AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing

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

Paolo Romano
INESC-ID, Lisbon, Portugal

Abstract

Software Transactional Memories (STMs) are emerging as a potentially disruptive programming model. In this paper we are address the issue of how to enhance dependability of STM systems via replication. In particular we present AGGRO, an innovative Optimistic Atomic Broadcast-based (OAB) active replication protocol that aims at maximizing the overlap between communication and processing through a novel AGGRessively Optimistic concurrency control scheme. The key idea underlying AGGRO is to propagate dependencies across uncommitted transactions in a controlled manner, namely according to a serialization order compliant with the optimistic message delivery order provided by the OAB service. Another relevant distinguishing feature of AGGRO is of not requiring a-priori knowledge about read/write sets of transactions, but rather to detect and handle conflicts dynamically, i.e. as soon (and only if) they materialize. Based on a detailed simulation study we show the striking performance gains achievable by AGGRO (up to 6x increase of the maximum sustainable throughput, and 75% response time reduction) compared to literature approaches for active replication of transactional systems.

1 Introduction

Software Transactional Memories (STMs) are emerging as a highly attractive and potentially disruptive programming paradigm. Leveraging on the proven concept of atomic and isolated transaction, STMs spare programmers from the pitfalls of conventional manual lock-based synchronization, significantly simplifying the development of parallel and concurrent applications. However, as STMs start making their way out of research labs and being used in real-life systems (see, e.g., the FenixEDU system [5]), they are faced with dependability challenges which cannot be efficiently tackled by existing replication-based schemes.

State of the art solutions for the replication of transactional systems [1, 14, 21] have in fact been targeted to traditional database systems. The fulcrum of these solutions is the synergic integration of an Atomic Broadcast (AB) service [11], ensuring replicas' agreement on the global serialization order (GSO) of transactions, and a local, deterministic, concurrency control scheme, guaranteeing that transaction scheduling at each replica matches the GSO output by the AB service.

However, as highlighted in [24], transaction execution times in (non-replicated) STM systems are typically several orders of magnitude smaller than in conventional database environments, leading to an amplification of the relative cost of distributed replica coordination schemes. Not only this has a significant negative effect on the transaction completion time. In STMs, in fact, being communication costs relatively higher than in database environments, they also induce relatively longer periods of stall for the local processing activities. This can cause severe under-utilization of the available computing resources, especially in modern massively parallel architectures.

This suggests the idea of optimistically processing transactions without waiting for the completion of the (AB-based) replica coordination scheme, improving efficiency by overlapping transaction processing and communication. Such an idea has been already exploited, to some extent, in the context of actively replicated database systems [14, 20] by leveraging on, so called, Optimistic Atomic Broadcast (OAB) services [22]. An OAB service provides early knowledge about message existence (via a so called optimistic delivery phase) and early indications of the corresponding final delivery order. As shown in [14], in fact, in typical LANs, the network normally ensures the, so called, spontaneous order property, i.e. high probability of matching between optimistic and final delivery orders.

In the aforementioned database replication schemes, optimistic delivery order indications are used to guess the final GSO and to optimistically activate transaction processing in a compliant serialization order. The mechanisms employed to ensure deterministic transaction scheduling are based on the atomic pre-acquisition of locks on the data items to be accessed by the transactions. This ensures that conflicting transactions are sequentially executed in an order compliant with the optimistically guessed GSO, but demands the a-priori knowledge of the data items to be accessed by a transaction (or, equivalently, of transaction conflict classes). If the optimistic delivery order of an activated transaction T

does not contradict the final GSO, any work carried out by T before the notification of the final GSO has been usefully anticipated, yielding an effective overlap between coordination and processing phases.

Unfortunately, when employed in the context of STM-based systems, these mechanisms suffer from two main drawbacks: (1) Due to the difficulty to exactly identify the data items to be accessed by transactions before these are actually executed, it is typically necessary to adopt conservative conflict assumptions based on coarse data granularity, e.g. whole, or large slices of, database tables [20]. However, unlike relational database systems, STM-based applications are characterized by arbitrary memory layouts and access patterns which may make significantly harder, or even impossible, to a-priori identify, with a reasonable accuracy, the boundaries of the memory regions that will be accessed by transactions prior to their actual execution. On the other hand, gross over-estimations of the actual transaction conflicts can strongly hamper concurrency, leading to significant resources' under-utilization, especially in (massively) parallel systems. (2) These approaches exhibit a limited degree of optimism since they process serially every optimistically delivered transaction that is known to conflict with another already activated transaction. While such a choice prevents cascading abort, it also strongly limits concurrency, which may significantly hamper performance. As we have also shown in [19], this is particularly true in STM scenarios where, being the transaction execution time typically much lower than the OAB finalization latency, the performance benefits achievable by optimistically executing *at most one* among a set of conflicting transactions are significantly slimmer than in conventional database settings.

To overcome the above limitations, in this paper we present AGGRO, an AGGRessively Optimistic replication protocol specifically tailored to STM systems. The key idea behind AGGRO is to seek maximum overlap between replica coordination and transaction execution phases by propagating the (uncommitted) post-images of complete, but not yet finally delivered, transactions across chains of conflicting transactions speculatively executed in a serialization order compliant with the optimistic delivery order. To ensure that the actual transaction schedule matches the serialization order determined by the sequence of optimistic deliveries, AGGRO relies on an innovative concurrency control mechanism that, unlike existing OAB-based replication approaches, does not require information on the transactions' data access patterns prior to their actual execution. Conversely, it detects any possible discrepancy between the transaction schedule and the optimistic delivery order a posteriori, namely as soon as (and if) conflicts materialize.

As we will show by means of a detailed simulation study, AGGRO allows achieving up to 75% reduction of the transaction execution latency and 6x throughput increase with respect to state of the art OAB-based replication schemes when used to handle STM applications deployed on replicas equipped with an eight-core CPU (which today represents a typical configuration for commodity server systems). Such performance gains are obtained without sacrificing consistency. In fact, beyond ensuring 1-copy serializability, AGGRO also enforces opacity [10] by guaranteeing that the snapshot observed by any (eventually committed or aborted) transaction is always equivalent to one generated by a serial schedule, albeit possibly not matching the one associated with neither the optimistic nor the final delivery order.

The remainder of this paper is structured as follows. In Section 2 we discuss related work. The target system model for AGGRO is defined in Section 3. The AGGRO protocol is presented in Section 4. The results of the simulation study are provided in Section 5.

2 Related Work

The use of Atomic Broadcast (AB) primitives to support replication of transactional systems has been widely explored in literature, especially for what concerns database systems (see, e.g., [1, 14, 21]). The key idea underlying these approaches is to exploit AB to determine, in a non-blocking fashion, a global transaction serialization order (across all replicas), so to circumvent scalability problems that are known to affect classical eager replication mechanisms based on distributed locking and atomic commit protocols [9]. AGGRO builds on Optimistic Atomic Broadcast (OAB) primitives [14, 22] and, compared to previous works, relies on a much more aggressively optimistic local concurrency control mechanism. Additionally, unlike traditional OAB-based protocols [14, 20], AGGRO does not require a-priori knowledge of the data items to be accessed by transactions.

Our work is also related to the approaches in [3] and [23], which explored the idea of speculatively executing transactions to enhance performance of database systems. 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 [23] targets distributed databases relying on distributed locking and on a final atomic commit phase for validating transactions. The substantial difference between our work and the aforementioned solutions is that AGGRO provides supports for *replication* of (software) transactional memory systems, ensuring strong consistency despite the crash of (a subset) of the replicas. Further, AGGRO executes transactions speculatively in a serialization order compliant with the optimistic delivery order defined by the OAB service. Conversely, the solutions in [24, 25] speculate across different serialization orders, thus coping with the complementary case where spontaneous ordering does not hold.

AGGRO is similar in spirit also to the PROMPT protocol [12]. The latter protocol targets distributed databases in which the atomicity of update transactions is supported by means of an atomic commit protocol [2] (e.g. 2PC). The key idea underlying PROMPT is to make the post-images generated by pre-committed transactions immediately available to at-most one conflicting transaction. This technique allows reducing the stalls incurred in by transactions conflicting with pre-committed transactions. Analogously, AGGRO exposes the post-images of uncommitted transactions with the purpose of overlapping processing with communication. On the other hand, being layered on top of an OAB service, AGGRO is structurally significantly different from PROMPT. Also, AGGRO adopts a much more aggressively optimistic concurrency control strategy that does not bound the length of the chain of uncommitted conflicting transactions from which an optimistically activated transaction may depend. While such a design choice allows for cascading aborts, as it will be also shown in Section 5, its advantages from the aggressive overlap of processing and communication in case of typical STM workloads largely outweight any performance penalty associated with cascading aborts.

Finally, our work is clearly related to the recent literature on distributed STMs [4, 6, 15, 16]. However, except [6], none of these solutions leverages on replication in order to enhance system dependability. In AGGRO, on the other hand, dependability represents a first class design goal, and the STM performance is optimized by seeking maximum overlap between the non-blocking OAB-based replica coordination phase and local transaction processing activities.

Like AGGRO, D^{2}STM [6] is a fault-tolerant replication scheme targeting STM systems and relying on AB for replicas' synchronization. Differently from AGGRO, however, D^{2}STM does not overlap communication and processing phases by exploiting the early, albeit potentially erroneous, indications provided by an OAB service.

3 System Model

We consider a classical asynchronous distributed system model [11] consisting of a set of STM processes $\Pi = \{p_1, \ldots, p_n\}$ 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 processes in Π.

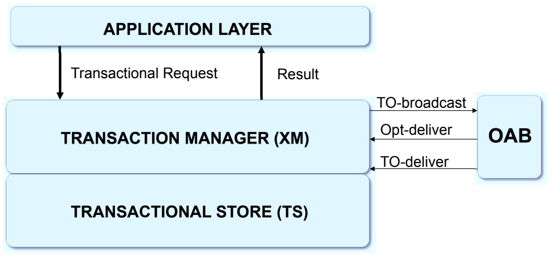


Figure 1. Architecture of a Replicated Software Transactional Memory Process.

For the readers' convenience, we also list the properties characterizing the OAB group communication primitive [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.

The diagram in Figure 1 shows the software architecture of each STM process $p_i \in \Pi$. Applications generate transactions by calling the `invoke` method of the local Transaction Manager (XM), specifying the business logic to be executed (e.g. the name of a method within the transactional memory system) and the corresponding input parameters (if any). The XM is responsible of (i) propagating (through the OAB service) the transactional request across the set of replicated STM 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 the XM about the completion of the execution of a transaction.

We assume that each data item X maintained by TS is associated with a set of versions $\{X^1, \ldots, X^n\}$. A single version of X is committed at any time. On the other hand, an uncommitted version can be in one of the following states: i) Work-in-progress (Wip) - the creator transaction has not reached the complete stage yet; ii) Complete (Comp) - the creator transaction has reached the complete stage, but is not finalized as committed or aborted yet.

Complete data versions are generated by fully executed transactions, and are used to aggressively propagate updates to conflicting transactions. On the other hand, declaration of the existence of Wip versions is used by AGGRO as a means to early express that a given data item is being currently manipulated by some transaction.

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 the transaction data access pattern can vary depending on the current state of the underlying transactional store. More precisely, we assume that the transactional business logic is *snapshot deterministic* [25], 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 a transaction T is re-executed and observes a given snapshot S, defined as the set of values returned by all its read operations, then it behaves deterministically by always executing the same set of read/write operations.

The manipulation of the data items occurs via the following primitives offered by the TS layer: `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; `setCompleteVersion(`X^T, T`)`, which is used for updating the state of a Wip data item X^T created by T (hence belonging to the write-set of transaction T) to Comp; `unsetCompleteVersion(`X^T, T`)`, which is used for removing a complete data item version X^T originally created by T.

4 The AGGRO Protocol

In our architecture, the transaction manager XM exploits the aforementioned data item versioning mechanism to locally drive the execution of transactions. Data item versions in the Comp state are aggressively made visible to other transactions independently of whether the creating transactions will be eventually committed. On the other hand, the XM selects the complete/committed data item versions to be returned by read operations in order to match a serialization order compliant with the order in which transactions are optimistically/finally delivered within the OAB scheme. As pointed out in the Introduction, for environments where the spontaneous network ordering property holds, the optimistic delivery order highly likely reflects the final total order. Hence, transactions reading Comp versions on the basis of the order according to which they have been optimistically delivered are expected not to be eventually (cascading) aborted. In other words, aggressiveness in transaction processing via access to uncommitted (but complete) data items is expected to pay-off (i) by avoiding to stall processing waiting for the finalization of the delivery order and (ii) by not requiring transaction abort and restart.

On the basis of the above considerations, the role of Wip data items becomes central. They represent an early declaration about the fact that a new data item version is likely to reach the Comp state in the (immediate) future. Hence, the XM can exploit the presence of Wip versions to regulate concurrency in a way to temporarily suspend the execution of a transaction T that requires read-access to that data item, and that follows the creating transaction T' in the optimistic/final delivery order. On the other hand, an adverse schedule may lead T to execute the read operation before T' has been able to issue its write on that data item, thus not being able to declare the existence of its Wip version before T issues the read operation. To cope with such a case, we have introduced in the XM an early abort mechanism ensuring that T gets aborted as soon as the Wip version by T' gets produced.

As for the above point, for STM systems hosted by massively (or even conventional) multi-core architectures, we expect minimal likelihood for the optimistically/finally delivered transaction T' not to have reached the complete phase (or to have declared the existence of Wip versions) before the subsequent optimistically/finally delivered transaction T gets activated (thus accessing the post image of data wrt T'). This is because: (A) transactions typically exhibit very fine granularity, (B) as we have also shown in [19], in typical settings, there are normally available computational resources to start processing transactions immediately upon their delivery. On the other hand, in environments with stricter hardware resources (CPU-cores) limitations, the AGGRO concurrency control scheme can be easily integrated with a CPU scheduling scheme (supported at the level of XM-handled threads) based on dynamic priorities, which can favor older transactions within the optimistic/final delivery order. This would create a time-sharing execution that is likely to allow the older transaction T' to declare the existence of Wip versions, or to even run to completion, before T gets actually executed. We omit such a CPU schedule integration mechanism in the presentation of the AGGRO pseudo-code exclusively for simplicity.

The behavior of the XM within the AGGRO protocol relies on a precedence relation between transactions, defined on the basis of the order according to which they are optimistically and/or finally delivered. The relation is expressed as a function of the state of two lists maintained by the XM, named OptDelivered and TODelivered. These lists keep, respectively, transactions that have been either optimistically or finally delivered, and are sorted according to the corresponding delivery order. When a transaction T is optimistically delivered, it gets recorded at the tail of the OptDelivered list. Upon the corresponding final delivery, the transaction is moved from the OptDelivered list (whichever is its current position within this list) to the tail of the TODelivered list. The move operation between the two lists is handled by the XM as an atomic action. Finally, the transaction is removed from the TODelivered list upon commit. In case of no discrepancy between the OAB optimistic and final delivery orders, the transaction moved at the tail of the TODelivered list is always the head-standing one (namely the oldest one) of the OptDelivered list.

By exploiting the above ordered lists, the precedence relation among transactions is expressed as follows. We say that transaction T_i precedes transaction T_j according to the current state of the OAB protocol (as expressed by the lists), using the notation $T_i \overset{OAB}{\rightarrow} T_j$, if one of the three below mutually exclusive conditions holds:

1. T_i and T_j are both currently recorded within OptDelivered, with T_i ordered before T_j;
2. T_i is currently recorded within TODelivered, while T_j is currently recorded within OptDelivered;
3. T_i and T_j are both currently recorded within TODelivered, with T_i ordered before T_j.

We note that the $\overset{OAB}{\rightarrow}$ relation is dynamic, in the sense that, when considering a couple of transactions T_i and T_j, their respective $\overset{OAB}{\rightarrow}$ ordering can change over time. This may occur in case they get sorted within the TODelivered list in the opposite manner, compared to the sorting they had within the OptDelivered list (i.e. in the case of discrepancy between optimistic and final delivery orders for the two transactions). However, once that both these transactions are recorded within the TODelivered list, their respective $\overset{OAB}{\rightarrow}$ order becomes stable (it can no longer be inverted), and depends on which of the two transactions is ordered (and hence TO-delivered) before the other one in the list (see point 3 above). This relative order persists until the preceding transaction gets removed from the TODelivered list upon its commit.

The pseudo-code for the behavior of the XM in shown in Figure 2. For shortness, we do not explicitly show the handler for the receipt of transactional requests by the overlying application, as this simply entails a TO-broadcast operation for propagating the request to the replicated sites via the OAB service. Similarly, we do not explicitly show the logic for the retrieval of the transaction result upon a commit operation, and the delivery of the result to the overlying application. In other words, the pseudo-code presentation is focused on the core mechanisms associated with transaction processing and concurrency regulation, which are activated as soon as a TO-broadcast transactional request gets Opt-delivered to the XM by the OAB layer.

Via the `Opt-deliver` handler, a transaction is inserted within the OptDelivered list, and then gets activated via the ActivateTransaction() function, which we use to encapsulate the transaction processing logic triggering an a-priori unknown sequence of read and write operations. Whenever a write on a data item X is issued, the XM activates the Write() function, which first checks whether X already belongs to the transaction write-set. In the positive case, the working copy within the write-set gets updated, and then the Write() function simply returns. On the other hand, if X does not currently belong to the write-set, it is added to it along with the to-be-written value. Successively, the XM declares via the `MarkAsWip()` primitive the existence of a Wip version associated with the currently writing transaction, say T_i. Then the XM verifies whether there are active transactions that follow T_i according to the $\overset{OAB}{\rightarrow}$ relation, and that read X from a transaction T_k different from T_i such that $T_k \overset{OAB}{\rightarrow} T_i$. These transactions are not correctly serialized according to $\overset{OAB}{\rightarrow}$ since they should have read X from T_i or a subsequent transaction within the $\overset{OAB}{\rightarrow}$ ordering. Hence an abort event for these transactions is issued.

```
List<Transaction> TODelivered,OptDelivered;

upon Opt-deliver(Transaction $T_i$) do
  OptDelivered.add($T_i$); // transaction $T_i$ is added at the tail of the OptDelivered list
  ActivateTransaction($T_i$);

void ActivateTransaction(Transaction $T_i$) { . . . }

void Write(Transaction $T_i$, DataItem X, Value v)
  if ($\nexists X \in$ WriteSet$_{T_i}$)
    WriteSet$_{T_i}$.add(X,v);
    MarkAsWip(X,$T_i$);
    $\forall$ Transaction $T_j$ s.t. $T_i \overset{OAB}{\rightarrow} T_j$ {
      if ($X \in$ ReadSet$_{T_j}$ and $T_k \in$ ReadFrom$_{T_j}$ with $T_k \overset{OAB}{\rightarrow} T_i$) event Abort($T_j$);
  else update $X$ within WritSet$_{T_i}$; // if X already in WriteSet it gets over-ridden

DataItemValue Read(Transaction $T_i$, DataItem X)
  if ($X \in$ WriteSet$_{T_i}$) return WriteSet$_{T_i}$.get(X).value;
  if ($X \in$ ReadSet$_{T_i}$) return ReadSet$_{T_i}$.get(X).value;
  until (MarkedAsWip($X, T_j$)) s.t. $T_j \overset{OAB}{\rightarrow} T_i$) suspend;
  select version of $X$ wrote by $T_j = max\{T_j | T_j \overset{OAB}{\rightarrow} T_i\}$;
  ReadFrom$_{T_i}$.add($T_j$);
  return selected version of $X$

void Complete(Transaction $T_i$)
  $\forall X \in$WriteSet$_{T_i}$ atomically do
    UnmarkAsWip($X, T_i$);
    setCompleteVersion(X,$T_i$);
  set $T_i$ complete;

upon Commit(Transaction $T_i$)
  $\forall X \in$WriteSet$_{T_i}$ atomically do
    UnmarkAsWip($X, T_i$);
    setCommittedVersion($X$,$T_i$);
  TODelivered.remove($T_i$);

upon Abort(Transaction $T_i$) do
  $\forall$ $T_j$ s.t. $T_i \in$ ReadFrom$_{T_j}$ event Abort($T_j$);
  if ($T_i$ is complete) unsetCompleteVersion($X$,$T_i$);
  else UnmarkAsWip($X$,$T_i$);
  release transactional context of $T_i$;
  new thread(ActivateTransaction($T_i$));

upon TO-deliver(Transaction $T_i$ ) do
  atomically do
    OptDelivered.remove($T_i$);
    TODelivered.add($T_i$);
  until $T_i$ not complete or $\exists T_j$ s.t. $T_j \overset{OAB}{\rightarrow} T_i$: suspend;
  if ($\exists X$ s.t. $X \in$ ReadSet$_{T_i}$, $X \in$WriteSet$_{T_j}$, $\neg(T_i \overset{OAB}{\rightarrow} T_j)$) event Abort($T_i$);
  else event Commit($T_i$);
```

Figure 2. Behavior of the XM.

By the above explanation of write operations, the multi-versioning mechanism supported by TS, and exploited by the XM, actually provides a means for avoiding stalls upon write/write conflicts.

In case the requested operation is a read on data item X, the XM activates the Read() function, which first checks whether X is already registered within the transaction write-set/read-set. In the positive case, the registered copy is returned. Otherwise, the XM checks whether the reading transaction, say T_i, follows, according to the $\overset{OAB}{\rightarrow}$ relation, some transaction for which a working copy of X is declared to exist. In the positive case, transaction T_i is temporarily suspended until the above condition is no more verified. Afterwards, the complete or committed version of data item X wrote by the latest transaction preceding T_i according to $\overset{OAB}{\rightarrow}$ is selected and added to the read-set. Then the read-from set of T_i is updated in order to include the read-from dependency associated with the transaction that wrote the selected version of X. Finally, this version is returned.

In the Complete() function, the XM simply removes the declaration about the existence of Wip versions associated with the transaction, and then marks each data item X belonging to the write-set as Comp. Afterwards, the transaction enters the complete state.

In the Commit() function, the XM installs the Comp versions of the data items written by the transaction as committed versions. Then the transaction is removed from the TODelivered list.

In the Abort() function, the XM triggers a (cascading) abort event for all the transactions that read whichever data belonging to the write-set of the currently aborting transaction. These data are then simply discarded, the current transactional context is released, and a new thread for reactivating the transaction is spawned.

Via the `TO-deliver` handler, ther XM moves the transaction from the OptDelivered list to the TODelivered list. Then the execution of this handler is suspended until the finally delivered transaction enters the complete state. The suspend condition also depends on whether there are other transactions that precede the currently TO-delivered one according to $\overset{OAB}{\rightarrow}$. In such a case, the handler waits until the currently TO-delivered transaction i) is fully executed and ii) becomes the minimum element within the $\overset{OAB}{\rightarrow}$ relation. Note that in AGGRO, waiting until a TO-delivered transaction T_i becomes the minimum element of the $\overset{OAB}{\rightarrow}$ relation ensures that every transaction preceding T_i according to $\overset{OAB}{\rightarrow}$ has already been safely committed. At this point T_i can be safely validated by checking whether all the values read by T_i coincide with the ones that are currently in the committed state. If the validation phase is successfully passed, the `TO-deliver` handler generates the commit event for the transaction, which causes the installation of the data items written by the transaction, and the corresponding values, as committed, as well as the removal of the transaction from the TODelivered list. This enables the redefinition of a new minimum element, which iteratively allows generation of the commit event for the corresponding transaction, once it reaches the complete stage.

4.1 Protocol Correctness

Due to space constraints we cannot detail a formal proof of the correctness of AGGRO. Nevertheless, we overview the set of safety and liveness properties ensured by AGGRO providing some informal correctness arguments.

As for safety, AGGRO ensures *opacity* [10] and *1-Copy Serializability* [2]. 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, AGGRO 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 always returns the value generated by the latest complete transaction that precedes T_i according to $\overset{OAB}{\rightarrow}$. Hence, the only possible anomaly that could affect T_i arises whenever T_i observes a value for a data item X generated by a transaction T_j such that $T_j \overset{OAB}{\rightarrow} T_i$, and then a transaction T_k, where $T_j \overset{OAB}{\rightarrow} T_k \overset{OAB}{\rightarrow} T_i$, writes X. In this case, if T_i were to issue a read on any data item generated by T_k, T_i would observe an inconsistent state (having already been serialized before T_k when it issued the read on X), thus violating (O.3). On the other hand, this scenario is avoided by AGGRO since, as soon as T_k writes on X, it would detect that T_i has been scheduled in a way that is inconsistent with $\overset{OAB}{\rightarrow}$, and would immediately abort T_i.

Concerning 1-Copy Serializability, this is ensured by AGGRO 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.

As for liveness, AGGRO ensures *lock-freedom*, which guarantees that there is always at least a thread to make progress, thus ruling out deadlock and livelock scenarios. This is a direct consequence of the fact that the transaction currently representing the minimum element according to $\overset{OAB}{\rightarrow}$ always experiences an abort free (re)run.

5 Simulation Study

Our performance evaluation study is based on a process-oriented simulator developed using the JavaSim simulation package which implements i) the OAB-based replication protocol in [14], referred to as OPT in the following, and ii) the proposed AGGRO protocol. In order to accurately model the execution dynamics of transactions in STM 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, standard benchmark applications for STMs. 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 [5]. The simulation model of the replicated STM system comprises a set of 4 replicated STM processes, each hosted by a machine equipped with eight cores processing transactions at the same rate as in the above architecture.

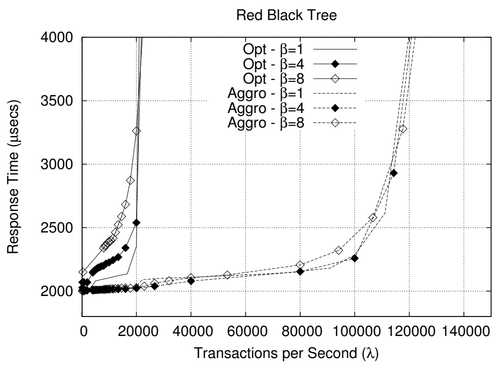
We configured the benchmarks to run in single threaded mode, so to filter out any potential conflict for both hardware resources and data. Also, we extended JVSTM in order to transparently assign a unique identifier to every object within the STM memory and to log every operation (namely, begin/commit/rollback 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 two benchmark applications, RB-Tree and List, that were originally used for evaluating DSTM2 [13] and, later on, adopted in a number of performance evaluation studies of STM systems [5, 6]. These applications perform repeated insertion, removal and search operations of a randomly chosen integer in a set of integers implemented either as a sorted single-linked list or as a red-black tree. We configured the benchmark not to generate any read-only transaction (i.e. searches). 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 STM, as well as the performance gains achievable by AGGRO.

The transactions' arrival process via optimistic and final message deliveries from the OAB layer is modeled in our simulations via a message source that injects messages having as payload a batch of β transactions with an exponentially distributed inter-arrival rate, having mean λ. We recall that batching is a technique very commonly employed to optimize the performance of (Optimistic) Atomic Broadcast protocols [7]. By amortizing the costs associated with the (O)AB execution across a set of messages, batching schemes have been shown to yield considerable enhancement of the maximum throughput achievable by (O)AB protocols. The inclusion of batching schemes in our study of OAB-based replication protocols for transactional systems allows keeping into account optimized configurations for this important building-block group communication primitive. As for the delays of optimistic and final message deliveries, several studies have shown that OAB implementations typically tend to exhibit flat message delivery latency up to saturation [8]. On the other hand, our study is not targeted to explicitly assess the saturation point of the OAB group communication subsystem. For this reason we decided to run the simulations by assuming that the OAB layer does not reach its saturation point. Therefore, independently of the value of the message arrival rate λ, we use in our simulations an average latency of 500 microseconds for the Opt-delivery, and of 2 milliseconds for the TO-delivery. These values have been selected on the basis of experimental measures obtained running the Appia [17] GCS Toolkit on a cluster of 4 quad-core machines (2.40GHz - 8GB RAM) connected via a switched Gigabit Ethernet.

Analysis of the Results: The plots in Figure 3 report the mean transaction response time, i.e. the average time since the TO-broadcast of a transaction till its commitment, for both AGGRO and OPT as a function of the transactions' arrival rate and the batching factor β. For space constraints we focus on the case of absence of mismatches between the optimistic and final delivery. An analysis of scenarios entailing mismatches between the optimistic and final delivery orders can be found in our extended technical report [18].

By the plots, we can draw two main considerations. First, AGGRO allows achieving a striking increase in terms of maximum sustainable throughput by a factor that, independently of the considered settings, fluctuates around the 6x value. The reason underlying this impressive performance gain is associated with AGGRO's ability to make effective use of the locally available computational resources. Specifically, the average CPU utilization with OPT ranges between 5% and 20%, depending on the considered benchmark, even when the system has reached the saturation point. Conversely, as the load increases, AGGRO succeeds in fully utilizing the whole set of cores (that we recall being equal to 8 in this study) locally available at each replica. This depends on the fact that the concurrency control policy adopted by OPT results way too conservative, inducing very long (relatively speaking) periods of stall in the processing activities. It is interesting to highlight that AGGRO's performance gains are achieved despite the rate of aborted transactions grows significantly at high load (getting over 50% close to the saturation point). This is a direct consequence of the aggressively optimistic approach to



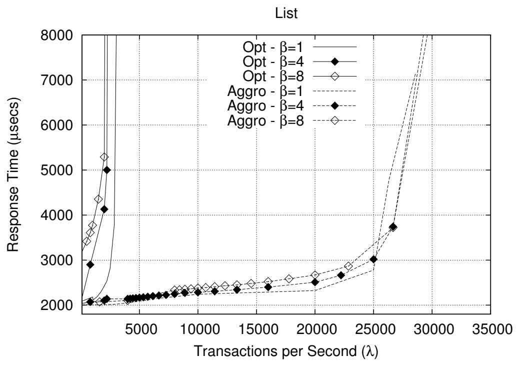


Figure 3. Comparison of the performance of AGGRO and OPT.

concurrency control undertaken by AGGRO, which opts for incurring the risk of (user transparent) cascading aborts in order to achieve maximum overlap between processing and communication.

It is also interesting to note how, at low load, e.g. around 1000 transactions per second, the performance of OPT rapidly degrades as the batching factor β increases. This phenomenon is particularly manifest for the List benchmark, where the mean transaction response time when β=8 is around 75% larger than in absence of batching (β=1). In fact, when a batch of transactions is Opt-delivered, in scenarios characterized by non-negligible conflict probability, they are likely to create convoys. In OPT, only the first transaction of a convoy is immediately processed, whereas the remaining ones stall till the final order notification. Conversely, in AGGRO, the whole batch of delivered transactions is very likely to have been completely processed in the interval since the optimistic to the final order notifications. This makes the transaction response time at low load almost insensitive to the variation of the batching factor (at least for the explored values of β).

6 Summary

In this work we have presented an active replication protocol suited for transactional memory systems. It relies on a classical Optimistic Atomic Broadcast service to determine a global transaction serialization order across all replicas, and on an innovative concurrency control scheme that allows for immediately processing optimistically delivered transactions (according to the guessed serialization order) while the broadcast service is being finalized. The performance gains achievable by our proposal are quantified via a detailed simulation study.

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] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proc. of PPOPP*, pages 247–258, New York, NY, USA, 2008. ACM.

[5] J. Cachopo. *Development of Rich Domain Models with Atomic Actions*. PhD thesis, Technical University of Lisbon, 2007.

[6] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D^{2}STM: Dependable Distributed Software Transactional Memory. In *Proc. of PRDC*. IEEE Computer Society Press, 2009.

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

[8] R. Ekwall and A. Schiper. Modeling and validating the performance of atomic broadcast algorithms in high-latency networks. In *Proc. of Euro-Par*, pages 574–586. Springer, 2007.

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

[10] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proc. of PPOPP*, 2008.

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

[12] J. R. Haritsa, K. Ramamritham, and R. Gupta. The PROMPT real-time commit protocol. *IEEE Trans. on Parallel and Distributed Systems*, 11(2):160–181, 2000.

[13] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. *SIGPLAN Not.*, 41(10):253–262, 2006.

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

[15] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujan, C. Kirkham, and I. Watson. Distm: A software transactional memory framework for clusters. pages 51–58, Sept. 2008.

[16] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proc. of PPOPP*, pages 198–208, New York, NY, USA, 2006. ACM.

[17] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. of ICDCS*, pages 707–710, Phoenix, Arizona, Apr. 2001. IEEE.

[18] R. Palmieri, F. Quaglia, and P. Romano. AGGRO: Boosting STM replication via aggressively optimistic transaction processing. Technical Report 26, INESC-ID, July 2010.

[19] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho. Evaluating database-oriented replication schemes in software transactional memory systems. In *Proc. of IPDPS/DPDNS*, IEEE Computer Society Press, April 2010.

[20] M. Patino-Martinez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Middle-r: Consistent database replication at the middleware level. *ACM Trans. on Computer Systems*, 23(4):375–423, 2005.

[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. *Theoretical Computer Science*, 291(1):79–101, 2003.

[23] P. K. Reddy and M. Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Trans. on Knowledge and Data Engineering*, 16(2):154–169, 2004.

[24] 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.

[25] P. Romano, R. Palmieri, F. Quaglia, N. Carvalho, and L. Rodrigues. Brief announcement: On speculative replication of transactional systems. In *Proc. of SPAA*, Santorini, Greece, June 2010. ACM.