OSARE: Opportunistic Speculation in Actively REplicated Transactional Systems

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

Paolo Romano
INESC-ID, Lisbon, Portugal

***Abstract*—In this work we present OSARE, an active replication protocol for transactional systems that combines the usage of Optimistic Atomic Broadcast with a speculative concurrency control mechanism in order to overlap transaction processing and replica synchronization. OSARE biases the speculative serialization of transactions towards an order aligned with the optimistic message delivery order. However, due to the lock-free nature of its concurrency control algorithm, at high concurrency levels, namely when the probability of mismatches between optimistic and final deliveries is higher, the chances of exploring alternative transaction serialization orders increase correspondingly. This is achieved by OSARE in an opportunistic and lightweight fashion. A simulation study we carried out in the context of Software Transactional Memory systems shows that OSARE achieves robust performance also in scenarios characterized by non-minimal likelihood of reorder between optimistic and final deliveries, providing remarkable speed-up with respect to state of the art speculative replication protocols.**

I. Introduction

Active replication [24] is a classical means for providing fault-tolerance and high availability. It relies on consensus among the replicas on a common total order for the processing of incoming requests. The non-blocking establishment of the agreed upon total order is typically encapsulated by a so called Atomic Broadcast (AB) group communication primitive [6].

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 [13]. According to this technique, the spontaneous network delivery order is used as an early, although possibly erroneous guess of the total delivery order of messages eventually defined via AB. This idea is encapsulated by the Optimistic Atomic Broadcast (OAB) primitive [13], representing a variant of AB in which the notification of the final message delivery order is preceded by an *optimistic message delivery* indication, typically available after a single communication step. By activating transactions' processing upon their optimistic delivery, rather than waiting for the final order to be established, OAB-based replication techniques overlap the (otherwise sequential) replica synchronization and local computation phases. However, serializing transactions according to the optimistic delivery order does not pay-off in case of non-minimal likelihood of mismatch between optimistic and final message ordering. In such a case, optimistically processed transactions may have to be aborted and restarted right after OAB completion, thus nullifying any performance gain associated with their early activation.

In order to cope with the above issue, we present a novel active replication protocol for transactional systems based on an *opportunistic* paradigm, which we name OSARE - Opportunistic Speculation in Active REplication. OSARE maximizes the overlap between replica coordination and transaction execution phases by propagating, in a speculative fashion, the (uncommitted) post-images of completely processed, but not yet finally delivered, transactions along chains of conflicting transactions. Also, speculation attempts to serialize any transaction after those that have preceded it within the optimistic delivery order. In case the miss of some write is experienced along the execution path, which we refer to as a *snapshot-miss*, the materialized serialization order is opportunistically kept alive, with the aim of increasing the likelihood of matching the final order established by the OAB service in case it reveals not aligned with the optimistic delivery sequence. Further, if a transaction T experiences a snapshot miss, OSARE reactivates a new instance of T (and, recursively, of the transactions having developed a read-from dependency from T), thus biasing the speculative exploration towards a serialization order compliant with the optimistic message delivery order.

Interestingly, 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. 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 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 a trace driven simulation study in the context of Software Transactional Memory (STM) systems, showing that response-time speedup of 160% can be achieved compared to recent proposals, such as [16], that systematically entail speculative transaction processing (even along chains of conflicting transactions), but that materialize speculation exclusively along the optimistic delivery sequence.

II. Related Work

Literature proposals targeting transactional systems' replication entail protocol specification (see, e.g., [7], [12], [20]) as well as replication architectures that have been based on middleware level approaches (see, e.g., [14], [18], [19]) and/or on extensions of the inner logic of transactional systems (see, e.g., [12], [26]). As shown in [25], 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, (O)AB primitives are exploited to coordinate processing activities by determining, in a non-blocking fashion, a global transaction serialization order, thus circumventing scalability problems that are known to affect classical eager replication mechanisms based on distributed locking and atomic commit protocols [7]. Relevant proposals along this direction can be found in, e.g., [1], [13]. Differently from OSARE, 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], [11]), or execute at most a single optimistically delivered transaction along the conflicting transactions chain, before the OAB gets completed (see [13]). Also, the latter protocols require a-priori knowledge of transactions' data accesses since each speculatively executed transaction needs to pre-acquire locks on its whole data-set. Conversely OSARE adopts an optimistic transaction scheduling approach that does not require a-priori knowledge of data access patterns.

Like OSARE, our recent works in [16] and [22] both make use of speculation along chains of conflicting transactions. The work in [16] 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, thus better fitting scenarios of mismatch between optimistic and final ordering by the OAB service. The proposal in [22] 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 final delivery order. Hence, differently from OSARE, there is no set of "preferential" serialization orders to be opportunistically processed in a speculative fashion.

OSARE also exhibits relations with replication approaches for main memory database systems (see, e.g., [3], [23]). These proposals differ from OSARE since they either target primary-backup replication schemes [3] or data-partitioned cluster-based systems [23], while our targets are actively, fully replicated systems.

III. System Model

We consider a classical distributed system model [9] 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. A formal specification of the properties ensured by the OAB service can be found in [21].

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.

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 a data item X. On the other hand, other versions can be 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.

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* [22] 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 S, defined as the set of values returned by all its read operations, it behaves deterministically.

The manipulation of the data items occurs via the primitives `setComplete(`X^T`,`T`)`, which marks a data item version X^T written by transaction T as complete, and `unsetComplete(`X^T`,`T`)`, which removes a complete data item version X^T exposed by transaction T.

IV. The OSARE Protocol

A. Protocol Notations and Data Structures

The OAB service delivers transactions, each of which is denoted as T_i. The delivered transactions are however never directly executed by XM, which only executes speculative transaction instances, denoted using the notation T_i^j.

Each speculative transaction T_i^j keeps track of its own *serialization view*, defined as the totally ordered sequence of transactions that are expected to be serialized before T_i^j. 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 OptDelivered, accessible by all the transactional threads, which maintains the identifiers of the transactions whose speculative serialization view is aligned with the order of optimistic deliveries; a local list of speculative transaction identifiers, referred to as T_i^j.SpeculativeOrder, which is associated with the transactional thread handling transaction T_i^j. The sequence of speculative transactions recorded within T_i^j.SpeculativeOrder 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. This determines a history of speculative transactions whose snapshots may be visible by T_i^j's read operations.

We use the notation $T_k^h \overset{T_i^j}{\rightarrow} T_s^t$ to indicate that T_k^h precedes T_s^t within the ordered list T_i^j.SpeculativeOrder. This expresses that, according to the view of T_i^j: i) T_k^h and T_s^t belong to the same speculative history of transactions; ii) T_k^h and T_s^t are both expected to be serialized before T_i^j; iii) 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 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.SpeculativeOrder. Always by convention, T_i^j represents the maximum element of the $\overset{T_i^j}{\rightarrow}$ relation. Overall, denoting with $(T_{k_1}^{h_1}, \ldots, T_{k_n}^{h_n})$ the sequence of transactions belonging to T_i^j.SpeculativeOrder, we have: $T_\alpha^\omega \overset{T_i^j}{\rightarrow} T_{k_1}^{h_1} \overset{T_i^j}{\rightarrow} \ldots \overset{T_i^j}{\rightarrow} T_{k_n}^{h_n} \overset{T_i^j}{\rightarrow} T_i^j$.

B. Protocol Logic

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

Optimistic delivery of transactions. Upon the Opt-deliver event of a transaction T_i, XM instantiates a speculative transaction T_i^0, and then sets up its serialization order by copying the current content of OptDelivered into T_i^0.SpeculativeOrder. Next, XM appends T_i^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, XM activates the processing activities for T_i^0 by invoking ActivateSpeculativeTransaction (which also adds T_i^0 to the set of active transactions ActiveXacts).

Handling of read and write operations. When a transaction T_i^s issues a read operation on a data item X, XM verifies whether a version of X belongs to the write set of the reading transaction. In the positive case, the written value is simply returned. Instead, if T_i^s has not previously issued a write on X, the precedence relation $\overset{T_i^s}{\rightarrow}$ is used to determine which version of X should be seen by T_i^s. To this end, the most recent version exposed by a completed or committed transaction, according to the serialization view of T_i^s, is identified. This is done by determining the maximum speculative transaction T_j^t preceding T_i^s according to the $\overset{T_i^s}{\rightarrow}$ relation, which has i) written X and ii) already completed its execution.

As for the write operation of a transaction T_i^s on a data item X, 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^s are in fact made all atomically visible only once that T_i^s reaches completion.

Completion of speculative transactions. When the Complete method is executed by XM for transaction T_i^s, each data item version created (i.e. written) by T_i^s is made speculatively visible by setting its state to the complete value. Before making the snapshot produced by T_i^s visible, however, it is first checked whether every transaction T_j^t that, according to its serialization view, is serialized after T_i^s, is still correctly executing along that order, or it 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^t has already issued a read operation, and, ii) T_i^s is the last speculative transaction to have written X among those in T_j^t's speculative view. In this case, in fact, T_j^t 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. The following three actions are taken to handle a snapshot-miss event:

```
OrderedList<Transaction> TODelivered, OptDelivered;
Set<Transaction> ActiveXacts;

upon Opt-deliver(Transaction $T_i$) do
  $T_i^0$ =$T_i$.createNewSpecXact();
  $T_i^0$.SpeculativeOrder = copy(OptDelivered);
  OptDelivered.enqueue($T_i^0$);
  ActivateSpeculativeTransaction($T_i^0$);

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

DataItemValue Read(Transaction $T_i^s$, DataItem $X$)
  if ($X \in T_i^s$.WriteSet) return $T_i^s$.WriteSet.get(X).value;
  select version of $X$ completed or committed by $T_j^t = max\{T_j^f \overset{T_i^s}{\rightarrow} T_i^s\}$;
  // the committed version is written by $T_\alpha^\omega$ by definition
  $T_i^s$.ReadSet.add($X$);
  $T_i^s$.ReadFrom.add($T_j^t$);
  return $T_i^s$.ReadSet.get($X$).value

void Write(Transaction $T_i^s$, DataItem $X$, Value $v$)
  if ($X \in T_i^s$.WriteSet) $T_i^s$.WriteSet.update($X, v$) ;
  else $T_i^s$.WriteSet.add($X, v$);

void Complete(Transaction $T_i^s$)
  atomically do
    $T_i^s$.isCompleted = TRUE
    $\forall X \in T_i^s$.WriteSet do setComplete($X, T_i^s$);
    $\forall\ T_j^t$ s.t. ($\exists X \in T_j^t$.ReadSet: ($X \in T_i^s$.WriteSet  and
      $T_i^s = max\{T_l^f : T_l^f \overset{T_j^t}{\rightarrow} T_j^t$ exposing a complete version of $X\}$) do
        $T_j^{xId}$ =$T_j$.createNewSpecXact();
        $T_j^{xId}$.SpeculativeOrder = copy($T_j^t$.SpeculativeOrder);
        $T_j^t$.SpeculativeOrder.remove($T_i^s$); / reflects the snapshot-miss of $T_j^t$
        if ($T_j^t \in OptDelivered$) OptDelivered.replace($T_j^t$, $T_j^{xId}$);
        wave($T_j^t$,$T_j^{xId}$,$T_i^s$);
  wait until TODelivered.topStanding == $T_i$;
  if ($\forall X \in T_i^s$.ReadSet: $X$.version == LatestCommitted) $T_i^s$.RaiseEvent(Commit);
  else $T_i^s$.RaiseEvent(Abort);

void wave(Transaction $T_j^t$, Transaction $T_j^{xId}$, Transaction $T_i^s$)
  $\forall T_l^f$ s.t. $T_j^t \in T_l^f$.ReadFrom do
    $T_l^{xId'}$ =$T_l$.createNewSpecXact();
    $T_l^{xId'}$.SpeculativeOrder = copy($T_l^f$.SpeculativeOrder);
    $T_l^{xId'}$.SpeculativeOrder.replace($T_j^t$,$T_j^{xId}$);
    $T_l^f$.SpeculativeOrder.remove($T_i^s$);
    if ($T_l^f \in OptDelivered$) OptDelivered.replace($T_l^f$, $T_l^{xId'}$);
    wave($T_l^f$,$T_l^{xId'}$,$T_i^s$)
  $\forall T_l^g$ s.t. ($T_j^t \in T_l^g$.SpeculativeOrder and $T_j^t \notin T_l^g$.ReadFrom) do
    $T_l^g$.SpeculativeOrder.replace($T_j^t$,$T_j^{xId}$);
  ActivateSpeculativeTransaction($T_j^{xId}$);
```

Fig. 1. Behavior of XM (Part A).

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

upon Abort(Transaction $T_i^s$) do atomically
   $\forall X \in T_i^s$.WriteSet do unsetComplete($X, T_i^s$);
   $\forall T_j^h \in$ActiveXacts $s.t.\ j \neq i$ and $T_i^s \in T_j^h$.SpeculativeOrder do
      $T_j^h$.SpeculativeOrder.remove($T_i^s$);
   $\forall T_j^h$ s.t. $T_i^s \in T_j^h$.ReadFrom do $T_j^h$.RaiseEvent(Abort);
   ActiveXacts.remove($T_i^s$);

upon Commit(Transaction $T_i^k$) do atomically
   ActiveXacts.Remove($T_i^k$);
   $\forall X \in T_i^k$.WriteSet do $T_i^k$.WriteSet.Commit($X$);
   TODelivered.Dequeue($T_i$);
   OptDelivered.Remove($T_i^*$);
   $\forall T_i^h \in$ActiveXacts $s.t.\ h \neq k$ do $T_i^h$.RaiseEvent(Abort);
   $\forall T_j^h \in$ActiveXacts $s.t.\ j \neq i$ and $T_i^k \in T_j^h$.SpeculativeOrder do
      $T_j^h$.SpeculativeOrder.remove($T_i^k$);
   $\forall T_j^h \in$ActiveXacts $s.t.\ T_i^k \in T_j^h$.ReadFrom do $T_j^h$.RaiseEvent(Validate);

upon Validate(Transaction $T_i^k$) do
   $\forall X \in T_i^k$.ReadSet do

      compute $T_j^h = max\{T_l^f : T_l^f \overset{T_i^k}{\rightarrow} T_i^k$ and $X \in T_l^f$.WriteSet$\}$;
      if ($T_i^k$.ReadSet.get($X$).Creator $\neq T_j^h$)
         $T_i^k$.RaiseEvent(AbortRetry);
         break;

upon AbortRetry(Transaction $T_i^s$) do atomically
   $\forall X \in T_i^s$.WriteSet do unsetComplete($X, T_i^s$);
   $\forall T_j^h$ s.t. $T_i^s \in T_j^h$.ReadFrom do $T_j^h$.RaiseEvent(AbortRetry);
   restart transaction $T_i^s$;
```

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

1. A new speculative instance T_j^{xId} is activated, setting its serialization view to the one currently associated with T_j^t. Given that T_i^s has now reached completion, the new instance T_j^{xId} is guaranteed not to miss the snapshot produced by T_i^s.
2. The serialization order of T_j^t is then updated by removing T_i^s (namely the transaction whose write operation has been missed by T_j^t) from T_j^t.SpeculativeOrder. Next, if T_j^t was originally recorded within OptDelivered, it is replaced by T_j^{xId} within this list. This reflects the fact that T_j^t is known not to be any longer in a serialization order compliant with that of the optimistic message delivery, and that there is now a new incarnation of T_j, namely T_j^{xId}, aligned to that 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^t involved in the snapshot-miss event.

After having handled all the snapshot-miss events detected upon its completion, T_i^s simply remains waiting for the corresponding transaction T_i to be TO-delivered, 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 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 this speculative transaction.

Recursive propagation of snapshot-miss events. As we have just explained, the completion of a transaction T_i^s can trigger a series of snapshot-miss events involving transactions T_j^t, for which a new speculative instance of transaction T_j, namely T_j^{xId} is activated, which will be guaranteed not to miss the snapshot created by T_i^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^t, in fact, may have already completed its execution and exposed its snapshot to a different speculative transaction, say T_l^f. In this case, even if T_l^f did not miss the snapshot generated by T_i^s, it is still transitively involved by the snapshot-miss event affecting T_j^t. Analogously to T_j^t, therefore, T_i^s needs to be removed by the speculative view of T_l^f. Further, in order to pursue the exploration of a serialization order compliant with the optimistic message delivery order, a new speculative instance of T_l, namely $T_l^{xId'}$, needs to be activated, which should now include T_j^{xId} in its serialization view. Finally, just like in the Complete method, it is verified if T_l^f 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_l^f with $T_l^{xId'}$, so to reflect the fact that the latter one is now 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 relation from T_j^t.

Upon returning from the recursive call, XM substitutes T_j^t with T_j^{xId} in the speculative view of every transaction T_l^g that i) contained T_j^t in its speculative view, and that ii) did not develop a read-from dependency from T_j^t. This is necessary in case T_l^g is still active, in order to ensure that during its subsequent reads, it will be able to observe the snapshot generated by T_j^{xId}, thus correctly realigning T_l^g'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^{xId} takes place right before returning from wave.

Final delivery of transactions. The logic for handling final delivery events 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, which allows all the replicas to 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 ActiveXacts 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.

Slightly more sophisticated is the handling of the commit event. In this case, the committing transaction identifier T_i^k

is removed from ActiveXacts, 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^* 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^k 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 an instance of T_i different from T_i^k. T_i^k 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^k is moved to the committed history. Therefore we need to verify whether the transactions exhibiting dependencies on the snapshot produced by T_i^k 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^k has been moved to the committed history (see the AbortRetry module).

C. Considerations

For space constraints we cannot provide details on the correctness of our protocol, which can be found in [17]. Anyway, OSARE guarantees opacity [8], 1-copy serializability and lock freedom. It also ensures non-redundant speculation [22], with the meaning that no two different speculative instances of a same transaction observe the same snapshot.

As for resource demand, we note that the protocol could be complemented by an admission control scheme aimed at bounding the number of speculatively executed transactions such in a way to prevent system saturation. This would lead to tradeoffs between the actual degree of speculation and the final delivered performance.

V. Simulation Study

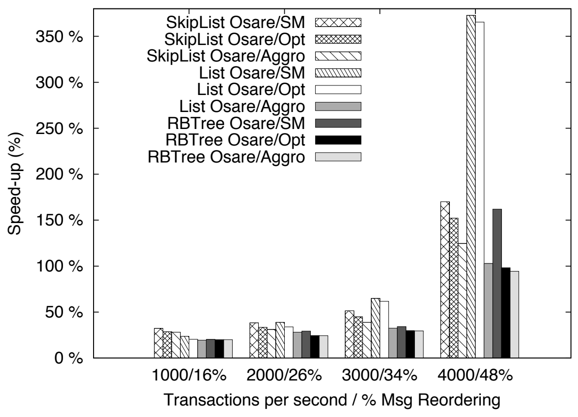
In order to assess the performance of OSARE we developed detailed simulation models for the following protocols: OSARE, AGGRO [16] and Opt [13], all relying on OAB, and traditional State Machine (SM), relying on AB. AGGRO and Opt are baseline protocols for the evaluation of OSARE since they entail some form of speculation, either limited to conflicting transaction chains of length one, or entailing multiple conflicting transactions along the chain. SM acts as a reference for assessing the performance of speculative vs non-speculative protocols.

In order to accurately model transactional execution dynamics we collected traces using a number of heterogenous STM benchmarks, namely: i) three microbenchmarks, Red Black Tree, List and SkipList, that have been adopted in a number of performance evaluation studies of STM systems [2], [5], [10]; and ii) two benchmarks of the STAMP suite [4], namely Yada and Labyrinth++. The above benchmarks were configured not to generate any read-only transaction. This choice depends on the fact that, in all the protocols considered in this study, read-only transactions can be executed locally, without the need for distributed coordination. By only considering update transactions, we can therefore precisely assess the impact of distributed coordination on the performance of the replicated system, as well as the performance gains achievable by OSARE. The machine used for the tracing process is equipped with an Intel Core 2 Duo 2.53 GHz processor and 4GB of RAM, running Mac OS X 10.6.2 and JVSTM [2]. The simulation model of the replicated system comprises a set of 4 replicated STM processes, each hosted by a machine equipped with 32-cores processing transactions at the same speed as in the above architecture.

The transactions' arrival process via optimistic and final message deliveries is also trace-driven. Specifically we use traces generated by running a sequencer based (O)AB implementation available in the Appia GCS Toolkit [15] on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a Gigabit Ethernet and using TCP at the transport layer. We injected in the system messages of 512 bytes (largely sufficient to encode the parameters of the transactional methods exposed by the considered STM benchmarks) with an exponentially distributed arrival rate having mean λ. We treat λ as the independent parameter of our study, letting it vary in the range [1000,4000] messages per second, thus expressing from low/moderate up to high load to be sustained by the GCS. As expected, the mismatch between optimistic and final delivery orders (or message reordering for the sake of brevity) increases along with the message arrival rate, ranging from 16%, at 1000 msgs/sec, up to 48%, at 4000 msgs/sec.

The plots in Figure 3 report the speed-up achieved by OSARE vs the other protocols, evaluated as the percentage of additional latency for executing a transaction (being the latency the average time since the TO-broadcast of a transaction till its commitment) in any of these protocols with respect to OSARE. The data highlight striking performance gains by OSARE compared to AGGRO, which increase (up to around 160%) as the load and the message reordering grow. This is due to the fact that, by opportunistically processing a transaction in multiple serialization orders, OSARE overlaps more effectively processing and communication. On the other hand, the gains over Opt (which unlike OSARE and AGGRO does not speculate along chains of conflicting transactions) and SM are even larger, being on the order of up to 350/360%.

Comparing more closely OSARE and AGGRO, which both speculate along chains of conflicting transactions, the number of transactions that have already started (or completed) along



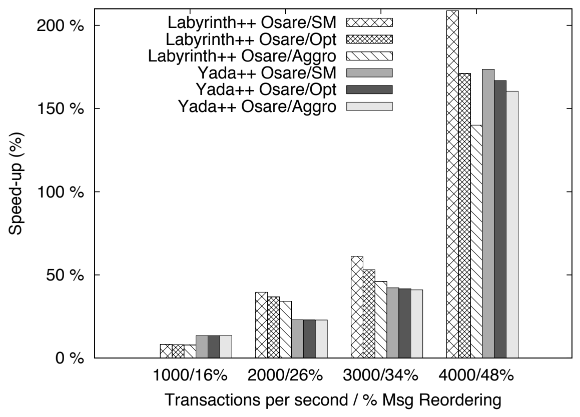


Fig. 3. Speed-up by OSARE.

a serialization order compliant with the OAB final delivery sequence is up to 50% higher in OSARE than in AGGRO. Also, at high load, the number of transactions aborted in AGGRO is around 4x larger than in OSARE. This depends on that AGGRO uses an aggressive rollback-retry mechanism which re-activate transactions as soon as they are detected not to be serialized according to the optimistic delivery order. This policy pays off at negligible levels of message reordering. On the other hand, as soon the probability of message reordering becomes non-minimal, AGGRO incurs in a significant waste of computation, which is 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 (exhibiting about 48% of message reordering), the average number of speculative instances activated by OSARE for a given transaction (across all the evaluated benchmarks) is 2.7. In other words, beyond the serialization order associated with the final delivery order, only 1.7 additional serialization orders are explored for each transaction. Also, our experimental data show that, on average, the corresponding CPU utilization is less than 8% with OSARE on the simulated hardware architecture. Overall, we can deduce that the speculative approach provided by OSARE is perfectly sustainable by off-the-shelf multi-core and many-core architectures, at least when considering scenarios resembling the simulated settings.

References

[1] D. Agrawal, G. Alonso, A. E. Abbadi, and I. Stanoi. Exploiting atomic broadcast in replicated databases (extended abstract). In *Euro-Par*, pages 496–503, 1997. Springer-Verlag.
[2] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.
[3] L. J. Camargos, F. Pedone, and R. Schmidt. A primary-backup protocol for in-memory database replication. In *NCA*, pages 204–211, 2006.
[4] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.
[5] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D^{2}STM: Dependable Distributed Software Transactional Memory. In *PRDC*, pages 307–313, 2009. IEEE Computer Society.
[6] X. Defago, A. Schiper, and P. Urban. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Computing Surveys*, 36(4):372–421, 2004.
[7] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *SIGMOD*, pages 173–182, 1996. ACM.
[8] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPOPP*, pages 175-184, 2008. ACM.
[9] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming*. Springer, 2006.
[10] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.
[11] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000. IEEE Computer Society.
[12] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, a new way to implement database replication. In *VLDB*, pages 134–143, 2000. Morgan Kaufmann.
[13] B. Kemme, F. Pedone, G. Alonso, A. Schiper, and M. Wiesmann. Using optimistic atomic broadcast in transaction processing systems. *IEEE Trans. Knowl. Data Eng.*, 15(4):1018–1032, 2003.
[14] Y. Lin, B. Kemme, M. Patiño-Martínez, and R. Jiménez-Peris. Middleware based data replication providing snapshot isolation. In *SIGMOD*, pages 419–430, 2005.
[15] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *ICDCS*, pages 707–710, 2001. IEEE Computer Society.
[16] R. Palmieri, F. Quaglia, and P. Romano. AGGRO: Boosting stm replication via aggressively optimistic transaction processing. In *NCA*, pages 20–27, 2010. IEEE Computer Society.
[17] R. Palmieri, F. Quaglia, and P. Romano. OSARE: Opportunistic speculation in actively replicated transactional systems. Technical Report 2, INESC-ID, January 2011.
[18] M. Patiño-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. Comput. Syst.*, 23(4):375–423, 2005.
[19] F. Pedone and S. Frølund. Pronto: High availability for standard off-the-shelf databases. *J. Parallel Distrib. Comput.*, 68(2):150–164, 2008.
[20] F. Pedone, R. Guerraoui, and A. Schiper. The database state machine approach. *Distributed and Parallel Databases*, 14(1):71–98, 2003.
[21] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theor. Comput. Sci.*, 291(1):79–101, 2003.
[22] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues. An optimal speculative transactional replication protocol. In *ISPA*, pages 449–457, 2010. IEEE Computer Society.
[23] R. Schmidt and F. Pedone. Consistent main-memory database federations under deferred disk writes. In *SRDS*, pages 85–94, 2005. IEEE Computer Society.
[24] F. B. Schneider. *Replication management using the state-machine approach.* ACM Press/Addison-Wesley Publishing Co., 1993.
[25] M. Wiesmann and A. Schiper. Comparison of database replication techniques based on total order broadcast. *IEEE Trans. Knowl. Data Eng.*, 17(4):551–566, 2005.
[26] S. Wu and B. Kemme. Postgres-r(si): Combining replica control with concurrency control based on snapshot isolation. In *ICDE*, pages 422–433, 2005. IEEE Computer Society.