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 highly attractive programming model, thanks to their ability to mask concurrency management issues to the underling applications. In this paper we are interested in dependability of STM systems via replication. In particular we present and AGGRessively Optimistic (AGGRO) active replication protocol relying on Optmitistic-Atomic-Broadcast primitives, which allows increased level of speculation while processing transactions at the replicated STM sites. The protocol relies on an innovative concurrency regulation scheme allowing propagation of dependencies across uncommitted transactions in a controlled manner, namely according to the serialization order guessed on the basis of optimistic message deliveries. Also, it has the distinguishing feature of not requiring a priori knowledge about read/write set of transactions. Instead, conflicts are dynamically detected and handled as soon (and only if) they actually materialize. A simulation study based on benchmark STM applications is also provided, demonstrating the striking performance gains achievable by our proposal (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 Memory ((S)TM) systems leverage on the proven concept of atomic and isolated transaction to spare programmers from the pitfalls of conventional manual lock-based synchronization, thus simplifying the development of concurrent applications. As (S)TM systems started making their way out of research labs and being used in real-life systems (see, e.g., the FenixEDU system [6]), they have been 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, 17, 23] have indeed been targeted to traditional database systems. The fulcrum of these solutions is the synergic integration of an Atomic Broadcast (AB) service [13], ensuring the replicas’ agreement on the global serialization order (GSO) of transactions, and a local, deterministic, concurrency control scheme, guaranteeing that the transaction scheduling at each replica matches the GSO output by the AB service.

However, as highlighted in [26], transaction execution times in (non-replicated) TM systems are typically several orders of magnitude smaller than in conventional database environments. This leads to an amplification of the relative cost of the distributed replica coordination schemes that has not only a significant negative effect on the transaction completion time. Specifically, the (relatively) high communication cost induces (relatively) long periods of stall for the local processing activities, which can cause 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 phases. Such an idea has been already exploited, to some extent, in the context of actively replicated database systems [17, 22] by leveraging on, so called, Optimistic Atomic Broadcast (OAB) services [24]. An OAB service provides early knowledge about message existence (via so called optimistic delivery phase) and early, albeit possibly erroneous, indications of the corresponding final delivery order within the broadcast scheme. As shown in [17], good matching between optimistic and final delivery orders is typical of systems relying on network infrastructures with controlled latency, e.g. LAN-based systems. These infrastructures have been referred to as adhering to the so called spontaneous order property.

In the aforementioned database replication schemes, the OAB’s optimistic indications on the final message delivery order are used to guess the final GSO and to optimistically activate transaction processing activities according to a compliant serialization order. The mechanisms employed in these approaches 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 (or conflict classes) to be accessed by a transaction. Clearly, transactions can only be committed once the final GSO is determined. In some cases 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...
Unfortunately, when employed in the context of (S)TM-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 [22]. However, unlike relational database systems, (S)TM-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 conflict probability 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 it avoids the activation of an optimistically delivered transaction if this is known to conflict with any other already activated transaction. Such a choice has the main advantage of preventing cascading abort, but, by limiting concurrency, may significantly hamper performance. As we have also shown in [21], this is particularly true in (S)TM scenarios where, being the transaction execution time typically much lower than the OAB final deliver latency, the performance benefits achievable by optimistically executing at most one transaction along a conflicting transaction path are significantly slimmer than in conventional database settings.

To overcome the above limitations, in this paper we present AGGRO, an AGGrressively Optimistic replication protocol specifically tailored to (S)TM 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 completed, but not yet finally delivered, transactions across chains of conflicting transactions speculatively executed in a serialization order compliant with the optimistic delivery order. Maximization of the useful work done during the overlap is again related to spontaneous ordering properties offered by the network infrastructure. In 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) transaction conflicts actually materialize.

As we will show by means of a detailed simulation study, AGGRO allows achieving up to 40% reduction of the transaction execution latency and XX% throughput increase with respect to state of the art OAB-based replication schemes when used to handle STM applications deployed on replicas equipped with eight core currently featured CPUs. Such performance gains are obtained without sacrifice of consistency. In fact, beyond ensuring 1-copy serializability, AGGRO also enforces opacity [12] 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 provided 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 primitives to support replication of transactional systems has been widely explored in literature, especially for what concerns database systems (see, e.g., [1, 16, 17, 23]). The key idea underlying these approaches is to exploit AB to determine, in a non-blocking fashion, a global transaction serialization order (across all the replicas), so to circumvent the scalability problems that are known to affect classical eager replication mechanisms based on distributed locking and atomic commit protocols [11]. AGGRO builds on Optimistic Atomic Broadcast (OAB) primitives [24, 17] and, compared to previous works, relies on much more aggressively optimistic local concurrency control mechanisms that seek to achieve maximum overlapping between processing and communication (i.e. coordination) phases. As we will also show in the experimental section, such an approach reveals particularly effective in the context of replicated STM systems, where the ratio between communication and local processing latency is significantly larger with respect to conventional database scenarios [26]. Additionally, unlike traditional OAB-based protocols [17, 22], 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 [25], 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 [25] 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 targets replication of (software) transactional memory systems, and provides strong consistency guarantees despite the crash of (a subset) of the replicas. Further, AGGRO does not blindly explore the whole set of alternative serialization orders in which a given transaction could be executed [26, 27]. Conversely, AGGRO speculatively executes transactions in a serialization order compliant with the delivery order defined by the OAB service. This allows to reduce significantly the computational resources demanded to support speculation, and,
as it will be shown, reveals particularly effective in scenario where the mismatches between the optimistic and the final delivery order are typically unlikely.

AGGRO is similar in spirit also to the PROMPT protocol [14]. The latter protocol targeted 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 (relatively) long stalls incurred in by transactions conflicting with pre-committed transaction. 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 which 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, we will show in Section 5 that this is necessary, when considering a typical STM workload, to achieve effective overlapping between the processing and communication phases.

Speculative, out-of-order processing in event stream applications has been recently proposed in [5]. The idea of leveraging optimism to boost performance is in common with our approach, which however targets replication in transactional systems.

Finally, our work is clearly related to the recent literature on distributed STM solutions [18, 4, 19, 8]. Except from the solution in [8], however, none of these solutions leverages on replication in order to ensure cluster-wide consistency and availability. In AGGRO, on the other hand, dependability represents a first class design goal, and the STM performance is optimized by seeking maximum overlapping between a non-blocking OAB-based replica coordination phase and the local transaction processing activities.

DSTM [8], like AGGRO, is a fault-tolerant replication scheme targeted towards STM systems and relying on an AB service for replicas’ synchronization. Differently from AGGRO, however, DSTM does not seek to 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 [13] consisting of a set of STM processes II = {p1, . . . , pn} 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 messages to all the replicated processes in II.
- **Opt-deliver(m)**, which delivers message m to a process in II in a tentative, also called optimistic, order.

**TO-deliver(m)**, which delivers a message m to a process in II in a so called final order which is the same for all the processes in II.

With no loss of generality, we assume that upon the invocation of **TO-deliver(m)**, message m is exactly the next one finally-ordered by the OAB service, as is the case for most implementations [1]. For the reader’s convenience, we also list the properties characterizing the OAB group communication primitive [24]: **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 p1 and p2, 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 pi ∈ II. 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). 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 for explicit notification to XM about the completion of the execution of a transaction (in terms of transactional statements). We assume that each data item X maintained by TS is associated with a set of versions {X1, . . . , Xn}, where each version X′ stores (i) the data item’s value, (ii) the identity of the creating transaction and (iii) the identity of any active transaction that read X′.

A single version of X is committed at any time. On the other hand, an uncommitted version can be in one of the below states:
• Work-in-Progress (WiP): the creator transaction has not yet reached the complete stage.

• Complete (Comp): the creator transaction has reached the complete stage, but is not yet finalized as committed or aborted.

Complete versions are reflections of transaction computations, and are used to aggressively propagate updates to subsequent conflicting transactions. On the other hand, declaration of the existence of WiP versions is used by AGGRO as a mean to early express that a given data item shall sooner or later have an additional complete version reflecting the updates issued by a transaction along the serialization order built by aggressively propagating dependencies via complete data item versions.

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 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, if whichever transaction \( T \) always sees a snapshot \( S \), defined as the set of values returned by all its read operations, then it behaves deterministically by always executing a same set of read/write operations.

The manipulation of the data items occurs via the following primitives offered by the TS layer: \( \text{MarkAsWiP}(T, X^T) \), which is used for declaring the existence of a WiP version of data item \( X \) created by transaction \( T \); \( \text{UnmarkAsWiP}(T, X^T) \), which is used for undoing the declaration of a previously declared WiP version of data item \( X \) by transaction \( T \); \( \text{MarkedAsWiP}(T, X) \) which is used to query the existence of a WiP declaration on \( X \) by transaction \( T \); \( \text{setCompleteVersion}(X^T, T) \), 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; \( \text{unsetCompleteVersion}(X^T, T) \), 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, XM selects the completed/committed data item versions to be delivered to read operations by other transactions 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 (which are the targets of our protocol, as well as of already existing active replication protocols relying on the OAB service), 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 total order finalization delay on the commitment of data and (ii) by not requiring transaction abort and restart.

On the basis of the above considerations, the role of WiP data item becomes central. They represent an early declaration about the fact that a new data item version shall reach the Comp state in the (immediate) future. Hence, 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. The corner case occurs when an adverse schedule brings \( T \) 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 upon the read by \( T \). To cope with such a case, we have introduce in XM an early abort mechanism such that \( T \) gets aborted as soon as the WiP version by \( T' \) gets produced.

As for the above point, we note that, for a TM system hosted by a massively (or even conventional) multi-core architecture, we expect that the likelihood for the older optimistically/finally delivered transaction \( T'' \) not to be able to run to the complete phase, or to declare the existence of WiP versions, before the subsequent optimistically/finally delivered transaction \( T' \) gets activated (thus accessing the post image of data wrt \( T'' \)), is likely minimal since: (A) transactions typically exhibit very fine granularity, (B) as we have also shown in [21], we typically have available resources so to immediately allow the processing of a transaction upon its delivery.

On the other hand, in environments with stricter hardware resources (CPU-cores) constraints, the AGGRO concurrency control scheme can be easily integrated with a CPU schedule 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 will create a time-sharing execution that is likely to again 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 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 XM, named \( \text{OptDelivered} \) and \( \text{TODelivered} \). 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 \( \text{OptDelivered} \) list. Upon the corresponding final delivery, the transaction is moved from the \( \text{OptDelivered} \) list (whichever is its cur-
rent position within this list) to the tail of the TODelivered list. The move operation between the two lists is handled by XM as an atomic action. Finally, the transaction is removed from the TODelivered list upon commit. In case of no discrepancy between optimistic and final delivery orders within the OAB, the transaction moved at the tail of the TODelivered list is always the head-standing one (namely the oldest one) within 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), denoted as \( T_i \xrightarrow{OAB} T_j \), if one of the three below mutually exclusive conditions holds:

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

We note that the \( \xrightarrow{OAB} \) relation is dynamic, in the sense that, when considering a couple of transactions \( T_i \) and \( T_j \), their respective \( \xrightarrow{OAB} \) 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 (this is exactly the case of discrepancy between optimistic and final delivery orders for the two transactions). However, once both these transactions get recorded within the TODelivered list, their respective \( \xrightarrow{OAB} \) order becomes stable (it cannot be inverted), and depends on which of the two transactions is sorted (and hence TO-delivered) before the other one in the list (see point 3 above). This respective order persists until the preceding transaction gets removed from the TODelivered list upon its commit.

The pseudo-code for the behavior of 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, since it only involves 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 such a 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-broadcasted transactional request gets Opt-delivered to XM by the OAB layer.

Upon the OptDelivered of a transaction, it is inserted within the OptDelivered list, and then gets activated via the ActivateTransaction() function. The exact behavior of the latter function is not shown since we consider is performs typical tasks associated with, e.g., the definition of a fresh transactional context, and the activation of transactional statements. During the execution of any statement, both read/write operations can be issued which are handled by XM as follows. For a write operation on a data item \( X \), 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, in case \( X \) does not currently belong to the write-set, it gets added, together 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 \( T_x \). Then XM verifies whether there are active transactions that follow \( T_i \) according to the \( \xrightarrow{OAB} \) relation, and that read \( X \) from a transaction \( T_k \) different from \( T_i \) such that \( T_k \xrightarrow{OAB} T_i \). These transactions are not correctly serialized according to \( \xrightarrow{OAB} \) since they should have read \( X \) from \( T_i \) or a subsequent transaction within the \( \xrightarrow{OAB} \) ordering. Hence an abort event for these transactions is issued.

By the above explanation of write operations, the multi-versioning mechanism supported by the TS, and exploited by XM, actually provides a mean for avoiding at all write/write conflicts. This avoids any suspend phase upon writing a data, and frames regulation of concurrency within view-serializable schedules.

In case the requested operation is a read on data item \( X \), 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 gets returned. Otherwise, XM checks whether the reading transaction, say \( T_i \), follows, according to the \( \xrightarrow{OAB} \) relation, some transaction for which a working copy of \( X \) is declared to exist. In the positive case, transaction \( T_i \) gets temporarily suspended until the above condition is no more verified. Afterwards, the complete or committed version of data item \( X \) wrote by the the latest transaction preceding \( T_i \) according to \( \xrightarrow{OAB} \) 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.

With the Complete() function, 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 is set as complete.

With the Commit() function, XM installs the Comp versions of the data items wrote by the transaction as committed versions. Then the transaction is removed from the TODelivered lists.

With the Abort() function, XM issues an abort event for all the transactions that read whichever data belonging to the write-set of the currently aborting transaction (i.e. they read from the currently aborted transaction). These data, and the corresponding wrote values, are then unset as complete, or unmarked as working, depending on whether the currently aborting transaction was already complete or not. The current transactional context is released, and a new thread for reactivating the transaction is spawned.
Upon the TO-deliver of a transaction, XM moves the transaction from OptDelivered to TODelivered. Then the TO-deliver handle suspends its execution 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 $OAB \rightarrow$. In such a case, handler keeps suspending its execution until the currently TO-delivered transaction becomes the minimum element within the $OAB \rightarrow$ relation. After that, the XM handler generates the commit event for the transaction. The reason for the above suspend state of the TO-deliver handler instance associated with transaction $T_i$ to depend on both both the completion of the $T_i$ and the existence of other transactions that precede the currently TO-delivered one according to $OAB \rightarrow$ is related to that:

(A) The transaction cannot be committed before it reaches the complete stage.

(B) A complete TO-delivered transaction $T_i$, registered within the TODelivered list, needs to also become the minimum element of the $OAB \rightarrow$ relation since TO-delivery of a complete transaction, by itself, does not suffice to enable a serializable schedule. This is because there might be other TO-delivered transactions, preceding $T_i$, according to $OAB \rightarrow$ which might still be in the execution phase, and possibly reveal a conflict with $T_i$ (e.g., by writing a data item $X$ already ready by $T_i$). This might occur just because the AGGRO protocol does not rely on a priori knowledge of the read/write set of transactions, thus dealing with conflicts as soon (an if) they actually materialize.

Once a complete transaction becomes the minimum element of the $OAB \rightarrow$ relation, the commit event is generated, and, as said, the Commit() handler, beyond installing the data items written by the transaction, and the corresponding values as committed, removes the transaction from the TODelivered lists. 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 for the proof of the correctness of AGGRO. Nevertheless, in this section we overview the set of safety and liveness properties ensured by AGGRO providing some informal correctness arguments.

As for safety, AGGRO ensures opacity [12] 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. AGGRO ensures (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 stage (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 completed transaction that precedes $T_i$ according to $OAB \rightarrow$. Hence, the only possible anomaly that could affect

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{Figure2.png}
\caption{Behavior of XM.}
\end{figure}
Concerning 1-Copy Serializability, it 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 deadlocks and livelocks scenarios. This is a direct consequence of the fact that the transaction currently representing the actual minimum element according to $OAB \rightarrow$ always experiences an abort free (re)run.

5 Simulation Study

Our performance evaluation study is based on a processor-oriented simulator developed using the JavaSim simulation package which implements i) a baseline AB-based replication protocol, not exploiting any indication concerning the optimistic delivery order, ii) the OAB-based protocol in [17], referred to as "Opt" in the following, and iii) the proposed AGGRO protocol.

5.1 Simulation Model

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 (S)TMs. 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 OSX 10.6.2, and the used STM layer is JVSTM [7]. The simulation model of the replicated STM system comprises a set of 4 replicated STMPs, each of which hosted by a multi-core machine with 8 cores exhibiting the same power 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 layout 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).

Since any tracing strategy unavoidably introduces overheads, especially in STM applications where transaction execution times are often less than 1 msec, in order to ensure the accuracy of the information concerning the duration of transactions, we repeated each benchmark run (ensuring the deterministic re-execution of an identical set of transactions) by also disabling the logging functionality. Then we compared the resulting mean transaction execution time with the one obtained when logging is enabled. This allowed us to compute a per-benchmark scaling factor (that, on the average, was found to be around 15x) used to adjust the duration of the transaction execution, thus filtering out the overheads associated with the logging layer, before feeding this information to the simulator.

The traces were collected running three benchmark applications, RB-Tree, SkipList and List, that were originally used for evaluating DSTM2 [15] and, later on, adopted in a number of performance evaluation studies of (STM) systems [7, 8]. 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, a skip list, or a red-black tree. We configured the benchmark to initialize the set of integers with 128 values, and allowed it to store up to a maximum of 256 values. Finally, we configured the benchmark not to generate any read-only transaction (i.e. searches). This choice depends on the fact that, in both the protocols considered in this study, read-only transactions can be executed locally, without the need for propagation via the atomic broadcast (as read-only transactions do not alter the state of the replicated transactional memory system). By only considering update transactions in our study, 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 the AGGRO protocol. The below table reports the average transaction execution time observed for the three benchmarks via the aforementioned tracing scheme:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Avg. Transaction Exec. Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>RB-Tree</td>
<td>77 μsec</td>
</tr>
<tr>
<td>SkipList</td>
<td>281 μsec</td>
</tr>
<tr>
<td>List</td>
<td>324 μsec</td>
</tr>
</tbody>
</table>

The transactions’ arrival process via Opt/TO message deliveries from the OAB layer is modeled in our simulations via a message source that injects messages having as payload a batch of $\beta$ transactions with an exponentially distributed inter-arrival rate, having mean $\lambda$. We recall that batching is a technique very commonly employed to optimize the performance of (Optimistic) Atomic Broadcast protocols [9]. 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. In order to derive results representative of a wide range of settings, we let the batching size $\beta$ vary in between 1 and
As for the delay due to the OAB layer (i.e. the delay of the Opt-delivery and of the corresponding TO-delivery), several studies have shown that OAB implementations typically tend to exhibit flat message delivery latency up to saturation [10]. 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 $\lambda$, we use in our simulations an average latency of 350 microseconds for the Opt-delivery, and of 2 milliseconds for the TO-delivery. These values have been selected based on experimental measures obtained running the Appia [20] GCS Toolkit on a cluster of 4 quad core machines (2.40GHz - 8GB RAM) connected via a switched Gigabit Ethernet.

5.2 Analysis of the Results

The first three plots (from the top) in Figure 3 report the mean transaction response time, i.e. since it is atomic broadcast till it is committed, of AGGRO and OPT in absence of mismatches between the optimistic and final delivery, as a function of the transactions' arrival rate and the batching factor $\beta$. By evaluating the performance of the two protocols in absence of any mismatch between the optimistic and final delivery orders, we are de facto assessing their efficiency in ideal working conditions where any optimistically activated transaction will always be processed in a correct serialization order.

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 achieved AGGRO is associated to its ability to make an effective use of the locally available computational resources. Specifically, the average CPU utilization with Opt ranges between the 5% and the 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, in the context of STM workloads, way too conservative, inducing very long (relatively speaking) period of stalls in the processing. It is interesting to highlight that the strong performance gains of AGGRO are achieved despite the rate of aborted transactions grows significantly at high load (getting larger than 50% close to the saturation point). This is an unavoidable consequence of the aggressively optimistic approach to concurrency control undertaken by AGGRO, which opts for incurring in the risk of (user transparent) cascading aborts in order to achieve maximum overlapping between processing and communication. An interesting conclusion that can be drawn from these data is, indeed, that, in typical STM settings (very short lasting transactions and medium/high degree of parallelism of the underlying hardware platform), cascading abort is a lesser evil compared to the threat of hampering parallelism with very conservative approaches to concurrency control.

It is also interesting to note how, at low load, e.g. around 1000 transactions per second, the performance of Opt can rapidly deteriorate as the batching factor $\beta$ increases. This phenomenon particularly manifest in the case of the List benchmark, where the mean transaction response time when $\beta=8$ is around 75% larger than in absence of batching ($\beta=1$). In fact, in scenarios of not negligible contention, as a batch of transactions is Opt-delivered, they tend to form convoys. With only the first transaction of a convoy is immediately processed, whereas the remaining one stall till the delivery of the final order notification. Conversely, in AGGRO, the whole chain of convoyed transaction is very likely to have been completely processed in the interval since the optimistic and the final delivery 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 $\beta$).

We conclude this section by assessing the extent to which the performance of the two considered protocols are affected in scenarios of non-negligible probability of mismatches between the optimistic and final delivery orders (or simply message reorderings in the following). The last graph in Figure 3 reports the protocols' performance in absence of batching, when considering the SkipList benchmark and a message reordering probability equal to 20%. The choice of this value is supported by the empirical study carried out in [17], which showed that, when using standard IP-multicast in small scale (around 10 nodes) clusters, the probability of out-of-order deliveries is lower than 20% when the broadcast rate is lower than 100K AB/sec. Considering that the rate of AB/sec is at most 30K, this means that we are actually making a rather pessimistic assumption concerning the likelihood of message reordering in our simulation model. By the plots, we observe that, as expected the performance degradation of Opt, with respect to an ideal scenario without message reordering, is more reduced with respect to the one experienced by AGGRO. However, the saturation point of AGGRO is practically unaltered, still scoring a throughput increase with respect to Opt in the house of 6x.

References


Figure 3. Contrasting the performance of AG-GRO and OPT.