies with the AB outcome is instead delegated to a replica control mechanism [27].

As for efficiency and performance aspects, AB can introduce significant latency along the critical path of request processing, with negative repercussions on the response time [29]. The replica control mechanism, on the other hand, can severely restrict parallelism in order to avoid mismatches between the actual order of request processing (and hence the shared state update trajectory) and the AB defined order [21], [22]. The latter aspect represents an extremely relevant issue of late, given that multi-core and many-core processors have nowadays become mainstream, and considering the current trend towards massively parallel computing platforms [22]. Unsurprisingly, in literature a number of approaches have been proposed aimed at coping with the above issues, which have been typically specialized to meet the requirements or to exploit the opportunities of specific application domains, such as data streaming [5] or transaction processing [18], [30].

In this article our focus is on active replication in the context of transaction processing systems, for which a key optimization technique has been presented in [18], and then further exploited in [21]. This technique is based on the observation that replicas can use the spontaneous network delivery order as an early, although possibly erroneous guess of the total delivery order of messages eventually defined via AB. This idea is nicely encapsulated by the Optimistic Atomic Broadcast (OAB) primitive [18], representing a variant of AB in which the notification of the final message delivery order is preceded by an early optimistic message delivery indication, typically available after a single communication step. By activating transaction processing upon the optimistic delivery of a message, rather than waiting for the final order to be established, OAB-based replication techniques allow to overlap the (otherwise sequential) replica synchronization and local computation phases.

Clearly, serializing the optimistically delivered transactions according to the optimistic message delivery order does not pay-off in case of non-minimal likelihood of mismatch between optimistic and final message ordering. In such a case, in fact, optimistically processed transactions may have to be aborted and restarted right after OAB completion, thus nullifying any performance gain associated with their early activation. Unfortunately, ordering mismatch is more likely to occur at high load, which leads the aforementioned approaches to loose their effectiveness precisely when the increased workload would have instead demanded an increased efficiency in order to ensure adequate performance. Experimental data provided in [18] show in fact that, even in case of predictable networks like LANs, the spontaneous order property typically holds only for normal load periods, but it is quite unlikely to
hold when the load tends to become heavy.

In order to cope with this issue, the work in [30] proposed the idea to speculatively explore the entire set of serialization orders in which optimistically delivered transactions would observe different snapshots (i.e., states) of the underlying transactional system. Such a complete-exploration approach ensures the ability to eventually guess any actual order established by the OAB service, provided the availability of sufficient time and computational resources. On the other hand, the number of alternative transaction serialization orders to be explored can grow up factorially with the number of optimistically delivered messages, which may induce prohibitive costs, thus limiting the viability of this solution in practical settings.

In this paper we present a novel active replication protocol for transactional systems based on an opportunistic paradigm, which we name OSARE - Opportunistic Speculation in Active Replication (1). Similarly to [21], OSARE maximizes the overlap between replica coordination and transaction execution phases by propagating, in a speculative fashion, the (uncommitted) post-images of completely processed (or simply complete in the following for the sake of brevity), but not yet finally delivered, transactions across chains of conflicting transactions. However, the rules according to which the speculated serialization orders are selected in OSARE are completely different from any existing literature solution.

To maximize concurrency, OSARE activates the processing of a transaction, say T, as soon as it is optimistically delivered and attempts to serialize it after any previously optimistically delivered transaction, say T'. Clearly, this attempt may fail, given that, at the time in which the processing of T is activated, T' may still be running. Therefore, the read operations issued by T may miss the values generated by T', an event that we refer to as “snapshot-miss” in the following. In such a case, OSARE does not abort and restart T to ensure that its serialization order is aligned with the optimistic message delivery order. Conversely, OSARE takes advantage of the occurrence of snapshot-miss events in an opportunistic fashion, by exploring additional serialization orders not only for T, but also for any transaction originally serialized after T whose execution may be affected, possibly transitively, by the snapshot-miss involving T. Also, OSARE biases the speculative exploration towards the serialization order aligned with the optimistic message delivery order, by triggering the (re-)activation of a new instance of T (and, recursively, of the transactions having developed a read-from dependency from T) as soon as it detects T’s snapshot-miss. The OSARE respawning logic guarantees that freshly activated transactions are correctly serialized after T', thus avoiding redundant executions of multiple instances of the same transaction observing identical snapshots. In addition, OSARE is designed to guarantee that, for each transactional request delivered by the OAB service, there eventually exists one speculative transaction instance whose view of the serialization order is aligned with the optimistic delivery order.

It is worthy to highlight that the likelihood of snapshot-miss events is higher in high concurrency scenarios, namely when the inter-arrival time of optimistic deliveries is relatively short compared to transaction processing latency. Interestingly, these scenarios are precisely those in which the probability of mismatches between the optimistic and final message delivery orders, and consequently the added value of exploring additional speculative serialization orders, are higher. The ability of OSARE to adjust adaptively its degree of speculation on the basis of the current level of concurrency in the system represents a unique, innovative feature, which, to the best of our knowledge, does not appear in any literature result in the field of actively replicated transactional systems.

We assess the performance of OSARE via an extensive simulation study. Our simulation model relies on traces collected by running well known benchmarks for Software Transactional Memory systems (STM) [6], [15], and the APPIA [20] group communication toolkit. Our experimental study highlights the effectiveness of the opportunistic speculative approach pursued by OSARE, showing that it can provide up to 160% response-time speedup compared to recent proposals [21] entailing speculation limitedly to the serialization order associated with the optimistic delivery sequence.

The remainder of this paper is structured as follows. Section II discusses related work. The system model we consider is described in Section III. Section IV presents the OSARE protocol. The properties provided by OSARE are described in Section V. The simulation study in presented in Section VI.

II. RELATED WORK

There exists a large amount of literature proposals targeting transactional systems’ replication, which entail protocol specification (see, e.g., [12], [17], [25]) as well as replication architectures that have been based on middleware level approaches (see, e.g., [19], [23], [24]) and/or on extensions of the inner logic of transactional systems (see, e.g., [17], [34]). As shown in [33], the most promising techniques are those based on total order broadcast primitives, which include active replication schemes like the one we present in this paper. In active replication, (Optimistic) Atomic Broadcast - (O)AB - primitives are exploited to coordinate processing activities by determining, in a non-blocking fashion, a global transaction serialization order (across all replicas), thus circumventing scalability problems that are known to affect classical eager replication mechanisms based on distributed locking and atomic commit protocols [12]. Relevant proposals along this direction can be found in, e.g., [1], [18]. Differently from OSARE, some of these proposals do not entail speculative processing across chains of conflicting transactions. In particular, they either execute transactions in a non-speculative fashion only after the AB service is already finalized (see [1], [16]), or execute at most a single optimistically delivered transaction, along a chain of conflicting transactions, before the OAB gets completed (see [18]). The latter protocols require a-priori knowledge of the data items to be accessed since each speculatively executed transaction needs to pre-acquire locks on its whole data set in order to block the execution of any other conflicting transaction. This is not the case for OSARE, which adopts an optimistic transaction scheduling approach that does not

---

1OSARE means “to dare” in Italian.
require beforehand knowledge of data access patterns, thus maximizing transaction concurrency when deployed in parallel computing architectures (e.g. multi- and many-core platforms) [22].

Like OSARE, our recent works in [21] and [30] both make use of speculation across chains of conflicting transactions while OAB is in progress. The work in [21] uses a lock-based concurrency control mechanism that throttles speculation to bias it towards a serialization order corresponding to the optimistic delivery order. Instead, OSARE exploits a fully optimistic transaction scheduling mechanism with no locks, that allows opportunistic exploration of alternative serialization orders and can enhance performance in scenarios of mismatch between optimistic and final ordering by the OAB service. The proposal in [30] is based on the complete speculative exploration of all the plausible serialization orders of optimistically delivered transactions (depending on actual transaction conflicts), which allows sheltering from any mismatch between the optimistic and the final delivery order. Hence, differently from OSARE, there is no set of "preferential" serialization orders to be opportunistically processed in a speculative fashion. As a consequence, compared to the work in [30], OSARE provides more productive usage of resources, since it only processes those serialization orders that are deemed as relevant to match the final OAB ordering depending on the (dynamically varying) level of concurrency and its reflection on the actual schedule within the system.

The present work has also relations with the approaches in [3] and [28], which explored the idea of speculatively executing transactions to enhance performance. The work in [3] targets non-replicated real-time databases and shows the benefits, in terms of transaction timeliness, by speculatively forking, upon detection of a conflict, a copy of the current transaction that remains idle and serves as a save-point to reduce the cost of aborts. The solution in [28] targets distributed databases relying on distributed locking and on a final atomic commit phase for validating transactions. Compared to these solutions, OSARE is completely different in structure since it targets actively replicated rather than distributed transactional systems.

Being our approach based on speculation, it naturally fits fine-grain transaction processing scenarios, such as Software Transactional Memory (STM) based applications. According to this perspective, it exhibits relations with replication approach for main memory database systems (see, e.g., [7], [31]). These proposals differ from OSARE since they either target primary-backup replication schemes [7] or data-partitioned cluster-based systems [31], while our targets are actively, fully replicated systems.

Finally, the works in [5], [4] propose out-of-order computation schemes in event stream processing applications. The idea of leveraging optimism to boost performance is in common with our approach, though OSARE targets a different application domain with different features/characteristics (transactional systems) and pursues the idea of speculatively exploring multiple out-of-order paths (i.e. serialization orders) to maximize the chances of successfully overlapping the communication and processing phases.

III. System Model

We consider a classical distributed system model [14] consisting of a set of transactional processes $Π = \{p_1, \ldots, p_n\}$ that communicate via message passing and can fail according to the fail-stop (crash) model. If a process does not fail we say it is correct.

We assume the availability of an OAB service offering the following classical API: $TO$-$broadcast(m)$, which allows broadcasting message $m$ to all the replicated processes in $Π$; $Opt$-$deliver(m)$, which delivers message $m$ to a process in $Π$ in a tentative, also called optimistic, order; $TO$-$deliver(m)$, which delivers message $m$ to a process in $Π$ in a so called final order that is the same for all the processes in $Π$. The OAB service provides the following set of properties [26]:

- **Termination** - If a correct process $TO$-$broadcasts m$, then it eventually $Opt$-$delivers m$;
- **Global Agreement** - If a process $Opt$-$delivers m$, then every correct process eventually $Opt$-$delivers m$;
- **Local Agreement** - If a correct process $Opt$-$delivers m$, then it eventually $TO$-$delivers m$;
- **Global Order** - If two processes $p_i$ and $p_j$ $TO$-$deliver messages m and m’, they do so in the same order;
- **Local Order** - If a process $TO$-$delivers m$, it does this only after having $Opt$-$delivered m$.

Figure 1 shows the software architecture of each process $p_i ∈ Π$. Applications generate transaction requests and submit them to their local Transaction Manager (XM), specifying the business logic to be executed (e.g. a stored procedure in a database or the name of a method in a transactional memory system) and the corresponding input parameters (if any). XM is responsible of (i) propagating (through the OAB service) the transactional request across the set of replicated processes, (ii) executing the transactional logic on the underlying Transactional Store (TS), and (iii) returning the corresponding result to the user-level application. With no loss of generality, we assume the existence of a function $Complete()$, used to explicitly notify XM about the completion of the business logic associated with a transaction.

We assume that each data item $X$ maintained by TS is associated with a set of versions $\{X^1, \ldots, X^n\}$, and that, at any time, there exists exactly one committed version of a data item $X$. On the other hand, other versions can be in the complete state, which means that the creator transaction has reached the complete stage, but that its outcome (commit/abort) has not yet been finalized. Complete data versions are generated by fully executed transaction computations, and are used
to aggressively propagate updates to conflicting transactions. While issuing read operations, only those versions that are either committed or complete are (speculatively) visible. This avoids that transactions can observe intermediate data item values, which can be still modified by a transaction that is manipulating them, before it reaches the complete stage.

We assume that neither the sequence of operations to be executed within a transaction, nor the data items to be accessed by each operation are a-priori known. Conversely, we assume that transaction data access patterns can vary depending on the current state of the underlying transactional store. More precisely, we assume that the transactional business logic is snapshot deterministic [30] in the sense that the sequence of read/write operations it executes is deterministic once fixed the return value of any of its read operations. In other words, whenever an instance of transaction \( T \) gets re-executed and observes a same snapshot \( S \), defined as the set of values returned by all its read operations, then it behaves deterministically by always executing a same set of read/write operations.

The manipulation of the data items occurs via the following primitives offered by the TS layer: \( \text{setComplete}(X^T, T) \), which marks a data item version \( X^T \) written by transaction \( T \) as complete, and \( \text{unsetComplete}(X^T, T) \), which is used for removing a complete data item version \( X^T \) originally exposed by transaction \( T \).

### IV. THE OSARE PROTOCOL

In this section we provide the description of the OSARE protocol, by first introducing some key notations and data structures and then discussing the protocol pseudo-code.

#### A. Protocol Notations and Data Structures

The OAB service delivers transactions, each of which is denoted as \( T_i \) in our protocol. OAB delivered transactions are however never directly executed by XM, which, conversely only executes speculative transaction instances, denoted using the notation \( T^i \).

Each transaction \( T^i \) specularly executed by OSARE keeps locally track of its own serialization view, defined as the totally ordered sequence of transactions that are expected to be serialized before \( T^i \). The construction of the per-transaction view of the serialization order relies on two main data structures: a global list of speculative transaction identifiers, called \( \text{OptDelivered} \), accessible by all the transactional threads; a local list of speculative transaction identifiers, referred to as \( T^i \cdot \text{SpeculativeOrder} \), which is associated with the transactional thread currently handling transaction \( T^i \). The sequence of speculative transactions recorded within \( T^i \cdot \text{SpeculativeOrder} \) expresses, on the basis of the view by \( T^i \), the order according to which speculative transactions preceding \( T^i \) should be serialized. This determines a history of speculative transactions whose snapshots may be visible by \( T^i \)'s read operations.

We use the notation \( T^h \xrightarrow{\text{SpeculativeOrder}} T^i \) to indicate that \( T^h \) precedes \( T^i \) within the ordered list \( T^i \cdot \text{SpeculativeOrder} \). This expresses that, according to the view of \( T^i \):

1. \( T_h^i \) and \( T_i^i \) belong to the same speculative history of transactions;
2. \( T_h^i \) and \( T_i^i \) are both expected to be serialized before \( T_i^i \);
3. \( T_h^i \) is expected to be serialized before \( T_i^i \).

By convention, the special transaction identifier \( T^0_\omega \) represents the minimum element for the \( \rightarrow \) relation for whichever transaction \( T^i \). This notation is used to encapsulate the history of already committed transactions that, according to \( T^0_\omega \)'s view of speculative serialization expressed via the relation \( \rightarrow \), must necessarily be serialized before \( T_i^i \) and before any transaction belonging to \( T_i^i \cdot \text{SpeculativeOrder} \). Always by convention, \( T_i^i \) represents the maximum element of the \( \rightarrow \) relation.

Overall, denoting with \( (T^{h_1}_k, \ldots, T^{h_n}_k) \) the sequence of transactions recorded within \( T^i \cdot \text{SpeculativeOrder} \), we have:

\[
T_\omega^0 \xrightarrow{T_k^j} T^{h_1}_k \xrightarrow{T_j^i} \ldots \xrightarrow{T^{h_n}_k} T^{h_n}_k \xrightarrow{T_k^j} T_i^i.
\]

The global list \( \text{OptDelivered} \) maintains the transaction identifiers whose speculative serialization view is aligned with the order of optimistic deliveries generated by the local OAB service. An addition to this list occurs upon the Opt-delivery of a transaction \( T_i \) by appending, to the tail of the list, the identifier of the first instance of speculative transaction associated with \( T_i \), by which we convention denote as \( T_i^0 \). Before such an update occurs, \( T_i^0 \cdot \text{SpeculativeOrder} \) is populated with the current content of \( \text{OptDelivered} \). In this way, \( T_i^0 \) is provided with a view of the speculative serialization order which is (at least upon its activation) aligned to the one associated with the current sequence of optimistically delivered transactions. The global list \( \text{OptDelivered} \) is also updated when a commit occurs for an already TO-delivered transaction. In this case one element is eliminated from the list, reflecting the fact that the corresponding speculatively executed transaction has been finalized. If the element is not the top standing one, it means that some TO-delivery has subverted the Opt-delivery order. A third type of update of \( \text{OptDelivered} \) occurs when a transaction \( T_i^i \) currently recorded within this list is discovered to have been actually executed along a speculative serialization order that diverges from the one currently expressed by the list content. As we will see, in this case, a new instance of speculative transaction \( T^{h_i}_k \) is spawned, substituting \( T_i^i \) within that list (possibly recursively causing other spawns and substitutions). This is done in order to guarantee the existence of a speculative transaction instance that is being processed in a serialization order compliant with the order expressed by the currently optimistically delivered transactions.

#### B. Protocol Logic

The protocol pseudo-code is shown in Figures 2 and 3, and is discussed in the following.

**Optimistic delivery of transactions.** Upon the Opt-deliver event of a transaction \( T_i \), the XM instantiates a speculative transaction \( T^i \), and then sets up the serialization order to be seen by this new transaction instance by copying the current content of \( \text{OptDelivered} \) into \( T^0 \cdot \text{SpeculativeOrder} \). Next,
void TODelivered(OrderedList<T> T;
OrderedList<T> TODelivered, OptDelivered;
Set<T> TransactionMemory, ActiveXacts;

upon Opt-Delivered(Transaction T) do
T_D = T, createNewSpecXact();
T_D.SpeculativeOrder = copy(OrderedList); Optimizer.enqueue(T_D);
ActivateSpeculativeTransaction(T_D);

void ActivateSpeculativeTransaction(Transaction T_i) 
ActiveXacts.add(T_i);
start processing thread;

DataItemValue Read(Transaction T'_i, DataItem X) 
if (X ∈ T'_i.WriteSet) return T'_i.WriteSet.get(X).value;

select version of X completed or committed by
if (T'_i.WriteSet.get(X).version == LatestCommitted)
// the committed version is written by T'_i by definition
T_i.ReadSet.add(X);
T_i.ReadFrom.add(T'_i);
return T'_i.ReadSet.get(X).value;

void Write(Transaction T'_i, DataItem X, Value v) 
if (X ∈ T'_i.WriteSet) T'_i.WriteSet.update(X, v);
else T'_i.WriteSet.add(X, v);

void Complete(Transaction T'_i) 
atomically do
T'_i.isCompleted = TRUE;
∀ T_j s.t. (3 X ∈ T'_i.WriteSet and T'_i = max{T'_i : T'_i.T_i = T'_i.j Dispose X});
T'_i.SpeculativeOrder = copy(T'_i.SpeculativeOrder);
T'_i.SpeculativeOrder.remove(T'_i.j Dispose X);
T'_i.SpeculativeOrder.replace(T'_iSpecifyOrder.
T'_i = T'_i.j Dispose X);
wait until TODelivered.topStanding == T'_i;
if (∀ X ∈ T'_i.ReadSet. X version == LatestCommitted) T'_i.RaiseEvent(Commit);
else T'_i.RaiseEvent(Abort);

void wave(Transaction T'_i, Transaction T'_j Dispose X, Transaction T'_k Dispose X)
∀ T'_j Dispose X s.t. T'_j Dispose X in T'_i.ReadFrom do
T'_jDisposeX = T_j.createNewSpecXact();
T'_jDisposeX.SpeculativeOrder = copy(T'_jDisposeX.SpeculativeOrder);
T'_jDisposeX.SpeculativeOrder.replace(T'_jDisposeX.T_i, T'_jDisposeX.T_i Dispose X);
T'_jDisposeX.SpeculativeOrder.remove(T'_jDisposeX);
if (T'_jDisposeX in OptDelivered) Optimizer.replace(T'_jDisposeX, T'_jDisposeX);
wave(T'_jDisposeX, T'_jDisposeX, T'_jDisposeX);
∀ T'_k Dispose X s.t. T'_k Dispose X in T'_jDisposeX.ReadFrom do
T'_kDisposeX.SpeculativeOrder.replace(T'_kDisposeX.T_i, T'_kDisposeX.T_i Dispose X);
ActiveSpeculativeTransaction(T'_kDisposeX);

Fig. 2. Behavior of XM (Part A).

the XM appends T'_0’s identifier within the global list OptDelivered to reflect that at least one instance of speculative transaction associated with T'_i exists, and that it should be serialized at the tail of the sequence of speculative transactions currently recorded within the OptDelivered list. Finally, the XM activates the transaction processing activities for T'_0 by invoking the ActivateSpeculativeTransaction function with T'_0 as input parameter. The latter function also adds T'_0 to the set of active transactions ActiveXacts.

Handling of read and write operations. When a transaction T'_i issues a read operation on a data item X, the XM verifies whether a version of X belongs to the write set of the reading transaction. In the positive case, the written value is returned (ensuring that the read operation returns the version of X associated with the last write issued by T'_i during its execution). On the other hand, if T'_i has not previously issued a write on X, the precedence relation associated with T'_i.SpeculativeOrder is used in order to determine which version of X that should be made visible to T'_i. To this end it is used a simple rule that allows identifying the most recent version exposed by a completed or committed transaction according to the serialization view of T'_i. Specifically, it is determined the maximum speculative transaction T'_j preceding T'_i according to the serialization view, which has i) written X and ii) already completed its execution.

The logic associated with a write operation of a transaction T'_i on a data item X is also very simple: the XM simply stores the updated value of X into the write-set of the writing transaction. As we will see, the data item versions generated by a transaction T'_i are in fact made all atomically visible to the remaining speculative transactions only once that T'_i reaches its completion phase.

Completion of speculative transactions. The core of the OSARE protocol is represented by the logic handling the completion of a speculative transaction. Specifically, when the Complete method is executed by the XM for transaction T'_i, each data item version created (i.e. written) by T'_i is made speculatively visible by setting its state to the complete value. Before making the snapshot produced by T'_i visible, however, it is first checked whether every transaction T'_k that, according to its serialization view, is serialized after T'_i, is still correctly initialized and has not reached its completion phase.
executing along that order, or whether it has instead missed the snapshot generated by the execution of $T_i^s$. More in detail, a snapshot miss event is detected in case: i) $T_i^s$ wrote some data item $X$ for which $T_j^{xl}$ has already issued a read operation, and, ii) $T_i^s$ is the last speculative transaction to have written $X$ among those in $T_i^s$’s speculative view. In this case, in fact, $T_j^{xl}$ has observed a different version of $X$, despite, according to its serialization view, it should have observed the version of $X$ generated by $T_i^s$.

A snapshot is detected if $T_j^{xl}$ should have seen the version that $T_i^s$ wrote, but it has observed some different version of $X$. In this case we are in the presence of a snapshot-miss event, since the current serialization view of $T_j^{xl}$ has not been respected, given that it did not observe the data item version that it should have read from $T_i^s$. The following three actions are taken to handle a snapshot-miss event:

1. A new speculative instance $T_j^{xl}$ is activated, setting its serialization view to be equal to the one currently associated with $T_j^{xl}$. Given that $T_j^{xl}$ has now reached completion, the new instance $T_j^{xl}$, during its execution, is guaranteed not to miss the snapshot produced by $T_i^s$.

2. The serialization order of $T_j^{xl}$ is then updated by removing $T_j^{xl}$ (namely the transaction whose write operation $T_i^s$ missed) from $T_j^{xl}$’s SpeculativeOrder. Next, if $T_j^{xl}$ was originally recorded within OptDelivered, it is replaced by $T_j^{xl}$ within this list. This reflects the fact that $T_j^{xl}$ is known not to be any longer in a serialization order compliant with that of the optimistic message delivery order, and that there is now a new incarnation of $T_j$, namely $T_j^{xl}$, which will be activated (in the wave, see below) in order to pursue this serialization order.

3. The snapshot-miss event is recursively propagated via the wave() method (described shortly afterwards) across chains of transactions that were transitively serialized (according to their own serialization view) after the transaction $T_j^{xl}$ involved in the snapshot-miss event.

After having handled all the snapshot-miss events detected upon its completion, $T_j^{xl}$ simply remains waiting for the corresponding transaction $T_j$ to be TO-delivered, and to become the top standing element within the TODelivered queue. As it is clearer in the following, this means that for any transaction $T_j$, which was TO-delivered before $T_j$, there exists a corresponding speculatively executed transaction $T_j^s$ that has been already committed. Hence $T_j^s$ can now be safely validated (by verifying whether it has read data items belonging to the latest committed snapshot) and, depending on the validation’s outcome, a commit, or an abort, event is raised to finalize the commit, or the abort, of the speculative transaction.

**Recursive propagation of snapshot-miss events.** As we have just explained, the completion of a transaction $T_i$ can trigger a series of snapshot-miss events involving transactions $T_j^{xl}$, which, according to their serialization view, should have observed the value of a data item updated by $T_i^s$ when executing a read operation and that have instead observed a version of that data item created by a different transaction. In this case $T_j^{xl}$’s speculative view is updated (removing $T_i^s$ from it) to reflect the occurrence of the snapshot-miss event, and a new speculative instance of transaction $T_j$, namely $T_j^{xl}$, is activated that will be guaranteed not to miss the snapshot created by $T_j^s$. In order to pursue, on one hand, the opportunistic exploration of additional serialization orders, and, on the other hand, the completion of a sequence of transactions serialized in an order compliant with the optimistic message delivery order, OSARE transitively propagates the handling of the snapshot-miss event via the wave method.

The transaction $T_j^{xl}$, in fact, may have already completed its execution and exposed its snapshot to a different speculative transaction, say $T_i^{xl}$. In this case, even if $T_i^{xl}$ had not missed the snapshot generated by $T_i^s$, it is still transitively involved by the snapshot-miss event affecting $T_j^{xl}$. Analogously to $T_j^{xl}$, therefore, $T_i^{xl}$ needs to be removed by the speculative view of $T_i^{xl}$. Further, in order to pursue the exploration of a serialization order compliant with the optimistic message delivery order, a new speculative instance of $T_i$, namely $T_i^{xl}$, needs to be activated, which should now include in its serialization view $T_i^{xl}$. Finally, just like in the Complete method, it is verified if $T_i^{xl}$ was considered to be serialized in an order compliant with the optimistic delivery order (by checking whether it is included in OptDelivered). In the positive case, the OptDelivered sequence needs to be updated, replacing $T_i^{xl}$ with $T_i^{xl}$, reflecting the fact that it is now the latter one to be expected to be serialized according to the optimistic delivery order. Note that the wave method relies on an elegant recursion technique to ensure the complete propagation of the snapshot-miss across the whole set of transactions that have established a transitive read-from relationship from $T_j^{xl}$.

Upon returning from the recursive call, the XM substitutes $T_i^{xl}$ with $T_i^{xl}$ from the speculative view of every transaction $T_i^{xl}$ that i) contained $T_j^{xl}$ in its speculative view, and that ii) did not develop a read-from dependency from $T_j^{xl}$. This is necessary in case $T_i^{xl}$ is still active, in order to ensure that during its subsequent reads, it will be able to observe the snapshot generated by $T_i^{xl}$, thus correctly realigning $T_i^{xl}$’s speculative view towards the serialization order compliant with the optimistic message delivery order.

Finally, activation of processing activities for the spawned transaction $T_j^{xl}$ takes place right before returning from wave().

**Final delivery of transactions.** As for the handling of final delivery events, the associated logic only entails the enqueuing of the delivered transaction within TODelivered. This ensures that the corresponding placeholder is sequentialized after all the already TO-delivered ones and regulates that all the replicas validate (and ultimately commit) transactions in the same total order.

**Abort and Commit Events.** The handling of the abort event simply removes the aborting transaction from the set ActiveXact and from any speculative order currently recording the transaction identifier. It also propagates the abort event towards all the transactions having read-from dependency from the currently aborting transaction.
A bit more sophisticated is the handling of the commit event. In this case, the committing transaction identifier $T_i^k$ is removed from ActiveActs, and every data item it wrote is marked as committed. Then, the corresponding transaction $T_i$ is dequeued from the TODelivered list. This can cause another TO-delivered transaction to become the top standing transaction of this list, eventually enabling the commit of one of its corresponding speculative instances. Next, whichever transaction instance $T_i^s$ currently present within OptDelivered is removed from this list, in order to ensure that instances of $T_i$ are no longer to be considered as belonging to the speculative portion of the serialization order associated with the sequence of optimistically delivered transactions. Further, the abort event is raised for all the transactions different from $T_i^s$ that are instances of $T_i$. This leads to the abort of all the speculative transactions that had developed a, possibly transitive, read-from dependency from a a instance of $T_i$ different from $T_i^s$. $T_i^s$ is then removed from any serialization view that is currently recording it (again because it is logically passed to the committed transaction history). Finally, it is necessary to verify whether transactions having (direct or transitive) read-from dependencies from the committing transaction are still valid. This is required since, as hinted, $T_i^s$ is moved to the committed history. Therefore we need to verify whether the transactions exhibiting dependencies on the snapshot produced by $T_i^s$ are still executing along a consistent speculative serialization path. This check is performed via the Validate function, which simply verifies whether the items read by those transactions still correspond to those produced by the transactions representing the maximum elements exposing these items as complete along the corresponding serialization orders. In the negative case it means that the transaction (directly or transitively) reading from the committing transaction $T_i^k$ needs to be restarted by speculating along the modified path where $T_i^s$ has been moved to the committed history (see the AbortRetry module).

V. Protocol Properties

In this section we provide some informal arguments on the set of properties ensured by OSARE:

Opacity [13] - The opacity property guarantees that (O.1) committed transactions should appear as if they were executed sequentially, in an order that agrees with their real-time ordering, (O.2) no transaction should ever observe the modifications to shared state done by aborted or live transactions, and (O.3) all transactions, including aborted and live ones, should always observe a consistent state of the system.

In each replica, OSARE ensures property (O.1) by committing transactions only after a validation phase that would detect any unserializable behavior. It ensures (O.2) because read operations can only return either a committed value, or the value generated by a transaction whose execution has already reached the complete phase (and hence is neither live nor aborted at the time of the read). It ensures (O.3) since the read of a transaction $T_i^s$ always returns the value generated by the latest complete transaction that precedes $T_i^s$ according to its own view of the speculative serialization order. This mechanism clearly excludes the possibility of incurring in any anomaly in case all the reads executed by $T_i^s$ take place i) after that all the transactions preceding it according to its speculative order have already completed, and ii) if the speculative order of $T_i^s$ is never altered.

Let us start by analyzing the scenario in which a transaction $T_j^j$, which is serialized after $T_i^s$ according to its speculative order, completes after $T_i^s$ has already issued at least a read operation. Assume, with no loss of generality, that this read is on a data item $X$ and returned a value written by a transaction $T_j^j$ such that $T_j^j \xrightarrow{T_j^h} T_i^s$. In this case, the only possible anomaly that could affect $T_i^s$ would arise if also $T_j^j$ had written $X$ and if $T_j^j \xrightarrow{T_j^h} T_i^s$. In this case, if $T_i^s$ were ever to read, along its execution, the value of any data item written by $T_j^j$, it would observe an inconsistent state. In fact, having read $X$ from $T_j^j$, $T_i^s$ has already serialized itself before $T_j^j$. This would therefore lead to a violation of property (O.3). On the other hand, this scenario is avoided in OSARE since, as soon as $T_j^j$ completes (namely, before making its data items speculatively visible), it detects that $T_i^s$ has missed its write on $X$ and removes itself from the speculative order of $T_i^s$, preventing $T_i^s$ from ever reading values written from $T_j^j$.

To complete the reasoning on the absence of non-opaque schedules it remains to assess the scenarios in which the speculative order of a transaction is altered during its execution. This can happen in three situations. The first one, whose correctness has just been discussed, is associated with a snapshot-miss event. The remaining two cases are associated, respectively, with the Commit and Abort events of a transaction $T_j^h$. Both events, in fact, trigger the removal of $T_j^h$ from the speculative order of every transaction $T_i^s$ that is supposed to serialize itself after $T_j^h$. In the case of a Commit event, $T_j^h$ is immediately aborted (and restarted) if it is detected that the previously executed read operations would have observed different values if re-executed according to the updated speculative order. On the other hand, in the case of an Abort event, $T_j^h$ is immediately aborted if it developed any (direct or transitive) read-from dependency from $T_i^s$. In both cases, therefore, it is guaranteed that $T_j^h$, were it still be running at the time in which its speculative order is altered, will not observe inconsistent snapshots when continuing its execution.

2. 1-Copy Serializability [2] - 1-Copy Serializability is ensured since transactions are committed at every site only upon a deterministic validation that is executed by all replicas in the same total order, i.e., the final delivery order of the OAB service.

3. Non-redundant speculation [30] - This property ensures that no two speculative instances $T_j^s, T_j^s$ of the same transaction $T_j$ observe the same snapshot. This follows by observing that OSARE activates a new speculative transaction $T_j^s$ only if it detects that a transaction $T_j^s$ has missed a value written by a transaction $T_j^s$ serialized before $T_j^s$ according to its speculative order. In this case, $T_j^s$ will observe at least a...
data item version different from those observed by $T_j^i$ since $T_j^{x \text{id}}$ will not miss the value written by $T_j^i$, having $T_j^n$ already competed and made visible its snapshot. The same is true for all the transactions that are (recursively) spawned due to an (direct or transitive) read-from dependency on $T_j^i$. They will execute along a serialization order where the snapshot by $T_j^i$ is not visible since it is replaced by the one provided by $T_j^{x \text{id}}$.

4. **Lock-freedom** [11] - Lock-freedom guarantees that there is always at least a thread to make progress, thus ruling out deadlock and livelock scenarios. In OSARE, this is a direct consequence of the fact that the transaction currently representing the top standing element within the **TODelivered** queue always experiences an abort free (re)run.

5. **Speculative trajectories explored by OSARE** - Rather than attempting a complete speculative exploration, OSARE biases speculation exploration towards the optimistic delivery order. More precisely, OSARE explores exclusively serialization orders obtained by extracting subsequences of variable length of the optimistic delivery order, see Figure 4. This choice allows, on one hand, to favor the opportunistic exploration of serialization orders “close” to the optimistic delivery order. On the other hand, it allows to rely on a simple and lightweight logic (when compared with previous protocols oriented to speculation completeness, such as [30]) to identify the set of speculative serialization orders to be explored.

   Given that OSARE activates additional speculative transactions in an opportunistic fashion (i.e. only upon the occurrence of snapshot-miss events), the actual number of speculative transactions spawned by OSARE depends on the level of concurrency and conflict among the set of optimistically delivered transactions.

   Nevertheless, it is still possible to determine analytically an upper bound on the number of speculative transactions activated in OSARE as a function $\theta(n)$ of the length $n$ of the sequence $\sigma$ of optimistically, but not yet finally, delivered transactions. Figure 4 illustrates the scenario in which OSARE explores the maximum number of alternative serialization orders for a sequence $\sigma$ of length 4. The number of nodes of such a (partial) permutation tree can be, in general, enumerated using the following expression:

$$\theta(n) = \sum_{i=1}^{n} \delta(i, n)$$

where $\delta(i, n)$ denotes the number of distinct subsequences of $\sigma$ obtainable after discarding from $\sigma$ the first $i-1$ messages, and is computable recursively as:

$$\delta(i, n) = \begin{cases} 1 + \sum_{j=i+1}^{n} \delta(j, n) & i \neq n \\ 1 & i = n \end{cases}$$

By using standard unfolding techniques, it is possible to show that $\theta(n) = 2^n - 1$.

**VI. SIMULATION STUDY**

In this section we show the results of a simulation study aimed at assessing the performance of OSARE. To this purpose we developed a process-oriented simulator using the JavaSim framework. We developed detailed simulation models of both the OSARE and AGGRO [21] protocols. We choose AGGRO as baseline protocol for the evaluation of OSARE since, as already discussed in Section II, it shares with OSARE the idea of performing speculative processing across chains of conflicting transactions while the OAB is in progress. Unlike OSARE, however, AGGRO relies on lock-based concurrency control that throttles speculation to force it to process transactions exclusively along the optimistic delivery order.

In order to accurately model the execution dynamics of transactions in replicated systems, we rely on a trace-based approach. Traces related to data accesses and transaction duration have been collected by running a set of widely used benchmarks for STMs[15]. The machine used for the tracing process is equipped with an Intel Core 2 Duo 2.53 GHz processor and 4GB of RAM. The operating system running on this machine is Mac OS X 10.6.2, and the used STM layer is JVSTM[6]. The simulation model of the replicated STM system comprises a set of 4 replicated STM processes, each hosted by a machine equipped with 32-cores processing transactions at the same rate as in the above architecture.

We configured the benchmarks to run in single threaded mode, so to filter out any potential conflict for both hardware resources and data. Also, we extended JVSTM in order to transparently assign a unique identifier to every object within the STM memory and to log every operation (namely, begin/commit/rollback operations, and read/write memory-object access operations) along with its timestamp. This allowed us to gather accurate information on the data access patterns of the benchmark applications and on the time required for processing each transaction (in absence of any form of contention).

The traces were collected running a number of heterogenous benchmarks. Specifically, we used: i) three microbenchmark applications, Red Black Tree, List and SkipList, which were originally used for evaluating DSTM2[15] and, later on, adopted in a number of performance evaluation studies of STM systems [6], [9]; ii) two benchmarks of the STAMP benchmark’s suite [8], namely Yada and Labyrinth++. Unlike the former microbenchmarks, both these benchmarks are fairly complex applications, characterized by transactions with large read and write sets and that encompass non trivial computations. Specifically, Yada++ is a parallel implementation of the Rupperts algorithm for Delaunay mesh refinement, which uses
transactions to concurrently modify a shared graph. Labyrinth, instead, implements a variant of the Lee’s algorithm to lay in parallel the junctions of an electrical circuit.

The above benchmarks were configured not to generate any read-only transaction. This choice depends on the fact that, in both protocols considered in this study, read-only transactions can be executed locally, without the need for propagation via the atomic broadcast. By only considering update transactions, we can therefore precisely assess the impact of the atomic broadcast latency on the performance of a replicated system, as well as the performance gains achievable by OSARE.

The transactions’ arrival process via optimistic and final message deliveries is also trace-driven. Specifically we use traces generated by running a sequencer based OAB implementation available in the Appia[20] GCS Toolkit on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a switched Gigabit Ethernet. We injected in the system messages of 512 bytes with an exponentially distributed inter-arrival rate, having mean $\lambda$. We treat $\lambda$ as the independent parameter of our study, letting it vary in the range [1000,4000] messages per second. In our experimental platform, this range of workloads encompasses scenarios of load for the GCS varying from low-moderate up to high. As expectable, the mismatch between optimistic and final delivery order (or message re-ordering for the sake of brevity henceforth) rate steadily increases along with the message arrival rate, ranging from 16%, at 1000 msgs/sec, up to 48%, at 4000 msgs/sec. In OSARE’s concurrency control algorithm, at high concurrency levels, namely when the probability of mismatches between the optimistic and final delivery orders is higher, the chances of exploring alternative (i.e. not equivalent to the optimistic message delivery order) transaction serialization orders correspondingly increase. OSARE takes advantage of the occurrences of these events by triggering, in an opportunistic and lightweight fashion, the speculative exploration of additional serialization orders across chains of conflicting transactions.

First, with OSARE, the number of transactions that have already started (or completed) processing in a serialization order compliant with the OAB final delivery sequence is up to 50% higher than in AGGRO. Second, at high load scenarios, the number of transactions aborted in AGGRO is around 4x larger than in OSARE. This depends on the fact that in scenarios of snapshot miss, where OSARE opportunistically explores additional serialization orders, AGGRO uses an aggressive rollback-retry mechanism that discards the speculative trajectory and re-activates the transaction to ensure that is correctly serialized according to the optimistic delivery order. This policy of AGGRO pays off in absence, or at negligible levels of message re-ordering, at it favors the timely materialization of the transaction serialization order associated with the optimistic delivery. On the other hand, as soon the probability of message re-order becomes non minimal, AGGRO leads to a significant waste of computational resources, which are conversely fruitfully exploitable by OSARE thanks to its opportunistic speculative approach.

Finally, interesting conclusions can be drawn by analyzing the statistics on the average and maximum number of speculative transactions generated in OSARE. At 4000 transactions per second, i.e. at very high load for the OAB (at this message rate the percentage of message reordering between optimistic and final delivery is of about 48%), the average number of speculative instances activated for a given transaction (across all the evaluated benchmarks) is 2.7. In other words, in OSARE, in addition to the serialization order associated with the final delivery order, 1.7 additional serialization orders are explored for each transaction. Our experimental data shows also that, on average, the CPU utilization at this throughput level is less than 8% with OSARE on the simulated hardware architecture. This is explainable by taking into account that, in replicated STM settings, such as the one evaluated in this study, the (AB) communication latency is up to orders of magnitude larger than the transaction processing time. Therefore, even despite the speculative exploration of additional serialization orders, several of the 32 cores of the simulated processor remain unutilized for long period of time. Overall, these results highlight that the additional demand for computational resources caused by OSARE’s speculative approach is expected to be perfectly sustainable by recent multicore and manycore architectures.

**Analysis of the Results.** The plots in Figure 5 report the speed-up achieved by OSARE vs AGGRO, evaluated as the percentage of additional latency for executing a transaction (i.e. the average time since the TO-broadcast of a transaction till its commitment) in AGGRO with respect to its counterpart in OSARE. The data highlight striking performance gains up to around 160% for OSARE, with the gain increasing as the load, and the message re-ordering, grows.

This is due to the fact that, by opportunistically processing a transaction in multiple serialization order, OSARE achieves a more effective overlapping between the processing and communication phases. This claim is supported by two main experimental findings.

First, with OSARE, the number of transactions that have already started (or completed) processing in a serialization order compliant with the OAB final delivery sequence is up to 50% higher than in AGGRO. Second, at high load scenarios, the number of transactions aborted in AGGRO is around 4x larger than in OSARE. This depends on the fact that in scenarios of snapshot miss, where OSARE opportunistically explores additional serialization orders, AGGRO uses an aggressive rollback-retry mechanism that discards the speculative trajectory and re-activates the transaction to ensure that is correctly serialized according to the optimistic delivery order. This policy of AGGRO pays off in absence, or at negligible levels of message re-ordering, at it favors the timely materialization of the transaction serialization order associated with the optimistic delivery. On the other hand, as soon the probability of message re-order becomes non minimal, AGGRO leads to a significant waste of computational resources, which are conversely fruitfully exploitable by OSARE thanks to its opportunistic speculative approach.

Finally, interesting conclusions can be drawn by analyzing the statistics on the average and maximum number of speculative transactions generated in OSARE. At 4000 transactions per second, i.e. at very high load for the OAB (at this message rate the percentage of message reordering between optimistic and final delivery is of about 48%), the average number of speculative instances activated for a given transaction (across all the evaluated benchmarks) is 2.7. In other words, in OSARE, in addition to the serialization order associated with the final delivery order, 1.7 additional serialization orders are explored for each transaction. Our experimental data shows also that, on average, the CPU utilization at this throughput level is less than 8% with OSARE on the simulated hardware architecture. This is explainable by taking into account that, in replicated STM settings, such as the one evaluated in this study, the (AB) communication latency is up to orders of magnitude larger than the transaction processing time. Therefore, even despite the speculative exploration of additional serialization orders, several of the 32 cores of the simulated processor remain unutilized for long period of time. Overall, these results highlight that the additional demand for computational resources caused by OSARE’s speculative approach is expected to be perfectly sustainable by recent multicore and manycore architectures.

**VII. Conclusions and Future Work**

In this work we presented OSARE (Opportunistic Speculation in Active REplication), an active replication protocol for transactional systems that combines the usage of Optimistic Atomic Broadcast service with a speculative concurrency control mechanism in order to aggressively overlap transaction processing and replica synchronization. The main innovative aspect of OSARE’s protocol consists in the speculative concurrency control mechanism that it employs to regulate the execution of optimistically delivered transactions. OSARE relies on a lock-free algorithm to bias the speculative serialization of transactions towards an order aligned with the optimistic message delivery order. Due to the lock-free nature of OSARE’s concurrency control algorithm, at high concurrency levels, namely when the probability of mismatches between the optimistic and final delivery orders is higher, the chances of exploring alternative (i.e. not equivalent to the optimistic message delivery order) transaction serialization orders correspondingly increase. OSARE takes advantage of the occurrences of these events by triggering, in an opportunistic and lightweight fashion, the speculative exploration of additional serialization orders across chains of conflicting transactions.

By adaptively adjusting its degree of speculation on the basis of the current level of concurrency in the system, OSARE achieves robust performance also in scenarios characterized by non-minimal likelihood of mismatches between the optimistic and final delivery orders, providing remarkable (up to 160%) speedups with respect to state of the art speculative replication
mechanisms.

An interesting research direction that we intend to pursue in our future work is the design of heuristics aimed at identifying which speculative serialization orders, among the ones explorable by OSARE, should be privileged/prioritized in order to meet constraints on the maximum number of available resources (e.g., processors or memory). The key challenge to address this problem consists in ideating mechanisms capable of estimating the cost/benefits associated with the exploration of additional serialization orders given the current workload characteristics for both the OAB and the transaction manager layers.

REFERENCES