On the Relevance of Total-Order Broadcast Implementations in Replicated Software Transactional Memories

Tiago M. Vale, Ricardo J. Dias, and João M. Lourenço

Departamento de Informática and CITI
Universidade Nova de Lisboa, Portugal
{t.vale,ricardo.dias}@campus.fct.unl.pt,
joao.lourenco@fct.unl.pt

Abstract. Transactional Memory (TM), an attractive solution to support concurrent accesses to main-memory storage, is already being deployed by some of the major CPU and compiler manufacturers. To address scalability and dependability challenges, researchers are now combining replication, TM and certification-based protocols. To maintain consistency and ensure common transaction serialisation order, these protocols rely in a total-order broadcast primitive, usually provided by some Group Communication System (GCS). The total-order broadcast service can be implemented by different algorithms, which hold different properties. In this paper we present a detailed analysis of the impact of some algorithms implementing total-order broadcast in different TM workloads, opening up future work to improve performance of replicated TMs.

1 Introduction

The interest in research on paradigms for parallel programming increased as multi-core computers hit mainstream. Software Transactional Memory (STM) [1] as earned interest from both the academic and industrial research communities. STM relieves the programmer from the subtleties of the traditional lock-based concurrency control by adapting the familiar concept of transaction inherited from the database world. Enterprise-class STM-based applications have already been deployed in production systems[1]. These real-world applications usually hold requirements such as scalability and reliability which are commonly tackled using replication. Distributed STM has been similarly motivated as an alternative to (distributed) lock-based concurrency control in distributed systems, where the problems associated with locks are exacerbated.

Given the similarities between STM and database transactions, research on STM replication have borrowed inspiration from the literature in replicated databases. A handful of replication protocols [2,3,4,5], commonly referred to as certification-based, have been proposed and evaluated in the context of replicated

[1] https://fenix-ashes.ist.utl.pt

J.M. Lourenço and E. Farchi (Eds.): MUSEPAT 2013, LNCS 8063, pp. 49–60, 2013.

STM systems. Replica consistency is ensured at commit time using a total-order broadcast to guarantee a common transaction serialisation order. Unfortunately, the relative overhead introduced by the certification of distributed memory transactions is much higher than the overhead introduced by the certification of distributed database transactions. On the one hand, memory transactions operate over main memory which is a very fast storage media, and on the other hand, some key features of databases require additional processing time, such as SQL parsing, plan optimisation, and secondary storage accesses.

Total-order broadcast can be implemented using different algorithms, which exhibit different properties such as latency, fairness and throughput. In this paper, and to the best of our knowledge, we present the first study of the impact that the different algorithms implementing total-order broadcast have on replicated STMs. In the remainder of this paper, we discuss certification-based protocols and their key ingredient, the total-order broadcast, in §2; followed by a discussion of its impact in replicated STM environments in §3. We proceed with a description of our implementation and discuss the experimental results in §4 and §5, respectively. We close the paper with a discussion of the related work in §6, and some concluding remarks in §7.

2 Software Transactional Memory Replication

While the transaction concept bridges the world of databases and STM, memory transactions' execution time is significantly smaller than database transactions. Memory transactions only access data in main memory, thus not incurring in the expensive secondary storage accesses characterising the latter. Furthermore, SQL parsing and plan optimisation are also absent in STM. On the other hand this increases the relative cost of remote coordination. Nonetheless, literature on replicated and distributed databases represents a natural source of inspiration when developing protocols for replicated STM. In fact, current research on replicated STM have embraced protocols commonly referred to as certification-based. These protocols rely on total-order broadcast, usually provided by some group communication system, to impose a global transaction serialisation order.

2.1 Total-Order Broadcast

Informally, a total-order broadcast ensures that messages sent to a set of recipients are delivered in the same order by all recipients. Total-order broadcast guarantees the following properties [6]: (1) *Validity*: if a correct replica TO-broadcasts a message m, then it eventually TO-delivers m; (2) *(Non-)uniform Agreement*: if a (correct) replica TO-delivers a message m, then all correct replicas eventually TO-deliver m; (3) *Uniform Integrity*: for any message m, every replica TO-delivers m at most once, and only if m was previously TO-broadcast by $\mathsf{sender}(m)$; and (4) *(Non-)uniform Total Order*: if (correct) replicas r_1 and r_2 both TO-deliver messages m and m', then r_1 TO-delivers m before m', if and only if r_2 TO-delivers m before m'. A broadcast that satisfies all these properties

except (4), *i.e.*, that provides no ordering guarantee, is instead called a reliable broadcast.

Notation. A total-order broadcast consists of two primitives, to-broadcast(m) and to-deliver(m). We say a replica r TO-broadcasts a message m when r executes to-broadcast(m), and r TO-delivers m when r executes to-deliver(m). We denote R-broadcast and R-deliver analogously for reliable broadcast. The sender of a broadcasted message m is denoted by sender(m).

There are several algorithms to implement total-order broadcast [6]. In sequencer-based algorithms one replica is elected as the sequencer and is responsible for ordering messages. For example, any replica r wanting to TO-broadcast a message m, starts by unicasting m to the sequencer, which in turn broadcasts m on behalf of r. A different approach is followed by privilege-based algorithms. These algorithms rely on the idea that senders can broadcast messages only when they are granted the privilege to do so. The privilege to broadcast (and order) messages is granted to only one process at a time, but this privilege circulates from process to process.

2.2 Certification-Based Protocols

Current research in STM replication has borrowed protocols from the database literature usually classified as certification-based. In certification-based protocols, unlike classical eager replication, transactions are optimistically executed on a single replica without any remote coordination. Updates are buffered and applied at all replicas if a transaction is found to be valid and successfully commits. Replicas coordinate at commit time by way of a distributed protocol that validates (certifies) transactions and establishes a global transaction serialisation order. Thus, the outcome of a transaction's validation is the same at every replica, and the updates of valid transactions are applied at every replica in the same order. While update transactions require replica coordination at commit time, as described, read-only transactions can validate and commit locally at the host replica.

To impose a global transaction serialisation order, certification-based protocols rely on total-order broadcast to disseminate transactions at commit time. This contrasts with classical eager replication protocols based on distributed locking that potentially incur in deadlocks and suffer from communication overheads during the transaction execution phase. Depending on which transactional data is TO-broadcasted, certification protocols can be classified in two schemes: Non-Voting [7] and Voting [8]. In the Non-Voting scheme, when a transaction t executing at some replica enters the commit phase, both its write set W and read set R are TO-broadcasted. This means that each replica is able to independently validate and abort or commit t, as they are in possession of all the necessary information, *i.e.*, both W and R. Note that given the total ordering of the deliveries, all replicas will process all transactions in the same order, so the result of any transaction's validation will be the same on all replicas.

By disseminating both W and R the Non-Voting scheme requires a single communication round to commit a transaction. However, the read set is typically much larger than the write set, thus the second scheme, Voting, explores the trade-off of exchanging potentially much smaller messages (since R, typically much larger than W, is not disseminated) at the expense of requiring two communication rounds (an additional R-broadcast) instead of just one.

3 On the Relevance of Total-Order Broadcast Implementations

Consider the typical work flow of a transaction-processing thread *Th* under a certification-based protocol. *Th* executes transaction t. If t is read-only it is locally validated and committed (or aborted, thus re-executed) by *Th*. Otherwise *Th* TO-broadcasts t and waits. Upon the respective TO-delivery, *Th* validates and commits or aborts t, re-executing if aborted. We refer to the time *Th* waits, *i.e.*, the time between the TO-broadcast and the respective TO-delivery, as latency.

Intuitively, different implementations of total-order broadcast will have a different impact on latency, and thus on the performance of typical replicated STM deployments, where each replica executes roughly the same percentage of update transactions. In the sequencer-based algorithm (see §2.1) it is expected that (1) latency in replica s, assigned as sequencer, is lower than in the remaining replicas; and (2) transaction ordering is biased towards transactions from s, because unlike other replicas, s can skip an initial unicast. With privilege-based implementations, latency is expected to be similar in all replicas, as they are only allowed to broadcast during their slot. Thus, a sequencer-based total-order broadcast implementation is likely to allow a replicated STM to achieve higher performance than a privilege-based implementation at the expense of an unequal contribution of each replica to the global throughput of the system. The sequencer will likely execute much more transactions, and faster, than the other replicas. This contrasts with the privilege-based solution in which each replica is expected to contribute evenly.

4 Implementation

To implement and evaluate the relevance of different total-order broadcast implementations in replicated STMs, we have extended an existing Java STM framework [9] to support replication in a transparent way for the programmer. In this paper we will only cover the architecture and API for implementing replicated STM algorithms, whose understanding is necessary for our study. The STM layer implements an STM algorithm and exposes an API to the application with the following primitives:

```
1.  stm_commit(t):
2.    if t is not read-only and stm_validate(t)
3.        certify(t)
4.    else
5.        if not stm_validate(t)
6.            stm_abort(t)
7.  certify(t):
8.    to-broadcast(t)
9.    wait until to-deliver(t)
10.   if stm_validate(t)
11.       stm_apply(t)
12.   else
13.       stm_abort(t)
```

Fig. 1. Non-Voting Certification implementation on our framework

stm_begin(t)	Begin transaction t.
stm_commit(t)	Request commit of transaction t.
stm_abort(t)	Aborts transaction t.
stm_read(t, m)	Transaction t reads value of memory location m.
stm_write(t, m, v)	Transaction t writes value v to memory location m.

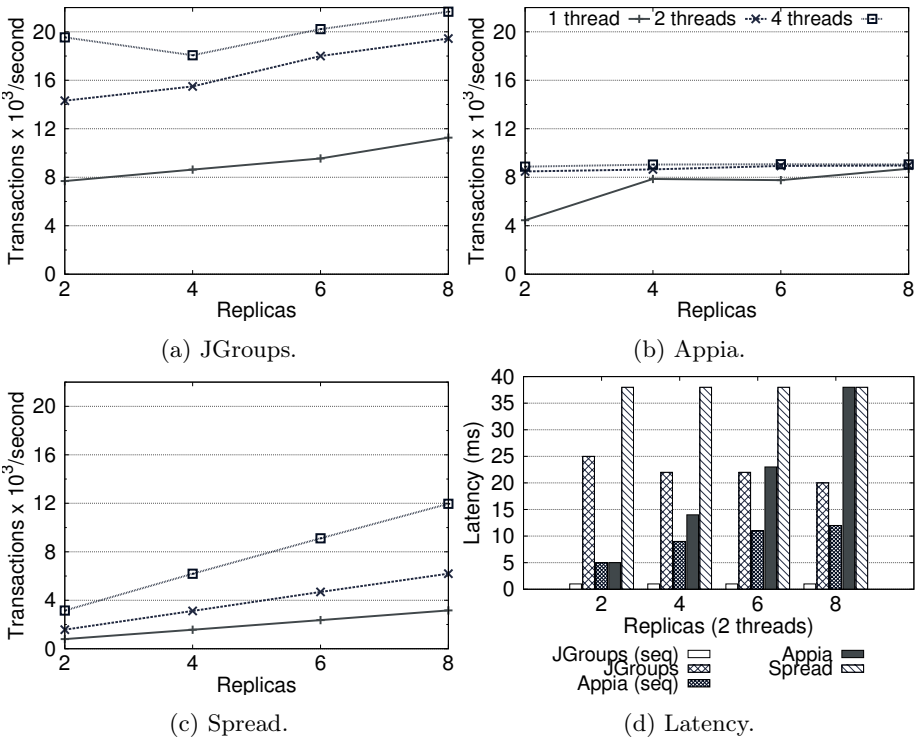
Application code defines transactions by tagging methods with an `@Atomic` annotation. These methods are bytecode-instrumented to inject calls to the underlying STM layer using the described API where appropriate. To support the distributed certification of transactions, the STM layer interacts with the certification-based protocol layer through the following API:

stm_validate(t)	Transaction t is validated against the local STM state.
stm_apply(t)	Transaction t's updates are applied to the local STM state.
certify(t)	Issues the distributed certification of transaction t.

The certify primitive allows the STM layer to trigger the distributed certification of an update transaction when stm_commit is issued by the application. When certifying a transaction t, stm_validate and stm_apply concede the certification-based protocol the ability to validate t in the local replica and apply its updates, respectively. On the bottom of the architectural stack we have a group communication system that provides the total-order broadcast primitives, to-broadcast(t) and to-deliver(t). These are used by the certification-based protocol layer to broadcast transactions for certification while ensuring a common order across all replicas. Figure 1 illustrates an implementation of the Non-Voting Certification under our framework.

5 Experimental Results

All the experiments were performed in a cluster of 8 nodes interconnected via gigabit ethernet, each one equipped with a Quad-Core AMD Opteron 2376 at 2.3 Ghz, 4×512KB cache L2, and 8 GB of RAM. The installed Java Platform was OpenJDK 6. The local STM layer of our infrastructure used the TL2 algorithm [10]. The certification-based protocol was the Non-Voting scheme. With regard to the underlying group communication system providing the total-order broadcast primitives needed by the certification-based protocol, three different implementations were considered: JGroups, Appia [11] and Spread. Switching between each group communication system (GCS) is done by parametrisation when executing the target program, hence no code rewriting whatsoever is needed.



(a) JGroups. (b) Appia.

(c) Spread. (d) Latency.

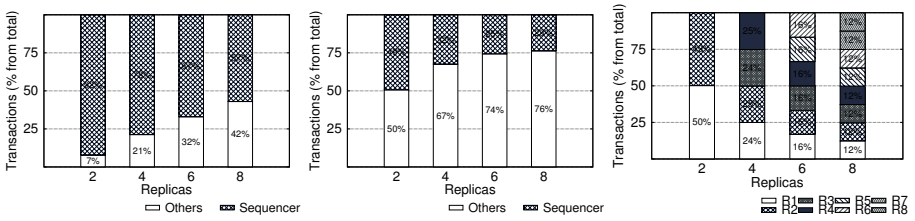
Fig. 2. Throughput and total-order broadcast latency in Red-Black Tree, configured with initial size 32078, range 131072 and 10% updates

JGroups is a well-known toolkit used in several projects and was configured according to the SEQUENCER protocol configuration from the freely available repository[2], providing non-uniform total order using a sequencer algorithm. Appia is a group communication system that has been used in contributions to STM replication, and was configured with `SequencerUniformLayer` to provide uniform total-order broadcast using a sequencer-based algorithm. Spread uses a privilege-based algorithm and we used the vanilla configuration and message type was set to AGREED, which guarantees non-uniform total order.

5.1 Red-Black Tree

We start by considering a common micro-benchmark from the literature, the Red-Black Tree. This benchmark is composed of three types of transactional operations: (1) insertions, which add an element to the tree (if not already present); (2) deletions, which remove an element from the tree (if present); and (3) searches, which search the tree for a specified element. Insertions and deletions are said to be update transactions. The tree was populated with 32768

[2] `https://github.com/belaban/JGroups`.



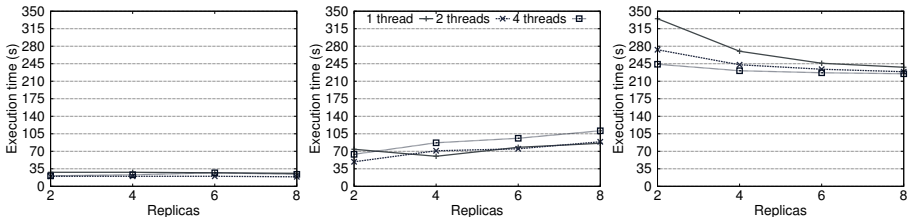
(a) JGroups. (b) Appia. (c) Spread.

Fig. 3. Throughput breakdown in Red-Black Tree, with 2 threads and configured with initial size 32078, range 131072 and 10% updates

pseudo-randomly generated values, ranging from 0 to 131072. Each thread executed 10% of update transactions. Hence, the workload is characterised by very small and fast transactions, with very low contention.

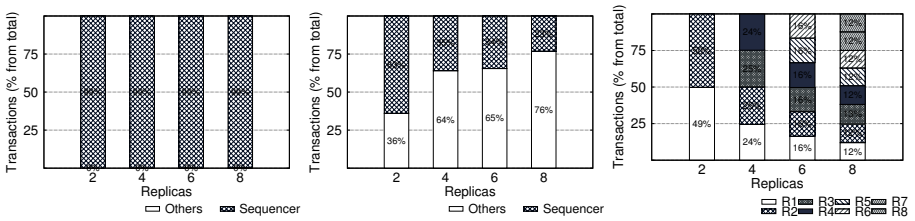
Fig. 2 shows the throughput of our system (in transactions$\times 10^3$ per second, higher is better) on the micro-benchmark, varying the number of replicas and the number of threads per replica. Unsurprisingly, JGroups (Fig. 2a) achieves the best performance of the three for every combination, due to both the fixed sequencer implementation and the uniformity relaxation. Appia (Fig. 2b) quickly peaks at around half the throughput of JGroups, which seems to be hitting a bottleneck, perhaps due to the requirements of the uniform property. Finally, in Fig. 2c, we have the throughput of the system using Spread. The linear scalability displayed is consistent with the idea that the algorithm implemented by Spread achieves fairness. Since every node is provided with equal opportunities to broadcast and order messages, the system scales either with more threads per node, or when nodes increase, not incurring in the bottleneck of a fixed sequencer. In Fig. 2d we have the average total-order broadcast latency for each group communication system. As discussed in §3, the sequencer replica in both JGroups and Appia – JGroups (seq) and Appia (seq), respectively – suffers from lower latencies than the rest of the replicas. With Spread and its privilege protocol the latency is the same for all replicas.

Intuitively, one expects that the higher performance of JGroups is achieved at the cost of unfairness. The sequencer incurs in substantially lower latency, so that replica alone should be dominating the system's throughput. Spread should exhibit the exact opposite behaviour, *i.e.*, each replica contributes evenly to the total throughput. Since the privilege protocol gives exclusive broadcast rights to each replica at a time, hence the same latency for all replicas. In Fig. 3 we breakdown each replica's contribution to the overall system throughput, where the x-axis represents the number of replicas and the y-axis the percentage of transactions each replica executed. Each colour represents the portion of transactions executed by a specific replica. The results corroborate our intuitions. In JGroups (Fig. 3a) the sequencer totally dominates the system's throughput, while in Spread (Fig. 3c) each replica contributes evenly. Appia (Fig. 3b) lies in the middle due to the additional overhead imposed by the uniform property, which is consistent with the measured latencies.



(a) JGroups. (b) Appia. (c) Spread.

Fig. 4. Execution time in Intruder, with the *intruder* configuration from [12]



(a) JGroups. (b) Appia. (c) Spread.

Fig. 5. Transaction breakdown in Intruder, with 2 threads and the *intruder* configuration from [12]

5.2 Intruder

In the Intruder benchmark each thread repeatedly executes 3 phases. The first phase basically involves a simple FIFO queue from which threads pop a packet. In the second phase threads add the packet to a dictionary (implemented by a self-balancing tree) that contains lists of packets that belong to the same flow. If all the packets of a flow have been delivered, they are reassembled and added to the completed packets FIFO queue. The final phase consists of taking a reassembled packet from the FIFO queue and checking if it has been compromised.

The benchmark was parameterized according to the *intruder* configuration in [12]. There were 2048 flows with 4 packets each, and 10 of the flows had been attacked. Transactions under this configuration are small and fast, and the workload is highly contented, due to both of the FIFO queues and the rebalancing of the tree in the reassembly phase. Thus, this workload distinguishes itself from Red-Black Tree's in the contention level.

Fig. 4 shows the execution time (y-axis) when varying the number of replicas (x-axis). The system behaves differently depending on the GCS used. When using JGroups (Fig. 4a) the performance is independent of the number of replicas. The sequencer does all the work and the other replicas have their transactions constantly aborted until the benchmark finishes, as can be seen in the transaction breakdown for JGroups in Fig. 5a. With Appia, Fig. 4b, the system degrades performance as more replicas are added. Since the sequencer does not totally dominate in Appia (Fig. 5b) as in JGroups, this is the expected behaviour due

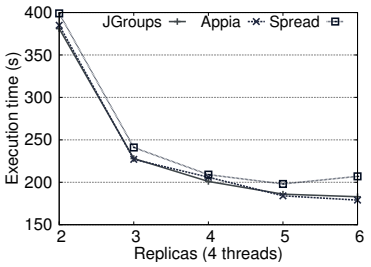


Fig. 6. Execution time in Genome, with the *genome* configuration from [12]

to the contended workload leading to a high abort ratio. With Spread the execution times are much higher (Fig. 4c) due to the privilege-based implementation. Nevertheless, the system performs better when more threads are added.

5.3 Genome

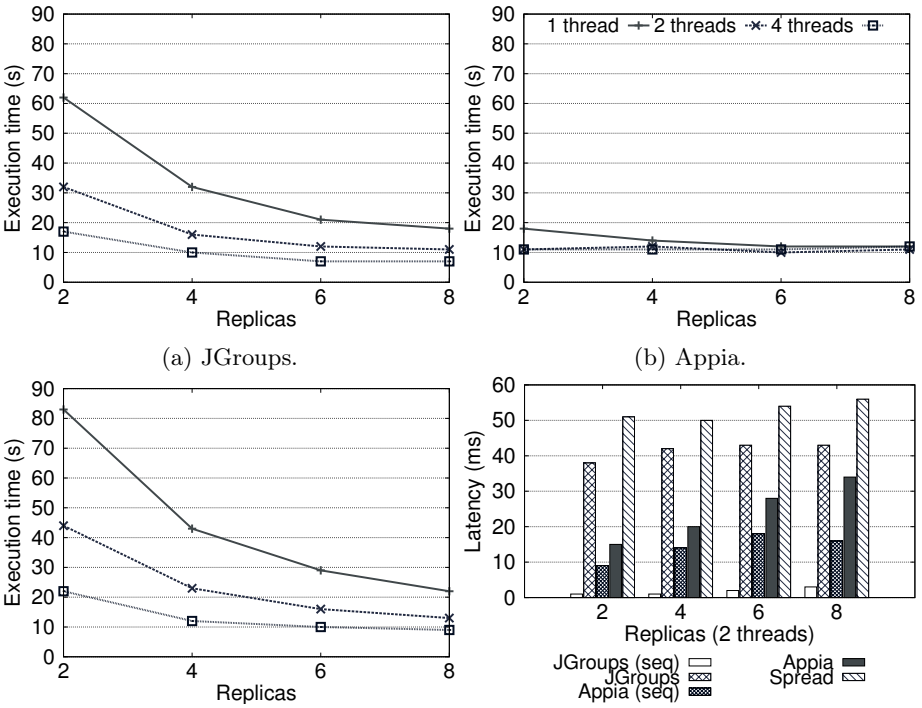
The Genome benchmark consists of several steps which are executed sequentially, but inside each step several threads execute concurrently. But since the steps are sequential, threads wait for each other when advancing from one step to the next. The last step is completely sequential (it is executed by a single thread), and there is one step which is a mix of concurrent and sequential parts.

The benchmark was parameterized according to *genome* configuration in [12]. This workload is radically different from both Red-Black Tree's and Intruder's. Overall, transactions are of moderate length (with regard to the number of operations) and there is little contention. Unlike the previous benchmarks, in Genome data is partitioned among threads. Threads execute a sequence of steps in synchrony, *i.e.*, threads must wait for each other when advancing from step a to step b. With this workload, it is expected that the different total-order broadcast implementations do not have a meaningful impact as replicas can only progress in group. In fact, the system exhibits similar execution times independently of the GCS employed, as seen in Fig. 6.

5.4 Vacation

The Vacation benchmark is implemented as a set of trees that keep track of customers and their reservations for various travel items. During the execution of the workload, several client threads perform a number of sessions that interact with the travel system's database. In particular, there are three distinct types of sessions: reservations, cancellations, and updates. Each of these client sessions is enclosed in a coarse-grain transaction to ensure validity of the database. Consequently, transactions are of moderate size.

The benchmark was parameterized according to the *vacation-low* configuration in [12], where contention is low. The database had 16384 records of each reservation item, and clients performed 4096 sessions. Of these sessions, 98% reserved or cancelled items and the remainder created or destroyed items. Sessions



(a) JGroups. (b) Appia.

(c) Spread. (d) Latency.

Fig. 7. Execution time and total-order broadcast latency in Vacation, with the *vacation-low* configuration from [12]

operated on up to 2 items and were performed on 90% of the total records. This workload is similar to Genome's considering that each thread has its own work to perform. Thus, the complete bias of JGroups towards the sequencer should not yield great performance comparing to Spread, since the sequencer can not "steal" the work from the remaining replicas. But unlike Genome, the whole thread execution path is concurrent.

Fig. 7 shows the execution time and latency when executing Vacation with the differents GCS. The most interesting aspect with this experiment is that Appia (Fig. 7b) performs better than both JGroups (Fig. 7a) and Spread (Fig. 7c) right from the start with 2 replicas. Analysing the latencies (Fig. 7d) we can observe that the average latency of Appia replicas is lower than Spread replicas, as expected. With JGroups the sequencer has the usual low latency, but the other replicas exhibit higher latency than any Appia replica. Thus, with Appia replicas can progress concurrently while with JGroups the sequencer finishes first and only afterwards the other replicas progress. Spread exhibits its usual behaviour.

6 Related Work

In this paper we have studied the impact of different total-order broadcast implementations on the regular Non-Voting Certification scheme. The following works are also related to the use of certification-based protocols in STM replication and share the common goal of reducing the coordination overhead, but none studies the impact of the GCS in the system's workload and throughput. Couceiro et al. in [2] propose the use of bloom filters to reduce the size of the messages TO-broadcasted, as the efficiency of the total-order broadcast is known to be strongly affected by the size of the exchanged messages [13]. The authors encode the read set in a bloom filter whose size is computed to ensure that aborts due to the bloom filter's false positives are less than a user-unable threshold. The work in [5] supports the coexistence of the Voting and Non-Voting schemes simultaneously, by relying on machine-learning techniques to determine, on a per transaction basis, the optimal certification strategy to be adopted.

In certification-based protocols transactions are validated at commit time and may be re-executed an unbounded number of times due to conflicts, leading to an undesirably high abort rate. The work in [3] tackles these issues using the concept of lease, informally, a token which gives its holder the privileges to manage a given subset of the whole data set. When certifying a transaction, replicas must first acquire the corresponding leases if not already in possession. Once in possession of the leases, replicas can certify transactions using reliable broadcast instead which is cheaper than total-order broadcast. If a transaction is aborted, the host replica re-executes it without relinquishing the leases.

In [4] the authors exploit the optimistic atomic broadcast primitive in order to reduce the message deliver latency [14]. As soon as a transaction t is optimistically delivered, t is speculatively certified instead of waiting for its final delivery as in conventional certification protocols. This allows an overlap between computation and communication by certifying transactions while the optimistic atomic broadcast computes the final order.

7 Concluding Remarks

STM replication based on certification protocols rely on total-order broadcast. This paper presents, to the best of our knowledge, the first study of the impact that different total-order broadcast implementations can have on several benchmarks used in the literature, each with different workload characteristics. We have found that the same system exhibits different behaviour with regard to performance, fairness and latency, depending on the combination of total-order broadcast implementation and workload characteristics.

These observations open up further work. One can exploit the low latency of the sequencer to schedule update transactions exclusively to the sequencer and read-only transactions (which do not require any remote coordination) to the remaining replicas, which only apply the updates in the background. Additionally, since the remote coordination overhead is very high relative to transaction execution time, and read-only transactions execute exclusively locally, we can execute

read-only transactions while waiting for the delivery of update transactions. This technique is specially appealing for non-sequencer-based implementations, such as Spread's, because latency is higher.

References

1. Shavit, N., Touitou, D.: Software Transactional Memory. In: Symposium on Principles of Distributed Computing (PODC), pp. 204–213 (1995)
2. Couceiro, M., Romano, P., Carvalho, N., Rodrigues, L.: D^2STM: Dependable Distributed Software Transactional Memory. In: IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), pp. 307–313 (2009)
3. Carvalho, N., Romano, P., Rodrigues, L.: Asynchronous Lease-Based Replication of Software Transactional Memory. In: Gupta, I., Mascolo, C. (eds.) Middleware 2010. LNCS, vol. 6452, pp. 376–396. Springer, Heidelberg (2010)
4. Carvalho, N., Romano, P., Rodrigues, L.: SCert: Speculative certification in replicated software transactional memories. In: International Systems and Storage Conference (SYSTOR) (2011)
5. Couceiro, M., Romano, P., Rodrigues, L.: PolyCert: Polymorphic Self-Optimizing Replication for In-Memory Transactional Grids. In: Kon, F., Kermarrec, A.-M. (eds.) Middleware 2011. LNCS, vol. 7049, pp. 309–328. Springer, Heidelberg (2011)
6. Défago, X., Schiper, A., Urbán, P.: Total order broadcast and multicast algorithms. ACM Computing Surveys 36(4), 372–421 (2004)
7. Agrawal, D., Alonso, G., El Abbadi, A., Stanoi, I.: Exploiting atomic broadcast in replicated databases. In: European Conference on Parallel and Distributed Computing (Euro-Par), pp. 496–503 (1997)
8. Kemme, B., Alonso, G.: A suite of database replication protocols based on group communication primitives. In: International Conference on Distributed Computing Systems (ICDCS), pp. 156–163 (1998)
9. Dias, R.J., Vale, T.M., Lourenço, J.M.: Efficient Support for In-Place Metadata in Transactional Memory. In: Kaklamanis, C., Papatheodorou, T., Spirakis, P.G. (eds.) Euro-Par 2012. LNCS, vol. 7484, pp. 589–600. Springer, Heidelberg (2012)
10. Dice, D., Shalev, O., Shavit, N.N.: Transactional Locking II. In: Dolev, S. (ed.) DISC 2006. LNCS, vol. 4167, pp. 194–208. Springer, Heidelberg (2006)
11. Miranda, H., Pinto, A., Rodrigues, L.: Appia, a flexible protocol kernel supporting multiple coordinated channels. In: Poster on the International Conference on Distributed Computing Systems (ICDCS), pp. 707–710 (2001)
12. Minh, C.C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford Transactional Applications for Multi-Processing. In: IEEE International Symposium on Workload Characterization (IISWC), pp. 35–46 (2008)
13. Kaashoek, M.F., Tanenbaum, A.S.: An Evaluation of the Amoeba Group Communication System. In: International Conference on Distributed Computing Systems (ICDCS), pp. 436–447 (1996)
14. Pedone, F., Schiper, A.: Optimistic atomic broadcast: a pragmatic viewpoint. Theoretical Computer Science 291(1), 79–101 (2003)