ASAP: an Aggressive SpeculAtive Protocol for Actively Replicated Transactional Systems

Roberto Palmieri and Francesco Quaglia
DIIAG, Sapienza University, Rome, Italy

Paolo Romano
INESC-ID/IST, Lisbon, Portugal

***Abstract*—Recent advances in the field of replicated, fault tolerant transactional systems make systematic use of Optimistic Atomic Broadcast (OAB) group communication primitives in order to coordinate the replicas. According to this scheme, the replicas gain information on the existence of transactional requests before a final and global agreement is reached on the transaction serialization order. Hence, speculative processing schemes can be exploited in order to maximize the overlap between local computation and distributed coordination activities. In this article we present ASAP, an innovative Aggressive SpeculAtive Protocol, which exhibits the following two peculiarities: (A) it allows speculating along different transaction serialization orders, thus increasing the likelihood of successful overlap between local processing and coordination in case of mismatches between the optimistic and the final delivery sequence of incoming requests; (B) it speculates along chains of conflicting transactions, tracking data dependencies among transactions via an innovative concurrency control mechanism, which allows determining in a timely fashion the alternative serialization orders to be speculatively explored. Via a simulation study in the context of Software Transactional Memory systems we show ASAP can achieve robust performance independently of the likelihood of reorder between optimistic and final deliveries, providing remarkable performance improvements (enhancing the maximum sustainable throughput up to a 2x factor) with respect to state of the art speculative replication protocols.**

I. Introduction

Active replication, a.k.a. state machine replication [29], 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 [9]. The issue of enforcing request processing in an order that deterministically complies with the AB outcome is instead delegated to a replica control mechanism [23].

As for efficiency and performance aspects, AB can be responsible for non-negligible latency along the critical path of request processing, with negative repercussions on the response time [25]. 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 [18], [20]. 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 [20].

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 [4], [28] or transaction processing [17], [26].

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 [17], and then further exploited in [18]. 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 [17], 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 overlapping the (otherwise sequential) replica synchronization and local computation phases.

In this article we propose ASAP, an Aggressive SpeculAtive Protocol relying on OAB which makes systematic use of speculative transaction processing in order to further optimize the overlap between processing and distributed coordination phases. The peculiarities of ASAP are: (A) It allows speculating along differentiated serialization orders in an adaptive manner, depending on the real concurrency among incoming transactional requests, which is used as a convenient estimator of the likelihood of mismatches between the optimistic and final delivery order [17]. Overall, via the exploration of multiple, alternative serialization paths, ASAP increases the likelihood of guessing a transaction serialization order equivalent to the one that will be eventually established by the final delivery order. (B) It allows speculating along chains of conflicting transactions by using an innovative concurrency control scheme that tracks dependencies among transactions in an aggressive fashion, i.e. while transactions are still being executed rather than at their completion, as instead done by existing speculative replication protocols [19], [26]. This allows for early identification of transaction dependencies, which in

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.

turn permits early identification of alternative serialization orders to be explored while coordination is in progress, hence favoring concurrency.

We note that the combination of the features in points (A) and (B) uniquely characterizes ASAP since, in literature, aggressive exposition of new data versions to early determine transaction dependencies has been exploited in replication protocols which only speculate along the serialization order that matches with the optimistic delivery sequence (see [18]). Hence, alternative paths are not considered at all. On the other hand, protocols that speculate along multiple paths (see [19]) track transaction dependencies using lazy schemes that are activated only when the entire transaction has been processed. As we will show, this limits significantly the degree of concurrency achievable by these schemes when compared to ASAP.

We assess the performance of ASAP via an extensive trace driven simulation study in the context of Software Transactional Memory (STM) systems, showing that 2x throughput increase can be achieved compared to recent literature proposals. The results also highlight that ASAP can ensure robust performance even in presence of shifts of the degree of concurrency and of the likelihood of mismatches between optimistic and final delivery sequences.

The remainder of this paper is structured as follows. Section II discusses related work. The model of the target replicated system is introduced in Section III. ASAP is presented in Section IV. The results of the simulation study are provided in Section V.

II. Related Work

Speculative approaches aimed at improving the performance of database systems have been investigated in literature to some extent. The work in [3] explores the idea of speculatively executing transactions by exploiting the notion of save-point. Specifically, upon the detection of a conflict, a copy of the current transaction is speculatively forked and remains idle, thus acting as the save-point to reduce the cost of transaction aborts, which in turn allows for increased timeliness while completing transactions. This work is targeted at non-replicated real-time databases, while our focus is on the design of protocols for replicated transactional systems.

The solution in [24] targets distributed databases relying on distributed locking and on a final atomic commit phase for validating transactions. In particular, the post-images of data whose locks are held by prepared distributed transactions are allowed to be accessed by conflicting transaction, thus reducing the lock-wait time. A similar scheme is used by the protocol presented in [13], which is explicitly targeted at real-time time distributed databases. Compared to these works, our proposal can be considered as orthogonal since we target speculative processing in the context of replicated transactional systems not relying on distributed locking protocols, but on the usage of OAB primitives and of (innovative) local concurrency control mechanisms, acting at the level of each individual replicated node. Further, total-order based replication schemes [1], [16], [21] allow avoiding the well-known scalability problems that affect replication mechanisms based on distributed locking and atomic commit protocols [11]. Additionally, the works in [24], [13] do not deal with concurrent exploration of multiple serialization orders, which is instead one of the main objectives of our proposal.

The literature results more strictly related to our proposal are those presenting (O)AB-based active replication protocols for transactional systems, some of which systematically exploit speculative schemes for processing the transactions locally at each site thus aiming at maximizing the overlap between processing and distributed coordination. Relevant proposals along this direction can be found in, e.g., [1], [17]. Differently from ASAP, some of these proposals do not speculate along chains of conflicting transactions. In particular, they either execute transactions in a non-speculative fashion after the AB service is already finalized (see [1], [16]), or execute at most a single optimistically delivered transaction along the conflicting transactions chain, before the OAB gets completed (see [17]). Also, the latter protocols require a-priori knowledge of transactions' data accesses since each speculatively executed transaction needs to pre-acquire locks on its accessed data-set. Conversely ASAP adopts an optimistic transaction scheduling approach that does not require a-priori knowledge of data access patterns.

Like ASAP, our recent works in [18], [19], [26] make use of speculation along chains of conflicting transactions. The work in [18] constrains speculation to adhere to the serialization order corresponding to the optimistic delivery order, while ASAP allows concurrent exploration of alternative serialization paths, thus enhancing robustness in scenarios of mismatch between optimistic and final orders. The proposals in [19], [26] are based on speculative exploration of distinct serialization orders of optimistically delivered transactions. However, the determination of transaction dependencies, and hence the identification of the alternative orders to be conveniently explored, takes place by exposing the post-image of the written data only when transactions are fully processed. Instead ASAP adopts an aggressive approach that allows identifying transaction dependencies as soon as they arise (i.e. during transaction execution), hence increasing the timeliness according to which alternative serialization orders to be speculatively explored are identified. This increases the level of concurrency (and the actual overlap between processing and coordination), thus also allowing better exploitation of massively parallel platforms, such as those based on many-core technologies.

Finally, solutions such as [6] address, via speculative transaction processing, the issue of certification-based replication, while in this article we cope with speculation within active replication, which is known to provide different trade-offs in terms of bandwidth requirements for coordination activities.

III. System Model

We consider a classical distributed system model [12] consisting of a set of transactional processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing and adhere 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 exposing the following API: *TO-broadcast*(m), which allows broadcasting message m to all the 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. OAB provides the following set of properties [22]: **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.

Applications submit transactional requests to their local Transaction Manager (XM), specifying the business logic to be executed 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, 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. In real transactional architectures it could be implemented just as a wrapper for the commit command.

We assume that each data-item X is associated with a set of versions $\{X^1, \ldots, X^n\}$, and that, at any time, there exists exactly one committed version of data-item X. On the other hand, other versions can be either (A) in the complete state, which means that the creating transactions have reached the complete stage, but their outcome (commit/abort) has not been finalized yet, or (B) in the Work-in-progress (Wip) state, which means that the creating transactions are still live, thus having not yet reached completion.

We assume that the data items accessed by transactions are not a-priori known, and that data access patterns can vary depending on the observed state. More precisely, we assume that the business logic is *snapshot deterministic* [26] 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 is re-executed and observes a same snapshot, it behaves deterministically.

The manipulation of the data items occurs via the following primitives:

- `MarkAsWip(`T`,`X^T`)`, which is used for declaring the existence of a Wip version of data-item X created by transaction T;
- `UnmarkAsWip(`T`,`X^T`)`, which is used for undeclaring the existence of a previously declared Wip version of data-item X by transaction T;
- `MarkedAsWip(`T`,`X`)` which is used to query the existence of a Wip declaration on X by transaction T;
- `setComplete(`T`,`X^T`)`, which marks a data-item version X^T written by transaction T as complete, and
- `unsetComplete(`T`,`X^T`)`, which removes a complete data-item version X^T exposed by transaction T.

IV. ASAP: the Aggressive SpeculAtive Protocol

A. Preliminaries

We denote with T_i each single transactional request delivered by the OAB service (either optimistically or finally ordered). T_i is however never directly executed by XM, which only executes speculative instances of transactions associated with T_i, each of which is denoted as T_i^j.

Each speculative transaction T_i^j perceives its own view of the correct serialization order, defined as the totally ordered sequence of transactions that are expected to be serialized before T_i^j. The main data structures used by ASAP are:

- a shared list of speculative transaction identifiers, called OptDelivered, accessible by all the transactional threads;
- a local (per-thread) list of speculative transaction identifiers, referred to as T_i^j.RefOrder, which is associated with the transactional thread handling transaction T_i^j.

OptDelivered keeps, at any time, the identifiers of the transactions whose speculative serialization view is aligned with the order of optimistic deliveries locally generated by the OAB service. On the other hand, the sequence of speculative transactions recorded within T_i^j.RefOrder expresses, on the basis of the view by T_i^j, the order according to which speculative transactions preceding T_i^j should be serialized. Hence, T_i^j.RefOrder determines a history of speculative transactions whose produced snapshots should be visible by T_i^j's read operations.

The notation $T_k^h \overset{T_i^j}{\rightarrow} T_s^t$ is used to indicate that T_k^h precedes T_s^t within the ordered list T_i^j.RefOrder. This expresses that, according to the view of T_i^j:

(a) T_k^h and T_s^t belong to the same speculative history of transactions;
(b) T_k^h and T_s^t are both expected to be serialized before T_i^j;
(c) T_k^h is expected to be serialized before T_s^t.

By convention, the special transaction identifier T_α^ω represents the minimum element of the $\overset{T_i^j}{\rightarrow}$ relation for whichever transaction T_i^j. This notation is used to encapsulate the history of already committed transactions that, according to T_i^j's view of the speculative serialization, expressed via the relation $\overset{T_i^j}{\rightarrow}$, must be serialized before T_i^j and before any transaction belonging to T_i^j.RefOrder. Always by convention, the maximum element of the $\overset{T_i^j}{\rightarrow}$ relation is always represented by T_i^j. Hence, the last element of the list T_i^j.RefOrder is always set to T_i^j.

B. Protocol Pseudo-Code

The pseudo-code for ASAP is shown in Figures 1 and 2, and is discussed in the following.

Opt/TO-deliver. Upon the Opt-deliver event of a transaction T_i, XM instantiates a speculative transaction T_i^0, whose identifier is appended to the global list OptDelivered. Then XM sets up the serialization order to be perceived by T_i^0 by simply copying the content of OptDelivered within T_i^0.RefOrder. Then, XM activates the processing activities for T_i^0 by invoking ActivateSpeculativeTransaction, which also adds T_i^0 to the set ActiveXacts of the active transactions.

```
Global structures:
OrderedList<Transaction> TODelivered, OptDelivered;
Set<Transaction> ActiveXacts;

upon Opt-deliver(Transaction T_i) do atomically
  T_i^0 =T_i.createNewSpecXact();
  OptDelivered.append(T_i^0);
  T_i^0.RefOrder = copy(OptDelivered);
  ActivateSpeculativeTransaction(T_i^0);

void ActivateSpeculativeTransaction(Transaction T_i^s) do atomically
  ActiveXacts.add(T_i^s);
  start processing thread;

DataItemValue Read(Transaction T_i^s, DataItem X)
  if (X ∈ T_i^s.WriteSet) return T_i^s.WriteSet.get(X).value;
  atomically select T_j^t = max{T_j^f : T_j^f -T_i^s-> T_i^s and T_j^f exposed a version of X};
  set V_X = version of X written by T_j^t;
  if (MarkedAsWip(T_j^t, V_X)){
    wait for T_j^t completion or for a Resume event;
    if (T_j^t ∉ T_i^s.RefOrder) re-start read operation;
  }
  T_i^s.ReadSet.add(V_X);
  T_i^s.ReadFrom.add(T_j^t);
  return T_i^s.ReadSet.get(V_X).value;

void Write(Transaction T_i^s, DataItem X, Value v) do atomically
  if (X ∈ T_i^s.WriteSet) T_i^s.WriteSet.update(X, v);
  else{
    T_i^s.WriteSet.add(X, v)
    MarkAsWip(T_i^s, X^{T_i^s})
    ∀ T_j^t ∈ ActiveXacts s.t. (∃ X ∈ T_j^t.ReadSet :
      T_i^s = max{T_l^f : T_l^f -T_j^t-> T_j^t and X ∈ T_l^f.WriteSet }) do {
        T_j^{xId} = T_j.createNewSpecXact();
        T_j^{xId}.RefOrder = copy(T_j^t.RefOrder);
        T_j^{xId}.RefOrder.replace(T_j^t,T_j^{xId});
        if (T_j^t ∈ OptDelivered) OptDelivered.replace(T_j^t, T_j^{xId});
        propagateSnapshotMiss(T_j^t,T_j^{xId},T_i^s);
        T_j^t.RefOrder.Remove(T_i^s);
        if (T_j^t is waiting for T_i^s's completion) T_j^t.RaiseEvent(Resume);
        ActivateSpeculativeTransaction(T_j^{xId});
    }/end do
  }/end if-else
```

Fig. 1. ASAP Pseudo-code (Part I).

Upon the TO-deliver of T_i, the delivered transaction is simply queued within the TODelivered list.

Read operations. As for a read operation by transaction T_i^s on a data-item X, the first action executed is a check to determine whether X belongs to the write-set of T_i^s. In the positive case, the value of X currently registered within the write-set is returned, hence ensuring that transactions always observe the snapshots they generated. On the other hand, in case X does not belong to the write-set, the version of X to be read is identified by exploiting the information stored within the list T_i^s.RefOrder. To this end, the most recent version exposed by any live, completed or committed transaction is identified by selecting the topstanding transaction according to the $\xrightarrow{T_i^s}$ relation, that wrote X. In case such a version is associated with a live transaction T_j^t, it represents a Wip version. Hence, transaction T_i^s enters a wait phase that is interrupted either upon the completion of T_j^t (which exposes T_j^t's post-images of X) or by an explicit Resume event destined to T_i^s. As it will be further discussed later on, the two different cases are related to different scenarios, in terms of speculative run-time dynamics. In particular, an explicit Resume event will be raised towards T_i^s in case, depending on the future activities of the live transaction T_j^t, it will be determined that T_i^s, or some other transaction in between T_j^t and T_i^s within the ordering defined by T_i^s.RefOrder, missed some data-item version eventually produced by T_j^t.

Upon resuming, it is checked whether T_j^t still belongs to the serialization view seen by T_i^s, which is encoded by T_i^s.RefOrder. Actually, a change of the value of T_i^s.RefOrder (e.g. via the elimination of T_j^t) is an expression that an alternative speculative serialization path has been dynamically

```
void propagateSnapshotMiss(Transaction T_j^t, Transaction T_j^{xId}, Transaction T_i^s)
  ∀T_l^f ∈ ActiveXacts s.t. (T_j^t ∈ T_l^f.RefOrder) {
    T_l^{xId'} = T_l.createNewSpecXact();
    T_l^{xId'}.RefOrder = copy(T_l^f.RefOrder);
    T_l^{xId'}.RefOrder.replace(T_l^f,T_l^{xId'});
    T_l^{xId'}.RefOrder.replace(T_j^t,T_j^{xId});
    if (T_l^f ∈ OptDelivered) OptDelivered.replace(T_l^f, T_l^{xId'});
    propagateSnapshotMiss(T_l^f,T_l^{xId'},T_i^s);
    T_l^f.RefOrder.Remove(T_i^s);
    if (T_l^f is waiting for T_i^s's completion) T_l^f.RaiseEvent(Resume);
    ActivateSpeculativeTransaction(T_l^{xId'});
  } / end do

void Complete(Transaction T_i^s) do atomically
  T_i^s is marked as Completed
  ∀ X ∈ T_i^s.WriteSet do {setComplete(T_i^s, X^{T_i^s})};
  wait until TODelivered.topStanding == T_i;
  if (∀X ∈ T_i^s.ReadSet: X.version == LatestCommitted) T_i^s.RaiseEvent(Commit);
  else T_i^s.RaiseEvent(Abort);

upon TO-Deliver(Transaction T_i)
  TODelivered.enqueue(T_i);

upon Abort(Transaction T_i^s) do atomically
  ∀X ∈ T_i^s.WriteSet do
    if (T_i^s is complete) unsetComplete(T_i^s, X^{T_i^s})
    else UnmarkAsWip(T_i^s, X^{T_i^s});
  ∀T_j^h ∈ActiveXacts s.t. j ≠ i and T_i^s ∈ T_j^h.RefOrder do T_j^h.RefOrder.remove(T_i^s);
  ∀T_j^h s.t. T_i^s ∈ T_j^h.ReadFrom do T_j^h.RaiseEvent(Abort);
  ∀T_j^h s.t. T_j^t is waiting for T_i^s's completion do T_j^h.RaiseEvent(Resume);
  ActiveXacts.remove(T_i^s);

upon Commit(Transaction T_i^k) do atomically
  ActiveXacts.Remove(T_i^k);
  ∀X ∈ T_i^k.WriteSet do T_i^k.WriteSet.Commit(X);
  TODelivered.Dequeue(T_i);
  OptDelivered.Remove(T_i^*);
  ∀T_j^h ∈ActiveXacts s.t. h ≠ k do T_j^h.RaiseEvent(Abort);
  ∀T_j^h ∈ActiveXacts s.t. j ≠ i and T_i^k ∈ T_j^h.RefOrder do T_j^h.RefOrder.remove(T_i^k);
  ∀T_j^h ∈ActiveXacts s.t. T_i^k ∈ T_j^h.ReadFrom do T_j^h.RaiseEvent(Validate);

upon Validate(Transaction T_i^k) do atomically
  ∀X ∈ T_i^k.ReadSet do {
    compute T_j^h = max{T_l^f : T_l^f -T_i^k-> T_i^k and X ∈ T_l^f.WriteSet};
    if (T_i^k.ReadSet.get(X).Creator ≠ T_j^h){
      T_i^k.RaiseEvent(AbortRetry);
      break;
    }/end if
  }/end do

upon AbortRetry(Transaction T_i^s) do atomically
  ∀X ∈ T_i^s.WriteSet do
    if (T_i^s is complete) unsetComplete(T_i^s, X^{T_i^s})
    else UnmarkAsWip(T_i^s, X^{T_i^s});
  ∀T_j^h s.t. T_i^s ∈ T_j^h.ReadFrom do T_j^h.RaiseEvent(AbortRetry);
  restart transaction T_i^s;
```

Fig. 2. ASAP Pseudo-code (Part II).

selected for T_i^s. In such a case, the read operation is re-executed so to allow T_i^s to follow such an alternative path by re-selecting the version of X to be read, according to what specified by the new value of T_i^s.RefOrder. On the other hand, in case such a change of the serialization order does not happen, it means that the originally selected version has reached the complete stage. Hence, T_j^t has been completed and is still seen by T_i^s along its own serialization path. In such a case, the version of X wrote by T_j^t has become stable and can be safely returned to T_i^s.

Write operations. In case of a write operation by T_i^s, if the data-item X to be written already belongs to T_i^s's write-set, then its value is simply updated. On the other hand, if X does not belong to the transaction write-set, it is added to the write-set and the newly produced version of X is marked as Wip, which allows to promptly notify the occurrence of a write on X to every other speculative transaction having T_i^s in its speculative reference order. On the other hand, some of these transactions may have already issued a read operation on X, missing the version currently produced by T_i^s. Hence it is checked whether some of these speculative transactions, say T_j^t, according to its own serialization view expressed by T_j^t.RefOrder, should have read the version of X currently produced by T_i^s. This is determined by checking whether T_i^s is the maximum element of the $\stackrel{T_j^t}{\rightarrow}$ relation, which exposed a version of X. If this is the case, then it means that T_j^t missed the snapshot produced by T_i^s. ASAP takes advantage of these events by pursuing a transaction serialization order different from the one originally expressed by T_j^t.RefOrder. On the other hand, ASAP also keeps on pursuing the transaction serialization order originally expressed by T_j^t.RefOrder.

To achieve the twofold aim of materializing both the original serialization order and the actual serialization order currently experienced by T_j^t, the following actions are taken. Another instance T_j^{xId} of speculative transaction associated with the transactional request T_j is generated, and its reference serialization order is set identical to the one originally seen by T_j^t (except for the substitution of T_j^t with T_j^{xId} as the maximum element within the ordering). Hence, T_j^{xId} will exactly follow that original serialization order. Further, in case T_j^t originally had a reference order aligned with the optimistic delivery sequence, then T_j^{xId} substitutes T_j^t within the OptDelivered list. This reflects the fact that T_j^t is known not to be any longer in a serialization order compliant with the optimistic message delivery sequence, and that there is now a new incarnation of T_j, namely T_j^{xId}, aligned to that order.

Next, during the handling of the write event, the transaction T_i^s is removed from the reference order T_j^t.RefOrder, and in case T_j^t was waiting for the completion of a data-item version Y written by T_i^s, then a Resume event towards T_j^t is raised (as already pointed out this will lead T_j^t to re-execute the read operation in order to determine a different version to be accessed). On the other hand, the miss of the snapshot involving T_j^t, caused by the write operation performed by T_i^s on X, and the consequent change of the reference serialization order of T_j^t (via the elimination of T_i^s) needs to be reflected on all those transactions that have T_j^t within their reference order. This is accomplished via the recursive module propagateSnapshotMiss, which implements a logic similar to the one above discussed while handling the miss experienced by T_j^t. In particular this module allows the modification of the serialization orders of transactions potentially depending on T_j^t (again via the elimination of T_i^s from their reference order), while also allowing to speculatively explore the original serialization orders seen by those transactions via the spawn of additional speculative instances associated with the same transactional requests.

Transaction completion. Upon reaching the complete stage, transaction T_i^s marks the versions of the data-items belonging to its write-set as completed (which might cause the wake-up of transactions with pending read operations on these data). After, T_i^s remains waiting for the corresponding transaction T_i to be TO-delivered (namely, final delivered by the OAB), and to become the top standing element within the TODelivered queue. As it will be clearer in the following, this means that for any transaction T_j, which was TO-delivered before T_i, there exists a corresponding speculatively executed transaction T_j^* that has been already committed. Hence T_i^s can be safely validated (by verifying whether it has read data-items belonging to the latest committed snapshot) and a commit, or an abort, event can be accordingly generated.

Abort events. In case of an abort, the data-item versions produced by T_i^s are eliminated. Also, T_i^s is eliminated by the reference order of other transactions. If these transactions already read some version of a data-item written by T_i^s, they are aborted by raising an Abort event, which propagates across transactions having transitive read-from dependencies with the currently aborting one. On the other hand, if T_i^s is eliminated by the reference order of a transaction T_j^h that is waiting for T_i^s's completion due to the read on a Wip data-item Y (hence T_j^h has not yet developed a read-from dependence on the aborting transaction T_i^s), then a Resume event is raised towards T_j^h, which will lead to the selection of a different snapshot for materializing the read operation on Y.

Commit events. Upon a commit event for transaction T_i^k, its written data-item values are committed, and the transaction is removed from the set of active transactions ActiveXacts. Also, the corresponding transactional request T_i is removed from the queue TODelivered, thus allowing the subsequent element within the queue, if any, to become the newly top standing one, which, in turn, allows triggering the commit for the subsequent TO-delivered transactional request. Additionally, any instance T_i^* of speculative transaction associated with the request T_i is removed from the OptDelivered list, reflecting that the transactional request T_i does not belong any more to the not yet finalized ordering, namely the speculative ordering. The other instances of speculative transactions associated with T_i are aborted (since a single one of them needs to be reflected into the history of committed transactions). Then T_i^k is eliminated from all the speculative reference orders of other transactions, if present, which reflects that it is encapsulated within the special transaction T_α^ω, representative of the whole history of already committed transactions. Finally, all the transactions

that developed a read-from dependency on the committing transaction T_i^k are killed with a Validate event. This will allow them to verify whether, once known that the committed transaction history has changed, they are still executing along a consistent serialization path. In particular, the validation phase for a transaction simply checks whether the already read data-items are still compliant with the ones exposed along its own speculative reference order. In the negative case, the transaction is aborted and restarted. Similarly to the case of the abort, also the abort-retry recursively propagates along chains of transactions having developed read-from dependencies on the currently aborting one.

C. Protocol Properties

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

1. **1-Copy Serializability** [2] 1-Copy Serializability is ensured as transactions are committed at every site only upon a deterministic validation that is executed by all replicas in the same total order established by OAB service.
2. **Non-redundant speculation** [26] - This property ensures that no two speculative instances T_j^t, T_j^{xId} of the same transaction T_j observe the same snapshot. This follows by observing that ASAP activates a new speculative transaction T_j^{xId} only if it detects that a transaction T_j^t has missed a value written by a transaction T_i^s serialized before T_j^t according to T_j^t.RefOrder. In this case, T_j^{xId} will observe at least a data item version different from those observed by T_j^t since T_j^{xId} will not miss the value written by T_i^s. In fact when T_j^{xId} will perform the read operation on the data-item whose write operation by T_i^s caused the snapshot miss, T_j^{xId} will exactly select the version written by T_i^s given that it has inherited the serialization order originally associated with T_j^t. The latter phenomenon is true for all the transactions that are (recursively) spawned due to direct or transitive dependency on T_i^s. They will execute along a serialization order where the snapshot by T_j^t is not visible since it is replaced by the one provided by T_j^{xId}.
3. **Lock-freedom** [10] - Lock-freedom guarantees that there is always at least a thread to make progress, thus ruling out deadlock and livelock scenarios. In ASAP, lock-freedom 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.

D. Trade-offs

All the speculative protocols providing exploration of multiple serialization orders according to the actual conflicts among concurrent transactions are affected by exponential growth in terms of number of speculative instances of transactions that are generated. The latter happens in case the application exhibits an extremely high transaction conflict probability [26]. As also shown in [26], real benchmarks typically do not generate such conflict levels. Therefore, in the presented pseudo-code for ASAP we didn't implement any condition aimed at bounding the speculation degree of the protocol. Anyway, ASAP is straightforwardly extensible to support two simple mechanisms for limiting the level of speculation, which might result useful in very high conflict scenarios and/or in contexts where computational resource are limited: (i) Bounding the number of speculative instances of the same transaction spawned, which could be done by simply adding a condition that prevents to activate new speculative transactions of the same transactional request whenever that bound is hit; and (ii) Favoring speculation only along specific serialization orders, thus limiting the snapshot-miss propagation level within the recursive function propagateSnapshotMiss.

V. Simulation Study

The performance achievable by ASAP has been assessed via a simulation study in the context of Software Transactional Memory (STM) systems [15]. By leveraging on the proven concept of atomic and isolated transaction, STMs spare the programmers from the pitfalls of conventional manual lock-based synchronization, drastically simplifying the development of parallel and concurrent applications. Originally proposed to simplify concurrent programming in non-distributed environments, STM systems are being growingly employed in distributed settings, e.g. in cloud-oriented distributed data grids [27], [8] or in high performance computing environments [7]. A characterizing aspect of STM systems is that their mean transaction execution time is typically several orders of magnitude smaller than in traditional database systems [25], [20]. With such a reduced transaction execution time, the overlapping between replica synchronization and local transaction processing might result very limited, unless advanced speculative processing techniques were employed. Hence, we argue that STM systems represent a natural test-bed for the evaluation of speculative transactional replication protocols [26].

In order to ensure the representativeness of the simulated data access patterns, we relied on a trace-based approach. Traces were collected by running a widely used STM benchmark [14], namely RB-Tree. This benchmark performs repeated insertion, removal and search operations of a randomly chosen integer in a set of integers implemented as a red-black tree. The configuration used for benchmarking prevented to generate read-only transactions. This choice is motivated by the fact that, in a replicated system, read-only transactions can be processed locally without the need for any interaction and synchronization among the replicas. Hence, this type of transactions does not provide a workload that could be usefully exploited for assessing the efficiency of replication protocols like ASAP. The tracing process for RB-Tree has bee carried out on a machine equipped with 2 CPU-Cores at 2.53 GHz and 4 GB of RAM, running Mac OS X 10 as the operating system. The adopted STM layer for the tracing process was JVSTM [5]. On the other hand, the distributed system we have simulated is composed by 4 replicated STM processes, each hosted on top of a machine equipped with 32 CPU-cores running at the same speed as in the architecture used for the tracing process ([1]).

We show comparative results between the performance of ASAP with respect to other speculative and non-speculative approaches in the following two scenarios: (a) A scenario in which the network behavior does not ensure the spontaneous

[1] 32 CPU-cores can nowadays be considered a typical value for commodity servers.

order property. In this case, mismatches between the optimistic and the final delivery sequences may occur with non-minimal probability. In particular, we focus our study on three different values for the probability of mismatch, namely 10%, 20% and 40%. (b) A scenario in which the network is highly predictable, thus not generating reorder across optimistic and final delivery sequences.

The traffic of messages towards the AOB has been simulated via a synthetic generator that permitted to build network events with a given percentage of reordering (0%, 10%, 20%, 40%,) and given mean values for optimistic and final delivery latencies that have been set, respectively, to 500 microseconds and 2 milliseconds. These are typical values we have observed when running the Appia Group Communication System Toolkit on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a Gigabit Ethernet and using TCP at the transport layer.

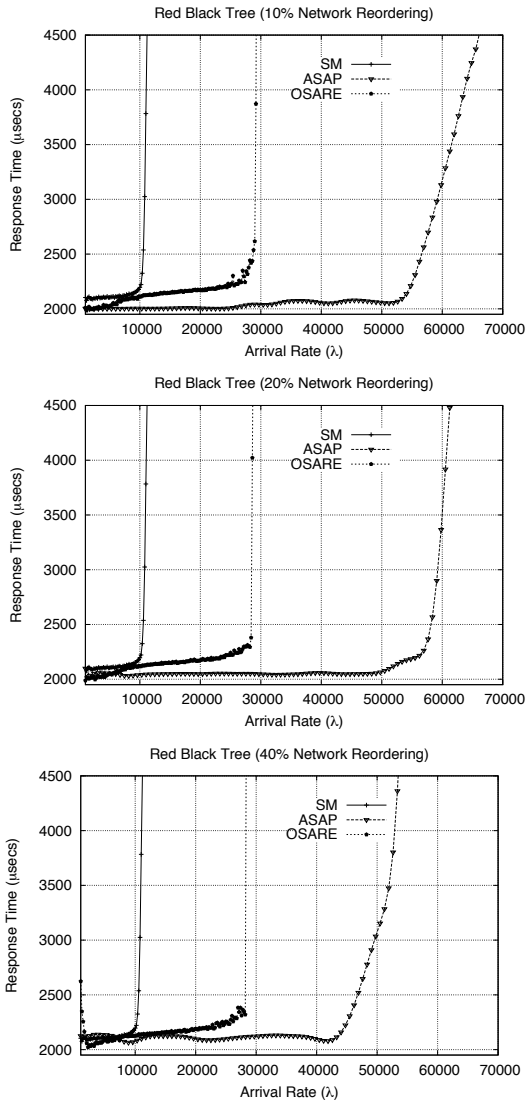

Fig. 3. Response Time for Networks with Reordering.

As for the scenario in point (a), we compare ASAP with the OSARE protocol recently presented in [19], and with a traditional, non-speculative, protocol, namely State Machine (SM) [21]. Analogously to ASAP, OSARE allows exploring multiple speculative serialization orders. Unlike ASAP, however, OSARE uses a lazy mechanism for tracking dependencies among transactions that is activated only upon the completion of speculative transactions (whereas ASAP traces dependencies as soon as they are generated, marking updated data versions as Wip). On the other hand, SM does not entail any speculative processing scheme, thus imposing to wait for the finalization of the total order protocol establishing the serialization order of a newly incoming request before executing the corresponding transaction.

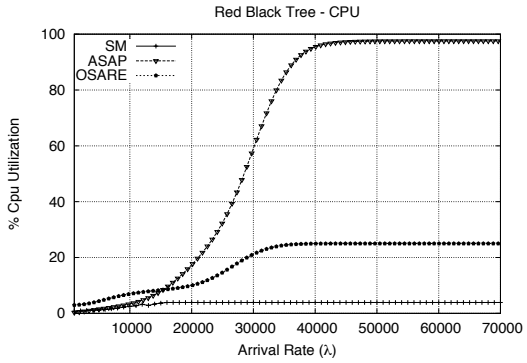

Fig. 4. CPU Usage.

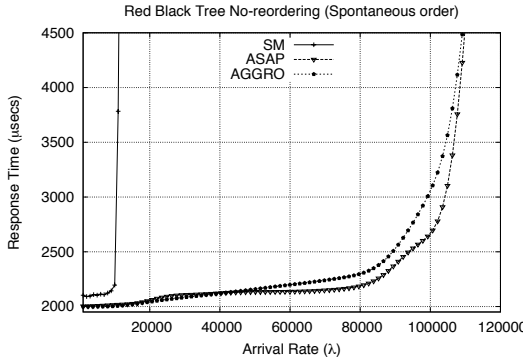

Fig. 5. Response Time for Predictable Networks.

As for the scenario in point (b), we compare ASAP with the AGGRO protocol presented in [18]. AGGRO employs an aggressive dependency propagation technique among speculative transactions that is similar in spirit to the one employed by ASAP. However, differently from ASAP, AGGRO is a speculative protocol explicitly tailored for networks ensuring spontaneous order (i.e. no reordering) since it only speculates along the optimistic delivery sequence.

The overall target of this study is to show that ASAP outperforms OSARE in scenario (a) and is not slower than AGGRO in scenario (b).

The plots in Figure 3 show the mean transaction response time, i.e. the average time since the TO-broadcast of a transaction till its commitment, for SM, OSARE and ASAP as a function of both the transactions' arrival rate and the network reordering probability. SM, which does not exploit any kind of speculation, exhibits a saturation point which is

independent of the percentage of network reordering and is less than 10k transactions per second. With 10% of network reordering, the saturation point for ASAP is reached when the transactions' arrival rate is more than 60k, while OSARE saturates at 30k. Similar, or slightly reduced gains are observed when the reordering percentage goes to 20% and 40%. Overall, ASAP allows on the order of 2x improvement of the system throughput wrt OSARE. Hence, the ability of ASAP to aggressively determine alternative speculative orders via early exposition of Wip data versions has a strong impact on the final achievable performance, in terms of (i) reduction of the average transaction finalization time, and (ii) increase of the maximum sustainable throughput.

In Figure 4 we report the CPU utilization caused by the different protocols for the case of 10% reordering in the OAB. The plots clearly show that the technique used by OSARE to detect conflicts only at transaction completion time can lead to resource under-utilization (in fact less than 30% of the CPU power is used) that can have a detrimental impact on performance, especially in case replicas are hosted on machines equipped with a high number of cores. On the contrary, the aggressive dependency propagation technique at the core of ASAP allows fruitful exploitation of the resources available, achieving up to 96% of CPU utilization in the simulated scenario.

As for scenario (b), whose results are shown in Figure 5, the early transaction spawning of ASAP and the early abort & restart mechanism used by AGGRO, can be seen as two different ways to reach the same target of speculatively materializing a serialization order compliant with the order of optimistic deliveries. This is reflected in the achieved performance, which is very similar for the two protocols.

VI. Conclusions

In this paper we presented ASAP, namely an Aggressive SpeculAtive Protocol for actively replicated transactional systems. By relying on both OAB and speculative processing techniques, ASAP aims at maximizing the overlap between replica coordination and local transaction processing activities. ASAP exploits an innovative speculative concurrency control, which allows tracing dependency among transactions in an early fashion. This maximizes the promptness of the exploration of alternative speculative serialization orders, thus enhancing the achievable level of overlapping between speculative computation phases and distributed coordination. This may result particularly useful in scenarios where the level of reordering exhibited by the OAB service is non-minimal, such as for poorly predictable networks or at very high concurrency levels. The effectiveness of our proposal has been assessed via a simulation study, whose results have shown that ASAP outperforms literature speculative and non-speculative protocols.

References

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Proc. of Euro-Par*, pages 496–503, London, UK, 1997. Springer-Verlag.

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

[3] A. Bestavros and S. Braoudakis. Value-cognizant speculative concurrency control. In *Proc. of VLDB*, pages 122–133, 1995.

[4] A. Brito, C. Fetzer, H. Sturzrehm, and P. Felber. Speculative out-of-order event processing with software transactional memory. In *Proc. of DEBS*, July 2008.

[5] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[6] N. Carvalho, P. Romano, and L. Rodrigues. Scert: Speculative certification in replicated software transactional memories. In *SYSTOR*, page 10, 2011.

[7] 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 *OOPSLA*. ACM, 2005.

[8] M. Couceiro, P. Romano, and L. Rodrigues. Polycert: polymorphic self-optimizing replication for in-memory transactional grids. In *Middleware '11*, pages 309–328. Springer-Verlag, 2011.

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

[10] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003.

[11] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the SIGMOD*, pages 173–182. ACM, 1996.

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

[13] J. R. Haritsa, K. Ramamritham, and R. Gupta. The prompt real-time commit protocol. *IEEE Trans. on Parallel and Distributed Systems*, 11:160–181, 2000.

[14] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC*. ACM, 2003.

[16] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000.

[17] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE TKDE*, 15(4):1018–1032, 2003.

[18] R. Palmieri, F. Quaglia, and P. Romano. AGGRO: Boosting STM replication via aggressively optimistic transaction processing. In *Proc. of NCA*, pages 20–27, Los Alamitos, CA, USA, 2010. IEEE Computer Society.

[19] R. Palmieri, F. Quaglia, and P. Romano. OSARE: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, pages 59–64, 2011.

[20] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *Proc. of DPDNS*, 2010.

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

[22] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1):79–101, 2003.

[23] C. Pu and A. Leff. Replica control in distributed systems: as asynchronous approach. In *Proc. of SIGMOD*, SIGMOD '91, pages 377–386. ACM, 1991.

[24] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE TKDE*, 16(2):154–169, 2004.

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

[26] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues. An optimal speculative transactional replication protocol. In *Proc. of ISPA*, pages 449–457, 2010.

[27] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo. Cloud-TM: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44(2), Apr. 2010.

[28] P. Romano, D. Rughetti, F. Quaglia, and B. Ciciani. APART: Low cost active replication for multi-tier data acquisition systems. In *Proc. of NCA*, pages 1–8. IEEE Computer Society, 2008.

[29] F. B. Schneider. *Replication management using the state-machine approach*. ACM Press/Addison-Wesley Publishing Co., 1993.