D^2STM: Dependable Distributed Software Transactional Memory [*]

Maria Couceiro
maria.couceiro@ist.utl.pt

Paolo Romano
romanop@gsd.inesc-id.pt

Nuno Carvalho
nonius@gsd.inesc-id.pt

Luis Rodrigues
ler@ist.utl.pt

Abstract

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications. At current date, however, the problem of how to build distributed and replicated STMs to enhance *both* dependability and performance is still largely unexplored. This paper fills this gap by presenting D^2STM, a replicated STM that makes use of the computing resources available at multiple nodes of a distributed system. The consistency of the replicated STM is ensured in a transparent manner, even in the presence of failures. In D^2STM transactions are autonomously processed on each node, avoiding any replica inter-communication during transaction execution, and without incurring in deadlocks. Strong consistency is enforced at transaction commit time by a non-blocking distributed certification scheme, which we name BFC (Bloom Filter Certification). BFC exploits a novel Bloom Filter-based encoding mechanism that permits to significantly reduce the overheads of replica coordination at the cost of a user tunable increase in the probability of transaction abort. Through an extensive experimental study based on standard STM benchmarks we show that the BFC scheme permits to achieve remarkable performance gains even for negligible (e.g. 1%) increases of the transaction abort rate.

Keywords: Dependability, Software Transactional Memory, Replication, Bloom Filters

1 Introduction

Software Transactional Memory (STM) systems have emerged as a powerful paradigm to develop concurrent applications [23, 21, 17]. When using STMs, the programmer is not required to deal explicitly with concurrency control mechanisms. Instead, she has only to identify the sequence of instructions, or

[*] This work was partially supported by the Pastramy (PTDC/EIA/72405/2006) project.

transactions, that need to access and modify concurrent objects atomically. As a result, the reliability of the code increases and the software development time is shortened.

While the study of STMs has garnered significant interest, the problem of architecting distributed STMs has started to receive the required attention only very recently [31, 8, 28]. Furthermore, the solutions proposed so far have not addressed the important issue of how to leverage replication not only to improve performance, but also to enhance dependability. This is however a central aspect of distributed STM design, as the probability of failures increases with the number of nodes and becomes impossible to ignore in large clusters (composed of hundreds of nodes [8]). Strong consistency and fault-tolerance guarantees are also essential when STMs are used to increase the robustness of classic service-oriented applications. This is the case, for instance, of the FenixEDU system [13], a complex web-based Campus activity management system that is currently used in several Portuguese universities. FenixEDU extensively relies on STM technology for transactionally manipulating the in-memory state of its (J2EE compliant) application server. Providing critical services (such as students' grading or research funds management) to a population of more than 14000 users, the FenixEDU system deployed at the IST Campus of Lisbon is one of the main drivers of our research in the quest for efficient and scalable replication mechanisms [10].

This paper addresses the problems above by introducing D^{2}STM, a Dependable Distributed Software Transactional Memory that allows programmers to leverage on the computing resources available in a cluster environment, using a conventional STM interface, transparently ensuring non-blocking and strong consistency guarantees even in the case of failures.

The replica synchronization scheme employed in D^{2}STM is inspired by recent database replication approaches [35, 26, 34], where replica consistency is achieved through a distributed certification procedure which, in turn, leverages on the properties of an Atomic Broadcast [16] primitive. Unlike classic eager replication schemes (based on fine-grained distributed locking and atomic commit), that suffer of large communication overheads and fall prey of distributed deadlocks [18], certification based schemes avoid any onerous replica coordination during the execution phase, running transactions locally in an optimistic fashion. The consistency of replicas (typically, 1-Copy serializability) is ensured at commit-time, via a distributed certification phase that uses a single Atomic Broadcast to enforce agreement on a common transaction serialization order, avoiding distributed deadlocks, and providing non-blocking guarantees in the presence of (a minority of) replica failures. Furthermore, unlike classic read-one/write-all approaches that require the full execution of update transactions at all replicas [6], only one replica executes an update transactions, whereas the remaining replicas are only required to validate the transaction and to apply the resulting updates. This allows to achieve high scalability levels even in the presence of write-dominated workloads, as long as the transaction conflict rate remains moderate [35].

For the reasons above, certification based replication schemes appear attractive to apply in the STM context. Unfortunately, as previously observed in [38] (and confirmed by the experimental results presented later in this paper), the overhead of previously published Atomic Broadcast based certification schemes can be particularly detrimental in STM environments. In fact, unlike in classical database systems, STMs incur neither in disk access latencies nor in the overheads of SQL statement parsing and plan optimization. This makes the execution time of typical STM transactions normally much shorter than in database settings [38] and leads to a corresponding amplification of the overhead of inter-replica coordination costs. To tackle this issue, D^{2}STM, leverages a novel transaction certification procedure, named BFC (Bloom Filter Certification), which takes advantage of a space-efficient Bloom Filter-based encoding to significantly reduce the overhead of the distributed certification scheme at the cost of a marginal, and user configurable, increase of the transaction abort probability.

D^{2}STM is built on top of JVSTM [12], an efficient STM library that supports multi-version concurrency control and, as a result, offers excellent performance for read-only transactions. D^{2}STM takes full advantage of the JVSTM's multi-versioning scheme, sheltering read-only transactions from the possibility of aborts due both to local or remote conflicts. Through an extensive experimental evaluation, based on both synthetic micro-benchmarks, as well as complex STM benchmarks we show that D^{2}STM permits to achieve significant performance gains at the cost of a marginal growth of the abort rate.

The rest of this paper is organized as follows. Section 2 discusses related work. A formal description of the considered system model and of the consistency criteria ensured by D^{2}STM is provided in Section 3, whereas Section 4 overviews the whole architecture of the D^{2}STM system and discusses the issues related to the integration of JVSTM within D^{2}STM. The BFC scheme is presented in Section 5 and Section 6 presents the results of our experimental evaluation study. Finally, Section 7 concludes the paper.

2 Related Work

In this section we briefly survey related research. We begin by analyzing the main design choices of existing distributed STM systems, critically highlighting their main drawbacks from both the fault-tolerance and performance perspectives. Next we review recent literature on database replication schemes, discussing pros and cons of these approaches when adopted in a distributed STM context. Finally, we discuss other works related to D^{2}STM in a wider sense.

2.1 Distributed STMs

The only distributed STM solutions we are aware of are those in [28, 8, 31]. As already noted in the introduction, unlike D^{2}STM, none of these solutions leverages on replication in order to ensure cluster-wide consistency and availability in scenarios of failures, or failure suspicions. While it could be possible to somehow extend the distributed STM solutions proposed in these works with orthogonal fault-tolerance mechanisms, this is far from being a trivial task and, perhaps more importantly, the overhead associated with these additional mechanisms could seriously hamper their performance. In D^{2}STM, on the other hand, dependability is seen as a first class design goal, and the STM performance is optimized through a holistic approach that tightly integrates low level fault-tolerance schemes (such as Atomic Broadcast) with a novel, highly efficient distributed transaction certification scheme.

In the following, we critically highlight the most relevant differences, from a performance oriented perspective, of the replica coherency schemes adopted by the aforementioned schemes with respect to D^{2}STM during failure-free runs. The work in [31] exploits the simultaneous presence of different versions of the same transactional dataset across the replicas, to implement a distributed multi-versioning scheme (DMV). Like centralized multi-version concurrency control schemes [6] (including JVSTM [12]), DMV allows read-only transactions to be executed in parallel with conflicting updating transactions. This is achieved by ensuring that the former is able to access older, committed snapshots of the dataset. However, in DMV each replica maintains only a single version of each data granule, and explicitly delays applying (local or remote) updates to increase the chance of not having to invalidate the snapshot of currently active read-only transactions (and to consequently abort them). This allows DMV to avoid maintaining multiple versions of the same data at each node, unlike in conventional multi-version concurrency control solutions (although DMV requires buffering the updates of not yet applied transactions). On the other hand, while multi-version concurrency control solutions provide deterministic guarantees on the absence of aborts for read-only transactions, the effectiveness of the DMV scheme depends on the timing of the concurrent accesses to data by conflicting transactions (actually, with DMV a read-only transaction may be aborted also due to the concurrent execution of “younger”, local read-only transaction). Optimizing the performance of read-only transactions, which largely dominate in many realistic workloads, is an important design goal common to both DMV and D^{2}STM. However, D^{2}STM relies on a multi-versioned STM, namely JVSTM, which maintains a sufficient number of versions of each transactionalized data item in order to *guarantee* that no read-only transaction is ever aborted. Further, this is done in an autonomous manner by the local STM, in a transparent manner for the replication logic, greatly simplifying the design and implementation of the whole system. Another significant difference between D^{2}STM and DMV is in that the latter requires each committing transaction to acquire a cluster-wide unique token, which globally serializes the commit phases of transactions. Unfortunately, given that committing a transaction imposes a two communication step synchronization phase (for updates propagation), the token acquisition phase can introduce considerable overhead and seriously hamper performance [28]. Conversely, in D^{2}STM the Atomic Broadcast-based replica coordination phase can be executed in full concurrency by the various replicas, which are required to sequentially execute only the local transaction validation phase aimed at verifying whether a committing transaction must be aborted due to some conflict.

The work in [28] does not rely on multi-versioning schemes, but, analogously to the one in [31], relies on a distributed mutual exclusion mechanism scheme. Mutual exclusion is aimed at ensuring that at any time there are no two replicas attempting to simultaneously commit *conflicting* transactions. The use of multiple leases, based on the actual datasets accessed by transactions, permits to partially alleviate the performance problems incurred by the serialization of the whole (distributed) commit phase. However, this phase may still become a bottleneck with conflict intensive workloads. As already discussed, this problem is completely circumvented in D^{2}STM thanks to the use of an Atomic Broadcast based certification procedure. Additionally, in [28] the lease establishment mechanism is coordinated by a single, centralized, node which is likely to become a performance bottleneck for the whole system as the number of replicas increase; In fact, the experimental evaluation in [28] relies on a dedicated node for lease management and does not report results for more than four replicas.

Finally, Cluster-STM, presented in [8], focuses on the problem of how to partition the dataset across the nodes of a large scale distributed Software Transactional Memory. This is achieved by assigning to each data item a home node, which is responsible for maintaining the authoritative version (and the associated metadata) of the data item. The home node is also in charge of synchronizing the accesses of conflicting remote transactions. In [8] any caching or replication scheme is totally delegated to the application level, which has then to take explicitly into account the issues related to data fetching and distribution, with an obvious increase in the complexity of the application development. Currently, D^{2}STM only provides support for total replication of the transactional dataset (even though leveraging transparent, selective replication of data across the nodes represents part of our future work). On the other hand, D^{2}STM provides programmers with the powerful abstraction of single system image, which permits to port applications previously running on top of non distributed STMs with minimal modifications. Further, Cluster-STM treats the processors as a flat set, not distinguishing between processors within a node and processors across nodes, and not exploiting the availability of shared memory between multiple cores/processors on each replica to speed up intra-node communication. Finally, Cluster-STM does not exploit a multi-versioning local concurrency control to maximize the performance of read-only transactions, and is constrained to run only a single thread for each processor. Being layered on top of a fully fledged, multi-version STM, D^{2}STM overcomes all of the above limitations.

2.2 Database Replication

The problem of replicating a STM is naturally closely related to the problem of database replication, given that both STMs and DBs share the same key abstraction of atomic transactions. The fulcrum of modern database replication schemes [35, 34, 15, 2, 26] is the reliance on an Atomic Broadcast (ABcast) primitive [16, 20], typically provided by some Group Communication System (GCS) [33, 4]. ABcast plays a key role to enforce, in a non-blocking manner, a global transaction serialization order without incurring in the scalability problems affecting classical eager replication mechanisms based on distributed locking and atomic commit protocols, which require much finer grained coordination and fall prey of deadlocks [18]. Existing ABcast-based database replication literature can be coarsely classified in two main categories, depending on whether transactions are executed optimistically [35, 26] or conservatively [27].

In the conservative case, which can be seen as an instance of the classical state machine/active replication approach [39], transactions are serialized through ABcast *prior* to their actual execution and are then deterministically scheduled on each replica in compliance with the ABcast determined serialization order. This prevents aborts due to concurrent execution of conflicting transactions in different replicas and avoids the cost of broadcasting the transactions' read-sets and write-sets. On the other hand, the need for enforcing deterministic thread scheduling at each replica requires a careful identification of the conflict classes to be accessed by each transaction, prior to its actual execution. Unfortunately, this requirement represents a major hurdle for the adoption of these techniques in STM systems which, unlike relational DBMSs with SQL-like interfaces, allow users to define arbitrary, and much less predictable, data layouts and transaction access patterns (e.g. determined trough direct pointer manipulations). In practice, it is very hard or simply impossible to predict the datasets that are to be accessed by a newly generated transaction. This is particular troublesome, given that a labeling error can lead to inconsistency, whereas coarse overestimations can severely limit concurrency and hamper performance.

Optimistic approaches, such as [35], avoid these problems, hence appearing better suited to be adopted also in STM contexts. In these approaches, transactions are locally processed on a single replica and validated *a posteriori* of their execution through an ABcast based certification procedure aimed at detecting remote conflicts between concurrent transactions. The certification based approaches can be further classified into voting and non-voting schemes [26, 37], where voting schemes, unlike non-voting ones, need to atomic broadcast only the write-set (which is typically much smaller than the read-set in common workloads), but on the other hand incur the overhead of an additional uniform broadcast [20] along the critical path of the commit phase. As highlighted in our previous work [38], the replica coordination latency has an amplified cost in STM environments when compared to conventional database environments, given that the average transaction execution time in STM settings is typically several orders of magnitude shorter than in database applications. This makes voting certification schemes, which introduce an additional latency of at least 2 extra communication steps with regard to non voting protocols, unattractive in replicated STM environments. On the other hand, as it will be demonstrated by our experimental study, and as one could intuitively expect, the actual efficiency of non voting certification protocols is, in practical settings, profoundly affected by the actual size of read-sets.

The replica coordination scheme employed in D^{2}STM, namely BFC (Bloom Filter Certification), can be classified as a non voting certification scheme that exploits a Bloom Filter based encoding of the transactions' read-set to achieve the best of both the voting and not voting approaches, requiring only a single ABcast while avoiding to flood the network with large messages, at the cost of a small, and user tunable increase in the transactions abort rate.

2.3 Other Related Works

The large body of literature on Distributed Shared Memories (DSM) is clearly related to our work, sharing our same base goal of providing developers with the simple abstraction of a single system image transparently leveraging the resources available across distributed nodes. To overcome the strong performance overheads introduced by straightforward DSM implementations [30] ensuring strong consistency guarantees with the granularity of a single memory access [29], several DSM systems have been developed that achieve better performance through relaxing memory consistency guarantees [25]. Unfortunately, developing software for relaxed DSM's consistency models can be challenging as programmers are required to fully understand sometimes complicated consistency properties to maximize performances without endangering correctness. Conversely, the simplicity of the atomic transaction abstraction, at the core of STMs and of our D^{2}STM platform, allows to increase programmers' productivity [11] with respect to both locking disciplines and relaxed memory consistency models. Further, the strong consistency guarantees provided by atomic transactions can be supported through efficient algorithms that, like in D^{2}STM, incur only in a single synchronization phase per transaction, effectively amortizing the unavoidable communication overhead across a set of (possibly large) memory accesses.

Finally, the notion of atomic transaction plays a key role also in the recent Sinfonia [3] platform, where these are referred to as "mini-transactions". However, unlike in conventional STM settings and in D^{2}STM, Symphonia assumes transactions to be static, i.e. that their datasets and operations are known in advance, which limits the generality of this solution.

3 System Model

We consider a classical asynchronous distributed system model [20] consisting of a set of processes $\Pi = \{p_1, \ldots, p_n\}$ that communicate via message passing and can fail according to the fail-stop (crash) model. We assume that a majority of processes is correct and that the system ensures a sufficient synchrony level (e.g. the availability of a $\Diamond S$ failure detector) to permit implementing an Atomic Broadcast (ABcast) service, with the following properties [16]: *Validity*: If a correct participant broadcasts a message, then all correct participants eventually deliver it. *Uniform Agreement*: If a participant delivers a message, then all correct participants eventually deliver it. *Uniform Integrity*: Any given message is delivered by each participant at most once, and only if it was previously broadcast. *Uniform Total Order*: If

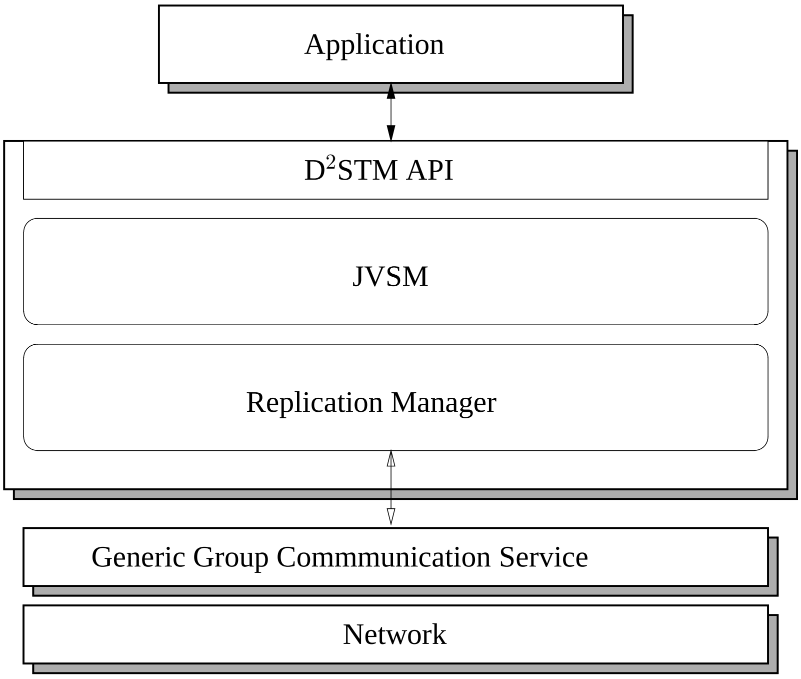


Figure 1: Components of a D^{2}STM replica.

some participant delivers message A after message B, then every participant delivers A only after it has delivered B.

D^{2}STM preserves the weak atomicity [32] and opacity [19] properties of the underlying JVSTM. The former property implies that atomicity is guaranteed only as to conflicting pairs of transactional accesses; conflicts between transactional and non-transactional accesses are not protected. Weak atomicity is less composable than strong atomicity (protecting all pairs where at least one is a transactional access). It also raises subtle problems, e.g., granular lost updates. However, the runtime overhead of strong atomicity can be prohibitively high in the absence of hardware support [32]. Opacity [19], on the other hand, can be informally viewed as an extension of the classical database serializability property with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states.

Finally, concerning the consistency criterion for the state of the replicated (JV)STM instances, D^{2}STM guarantees 1-copy serializability of reads and writes to transactional data [6], which ensures that transaction execution history across the whole set of replicas is equivalent to a serial transaction execution history on a not replicated (JV)STM.

4 D^{2}STM Architecture

4.1 Node Components

The components of a node of the D^{2}STM platform, depicted in Figure 1, is structured into 4 main logical layers. The bottom layer is a Group Communication Service (GCS) [16] which provides two main building blocks: view synchronous membership [20], and an Atomic Broadcast service. Our implementation uses a generic group communication service (GCS) [14], which supports multiple implementations of the GCS (all the experiments described in this paper have been performed using the Appia GCS [33]). The core component of D^{2}STM is represented by the Replication Manager, implementing the distributed coordination protocol required for ensuring replica consistency (i.e. 1-copy serializability); this component is described in detail in Section 5. The Replication Manager interfaces, on one side, the GCS layer and, on the other side, with a local instance of a Software Transactional Memory, more precisely JVSTM [11]. A detailed discussion of the integration between the replication manager and JVSTM, along with a summary of the most relevant JVSTM internal mechanisms, is provided in Section 4.2. Finally, the top layer of D^{2}STM is a wrapper that intercepts the application level calls for transaction demarcation (i.e. to begin, commit or abort transactions), not interfering at all with the application accesses (read/write) to the VBoxes which are managed directly by the underlying JVSTM layer. This approach allows D^{2}STM to transparently extend the classic STM programming model, while requiring only minor modifications to pre-existing JVSTM applications.

4.2 Integration with JVSTM

JVSTM implements a multi-version scheme which is based on the abstraction of a *versioned box* (VBox) to hold the mutable state of a concurrent program. A VBox is a container that keeps a tagged sequence of values - the history of the versioned box. Each of the history's values corresponds to a change made to the box by a successfully committed transaction and is tagged with the timestamp of the corresponding transaction. To this end, JVSTM maintains an integer timestamp, *commitTimestamp*, which is incremented whenever a transaction commits. Each transaction stores its timestamp in a local *snapshotID* variable, which is initialized at the time of the transaction activation with the current value of *commitTimestamp.* This information is used both during transaction execution, to identify the appropriate values to be read from the VBoxes, and, at commit time, during the validation phase, to determine the set of concurrent transactions to check against possible conflicts. JVSTM relies on an optimistic approach which buffers transactions' writes and detects conflicts only at commit time, by checking whether any of the VBoxes read by a committing transaction T was updated by some other transaction T' with a larger timestamp value. In this case T is aborted. Otherwise, T's *commitTimestamp* is increased, its *snapshotID* is set to the new value of *commitTimestamp* and the new values of all the VBoxes it updated are atomically stored within the VBoxes.

To minimize performance overheads, the D^{2}STM's replica coordination protocol, namely BFC, is tightly integrated with the JVSTM's transaction timestamping mechanisms. The integration of JVSTM within the D^{2}STM required the implementation of three main (non-intrusive) modifications to JVSTM, extending its original API in order to allow the Replication Manager layer to:

1. extract information concerning internals of the transaction execution, i.e., its read-set, write-set, and *snapshotID* timestamp. In the remaining, we refer to the methods providing the aforementioned services for a transaction T_x, respectively, as *getReadset(Transaction T_x), getWriteset(Transaction T_x)* and *getSnapshotID(Transaction T_x).*

2. explicitly trigger the transaction validation procedure (method *validate(Transaction T_x)*), that aims at detecting any conflict raised during the execution phase of a transaction T_x with any other (local or remote) transaction that committed after T_x started.

3. atomically apply, through the *applyRemoteTransaction(Writeset WS)* method, the write-set WS of a remotely executed transaction (i.e. atomically updating the VBoxes of the local JVSTM with the new values written by a remote transaction) and simultaneously increasing the JVSTM's *commitTimestamp.*

4. permit cluster wide unique identification of the VBoxes updated by (remote) transactions, as well as of any object, possibly dynamically generated within a (remote) transaction, whose reference could be stored within a VBox. This is achieved by tagging each JVSTM VBox (and each object, mutable or immutable, assigned to a VBox within a Transaction) with a unique identifier. A variety of different schemes may be used to generate universal unique identifiers (UIDs), as long as it is possible to guarantee the cluster-wide uniqueness of UIDs and to enable the independent generation of UIDs at each replica. The current implementation of D^{2}STM relies on a widely recognized international standard, namely the ISO/IEC 11578:1996[1], which uses a 128 bits long encoding scheme[2] that includes the identifier of the generating node and a local timestamp based on a 100-nanosecond intervals.

5 Bloom Filter Certification

Bloom Filter Certification (BFC) is a novel non-voting certification scheme that exploits a space-efficient Bloom Filter-based encoding [7], allowing to drastically reduce the overhead of the distributed certification phase at the cost of a reduced (but controlled) increase in the risk of transaction aborts.

[1] Also ITU-T Rec. X.667 - ISO/IEC 9834-8:2005, and integrated within the official Java library since version 1.5.

[2] The standard Leach-Salz variant layout encoding was used.

```
int oldestActiveXact[$n$]={0,. . .,0};
Set ActiveXacts, CommittedXacts, AbortedXacts=$\emptyset$;
int avgBFQueries=$initialAvgBFQueries$;

Transaction begin()
  Transaction $T_x$=JVSTM.begin();
  ActiveXacts=ActiveXacts$\cup T_x$;
  return $T_x$;

boolean commit(Transaction $T_x$)
    // Read-only transactions are processed locally
  if (getWriteset($T_x$)=$\emptyset$)
    ActiveXacts=ActiveXacts$\setminus T_x$;
    return true;
    // Update transactions are first locally validated
  if ($\neg$validate($T_x$))
    ActiveXacts=ActiveXacts$\setminus T_x$;
    return false;
  int BFSize=estimateBFSize(avgBFQueries);
  BloomFilter BF=new BloomFilter(BFSize);
  $\forall UID \in getReadset(T_x)$ BF.add($UID$);
  AB-send[$T_x$, getSnapshotID($T_x$), BF, getWriteset($T_x$),
      min $_{T_y \in ActiveXacts}$( getSnapshotID($T_y$) ) ];
  // The xact's outcome is determined upon AB-delivery
  wait $T_x \in$ ( AbortedXacts $\cup$ CommittedXacts )
  ActiveXacts=ActiveXacts$\setminus T_x$;
  if (T$_x \in AbortedXacts$)
    AbortedXacts=AbortedXacts$\setminus T_x$;
    return false;
  else return true;
```

```
upon AB-deliver[Transaction $T_x$, int snapshotID, BloomFilter BF,
                    WriteSet WS, int oldestActiveXact] from $p_j$ do
    // Garbage collect the CommittedXacts set
  if (oldestActiveXact<oldestActiveXact[j])
    oldestActiveXact[j]=oldestActiveXact;
    $\forall T_k \in$ CommittedXacts s.t.
      getSnapshotID($T_k$)$\leq$ min$_{i\in[1,n]}$(oldestActiveXact[i]) do
        CommittedXact=CommittedXact$\setminus T_k$;

    // Validate Transaction
  int BFQueries=0;
  $\forall T_y \in$ CommittedXacts s.t. getSnapshotID($T_k$)>snapshotID do
    $\forall < UID, \cdot > \in getWriteset(T_y)$ do
      BFQueries++;
      if ($BF.contains(UID)$)
        // Xact failed validation and is aborted
        AbortedXacts=AbortedXacts$\cup\{T_x\}$
        if (isLocal($T_x$))
          ActiveXacts=ActiveXacts$\setminus T_x$;
        return;

    // Xact passed validation: update estimator for q and commit xact
  avgBFQueries=updateAvg(BFQueries,$recComXacts$);
  CommittedXacts=CommittedXacts$\cup T_x$;
  if (isLocal($T_x$))
    ActiveXacts=ActiveXacts$\setminus T_x$;
    JVSTM.commit($T_x$);
  else
    applyRemoteTransactionWS(WS);
```

Figure 2: Pseudo-code of the BFC algorithm executed by the Replication Manager at Process p$_i$

Before delving into the details of the BFC protocol, we review the fundamentals of Bloom filters (the interested reader may refer to [9] for further details). A Bloom filter for representing a set $S = \{x_1, x_2, \ldots, x_n\}$ of n elements from a universe U consists of an array of m bits, initially all set to 0. The filter uses k independent hash functions $h_1, \ldots, h_k$ with range $\{1, \ldots, m\}$, where it is assumed that these hash functions map each element in the universe to a random number uniformly over the range. For each element $x \in S$, the bits $h_i(x)$ are set to 1 for 1$\leq i \leq k$. To check if an item y is in S, we check whether all $h_i(y)$ are set to 1. If not, then clearly y is not a member of S. If all $h_i(x)$ are set to 1, x is assumed to be in S, although this may be wrong with some probability. Hence a Bloom filter may yield a *false positive*, where it suggests that an element x is in S even though it is not. The probability of a false positive f for a single query to a Bloom Filter depends on the number of bits used per item m/n and the number of hash functions k according to the following equation:

$$f = (1 - e^{-kn/m})^k \tag{1}$$

where the optimal number k of hash functions that minimizes the false positive probability f given m and n can be shown to be equal to:

$$k = \lceil ln\ 2 \cdot m/n \rceil \tag{2}$$

We now describe BFC in detail, with the help of the pseudo-code depicted in Figure 2. Read-only transactions are executed locally, and committed without incurring in any additional overhead. Leveraging on the JVSTM multi-version scheme, D^{2}STM read-only transactions are always provided with a consistent committed snapshot and are spared from the risk of aborts (due to both local or remote conflicts).

A committing transaction with a non-null write-set (i.e. it has updated some VBox), is first locally validated to detect any local conflicts. This prevents the execution of the distributed certification scheme for transactions that are known to abort using only local information. If the transaction passes the local validation phase, the Replication Manager encodes the transaction read-set (i.e., the set of identifiers of all the VBoxes read by the transaction) in a Bloom Filter, and ABcasts it along with the transaction write-set (which is not encoded in the Bloom Filter). The size of the Bloom Filter encoding the transaction's read-set is computed to ensure that the probability of a transaction abort due to a Bloom Filter's false positive is less than a user-tunable threshold, which we denote as *maxAbortRate.* The logic for sizing of the Bloom Filter is encapsulated by the *estimateBFSize()* primitive, which will be detailed later in the text.

As in classical non-voting certification protocols, update transactions are validated upon their ABcast-delivery. At this stage, it is checked whether T_x's Bloom Filter contains any item updated by transactions with a *snapshotID* timestamp larger than that of T_x's. If no match is found, then T_x can be safely committed. Committing a transaction T_x consists of the following steps. If T_x is a local transaction, it just suffices to request the local JVSTM to commit it. If, on the other hand, T_x is a remote transaction, its write-set is atomically applied using the *applyRemoteTransaction(WS$_{T_x}$)* method.

Given that the validation phase of a transaction T_x requires the availability of the write-sets of concurrent transactions previously committed, the Replication Manager locally buffers the UIDs of the VBoxes updated by any committed transaction in the *CommittedXacts* set. To avoid an unbounded growth of this data structure, we rely on a distributed garbage collection scheme (analogous to the one employed in [36]), in which each replica exchange (as a piggyback to the AB-casted transaction validation message) the minimum *snapshotID* of all the locally active update transactions. This allows each replica to gather global knowledge on the oldest timestamp among those of all the update transactions currently active on any replica. This information is used to garbage collect the *CommitXacts* set by removing the information associated with any committed transactions whose execution can no longer invalidate any of the active transactions.

We now describe how the size of the Bloom Filter (BF) of a committing transaction is computed. The reader should note that for a transaction T_x to be aborted due to a false positive it is sufficient to incur in a false positive for any of the items updated by transactions concurrent with T$_x$'s. In other words, determining the size of the Bloom Filter for a committing transactions, so to guarantee that a target *maxAbortRate* is never exceeded, would require to know *exactly* the number q of queries that will have to be performed against the Bloom Filter once the transaction gets validated (i.e. once it is ABcast-delivered). On the other hand, at the time in which T$_x$ enters the commit phase, it is not possible to exactly foresee neither how many transactions will commit before T$_x$ is ABcast-delivered, nor what will be the size of the write-sets of each of these transactions. On the other hand, any error in estimating q does not compromise safety, but may only lead to (positive or negative) deviations from the target *maxAbortRate* threshold. Hence, BFC uses a simple and lightweight heuristic, which exploits the fact that each replica can keep track of the number of queries performed to the BF of any locally ABcast-delivered transaction. In detail, we rely on the moving average across the number of BF queries performed during the validation of phase of the last *recComXacts* transactions as an estimator of q. Once q is estimated, we can then determine the number m of bits in the Bloom Filter by considering that the false positives for any distinct query are independent and identically distributed events which generate a Bernoullian process. At the light of this observation, the probability of aborting a transaction because of a false positive in the Bloom Filter-based validation procedure, *maxAbortRate*, can be expressed as:

$$\mathit{maxAbortRate} = 1 - (1 - f)^q$$

which, combined with Equations 1 and 2, allows us to estimate m as:

$$m = \left\lceil -n \, \frac{log_2(1 - (1 - \mathit{maxAbortRate})^{\frac{1}{q}})}{ln\ 2} \right\rceil$$

The striking reduction of the amount of information exchanged, achievable by the BFC scheme, is clearly highlighted by the graph in Figure 3, which shows the BFC's compression factor (defined as the ratio between the number of bits for encoding a transaction's read-set with the ISO/IEC 11578:1996 standard UID encoding, and with BFC) as a function of the target *maxAbortRate* parameter and of the number q of queries performed during the validation phase. The plotted data shows that, even for marginal increases of the transaction abort probability in the range of [1%-2%], BFC achieves a [5x-12x] compression factor, and that the compression factor extends up to 25x in the case of 10% probability of transaction aborts induced by a false positive of the Bloom Filter.

The correctness of the BFC scheme can be (informally) proved by observing that i) replicas validate all write transactions in the sme order (the one determined by the Atomic Broadcast primitive), and that, ii) the validation procedure, despite being subject to false positives, is deterministic given that all replicas rely on the same set of hash functions to query for the presence/determine the encoding of data items in the Bloom filter. Hence, as already highlighted, the occurrence of false positives results in an increase of the transaction abort rate, but can never lead to inconsistencies of the replicas' states.

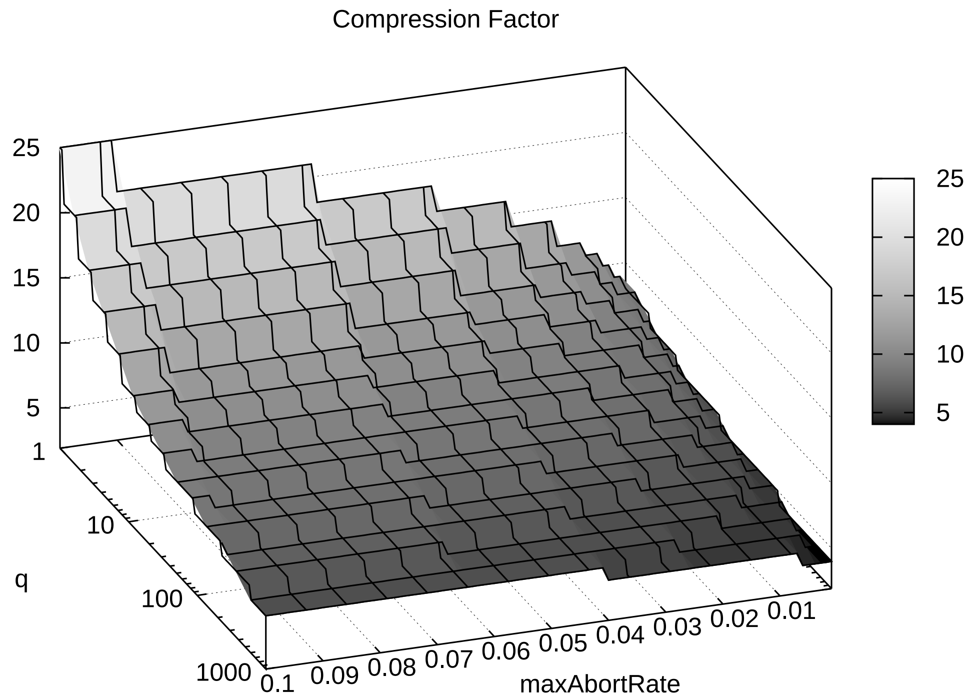


Figure 3: Compression Factor achieved by BFC considering the ISO/IEC 11578:1996 UUID encoding.

As a final note, in order to speed up the Bloom Filter construction (more precisely the insertion of items within the Bloom Filter), D^{2}STM exploits a recently proposed optimization [1] which generates the $k = \lceil ln\ 2 \cdot m/n \rceil$ hash values required for encoding a data item within the Bloom Filter via a plain (and very efficient) linear combination of the output of only two independent hash functions. The choice of the hashing algorithm to be employed within D^{2}STM has been based on an experimental comparison of a spectrum of different hash functions trading off complexity, speed, and collision resistance. The one that exhibited the best performance while matching the analytically forecast false positive probability turned out to be MurmurHash2 [5], a simple, multiplicative hash function whose excellent performances have been also confirmed by some recent benchmarking results [24].

6 Evaluation

We now report results of an experimental study aimed at evaluating the performance gains achieved by the BFC scheme in a real distributed STM system, namely when using our D^{2}STM prototype, in face of a variety of both synthetic and more complex STM workloads. These results allow to assess the practical impact of the benefits estimated in the previous section, using the analytical model. The target platform for these experiments is a cluster of 8 nodes, each one equipped with an Intel QuadCore Q6600

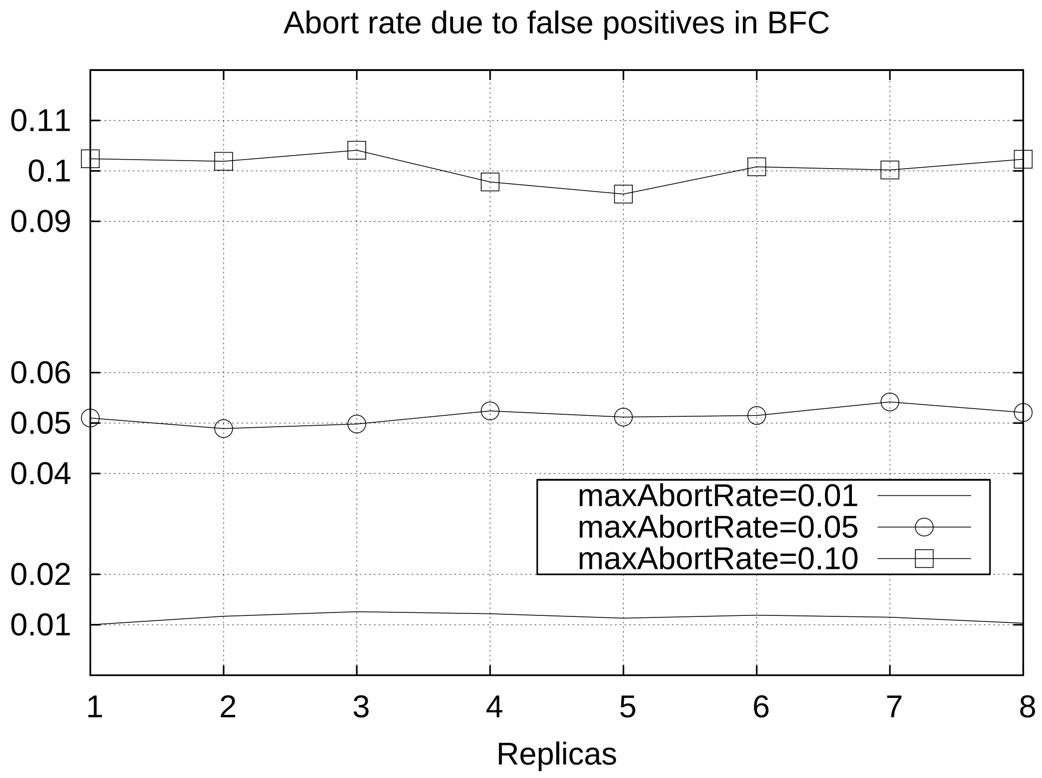


Figure 4: Transaction abort rate due to false positives in the Bloom Filter-based validation.

at 2.40GHz equipped with 8 GB of RAM running Linux 2.6.27.7 and interconnected via a private Gigabit Ethernet. The Atomic Broadcast implementation used is based on a classic sequencer-based algorithm [20, 16].

We start by considering a synthetic workload (obtained by adapting the Bank Benchmark originally used for evaluating DSTM2 [22]) which serves for the sole purpose of validating the analytical model introduced in Section 5 for determining the Bloom Filter's size as a function of a target *maxAbortRate* factor. In detail, we initialize the STM at each replica with a vector of $\mathit{numThreads} \cdot \mathit{numMachines} \cdot 10.000$ items. Each thread $i \in [0, \mathit{numThreads} - 1]$ executing on replica $j \in [0, \mathit{numMachines} - 1]$ accesses a distinct fragment (of indexes $[(i + j \cdot \mathit{numThreads}) \cdot 10.000, (1 + i + j \cdot \mathit{numThreads}) \cdot 10.000 - 1]$) of 10.000 elements of the array, reading all these elements and randomly updating a number of elements uniformly distributed in the range [50-100]. Given that the fragments of the array accessed by different threads never overlap, this ensures that any transaction abort is only due to false positives in the Bloom Filter based validation.

The plots in Figure 4 show the percentage of aborted transactions when using the BFC scheme with a target *maxAbortRate* of 1%, 5%, 10% as we vary the number of active replicas from 1 to 8 (with 4 threads executing on each replica), highlighting the tight matching between the analytical forecast and the experimental results in presence of heterogeneous load conditions.

Next we consider a more complex micro-benchmark, namely a Red Black tree (again obtained by adapting the implementation originally used for evaluating DSTM2 [22]). In this case we consider a mix of three different transactions: i) a read-only transaction, performing a sequence of searches, ii) a write transaction performing a sequence of searches and insertions, and iii) a write transaction performing a sequence of searches and removals. More in detail, the tree is pre-populated with 50.000 (randomly determined) integer values in the range [-100.000,100.000]. Read-only transactions consist of 200 range queries, each one spanning 5 tree's entries around a randomly chosen integer value. The insertion, resp. removal, write transactions perform first of all 20 range queries, where each query range spans 50 tree's entries, which are aimed at identifying at least a value v which is absent, resp. present, in the tree. If the sequence of range queries fail to identify any such element, the tree is sequentially scanned starting from a randomly chosen value as long as v is found or the maximum value storable by the tree, namely 100.000 is reached (though this case is in practice extremely rare). Finally, if v was found, it is inserted in, resp. removed from, the tree. Note that this logic is aimed at ensuring that the insertion/removal transactions actually perform an update of the tree without, in the case of insertions, introducing duplicate keys. Also, the initial size of the data structure is sufficiently large to yield a light/moderate contention level.

In Figure 5, Figure 6 and Figure 7, we depict the throughput of the system (i.e. number of committed transactions per second) for the three considered workloads when using BFC with the *maxAbortRate* parameter set to 1%. Each plot shows the system throughput for a different combinations of number of replicas and number of server threads in each replica. The number of replicas is varied from 2 to 8 and the number of threads in each replica is varied from 1 to 4. One interesting aspect of these results is that one can observe linear speedups when the number of replicas increases, even in the scenario where 90% of the transactions are write transactions (Figure 5). The latter is, naturally, the scenario with worse performance, given that almost all transactions require the write set to be AB-casted and applied everywhere. Still, even in this case, we can double the throughput of the system when we move from 2 to 6 replicas. As expectable, when the percentage of update transactions is smaller, the system's performance remarkably improve. For instance, for 10% updates (Figure 7) a configuration with 8 replicas and 4 threads achieves a throughput above 8000 tps (against the 1600 tps for the 90% update case). Also, when considering the workload with 10% updates, the configuration with 8 replicas and 4 threads per replica almost triplicates the performance of the same system with only 2 replicas (more precisely, throughput grows from 3000 tps to more than 8000 tps).

In Figure 8 we show the improvement in the execution time of write transactions that is obtained by the use of Bloom Filters for the scenario with 90% write transactions with respect to a standard non-voting certification algorithm requiring to atomically broadcast the whole transaction's readset, e.g. [2]. As below, Bloom Filters are configured to induce less the 1% of aborts due to false positives. As it can be observed in the plot, our optimizations reduce the execution time of write transactions up to

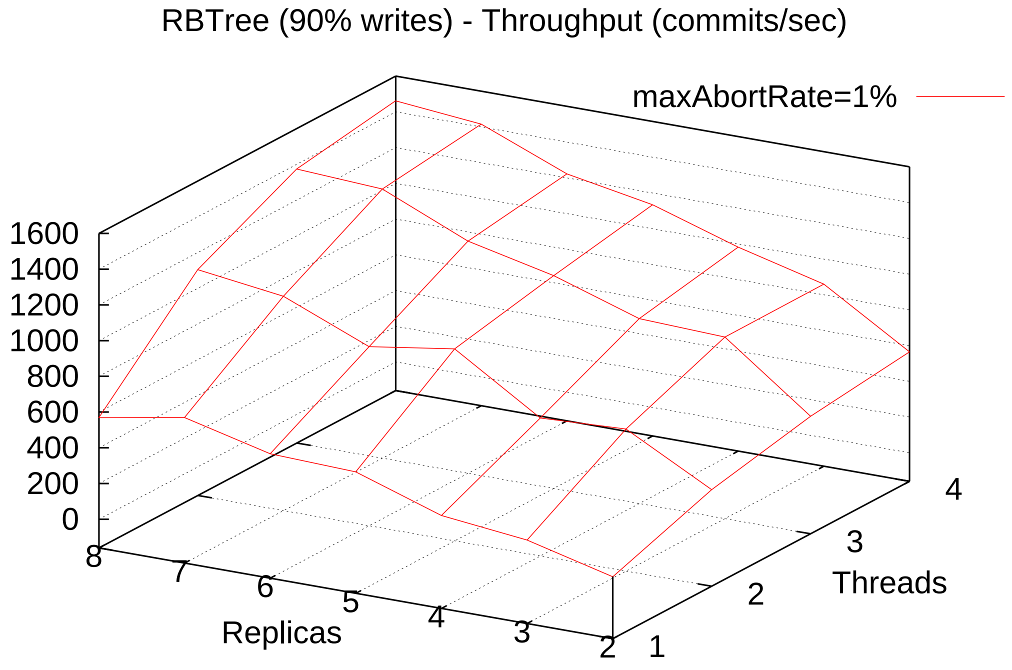


Figure 5: Throughput - Red Black Tree, *maxAbortRate*=1%, 90 % writes

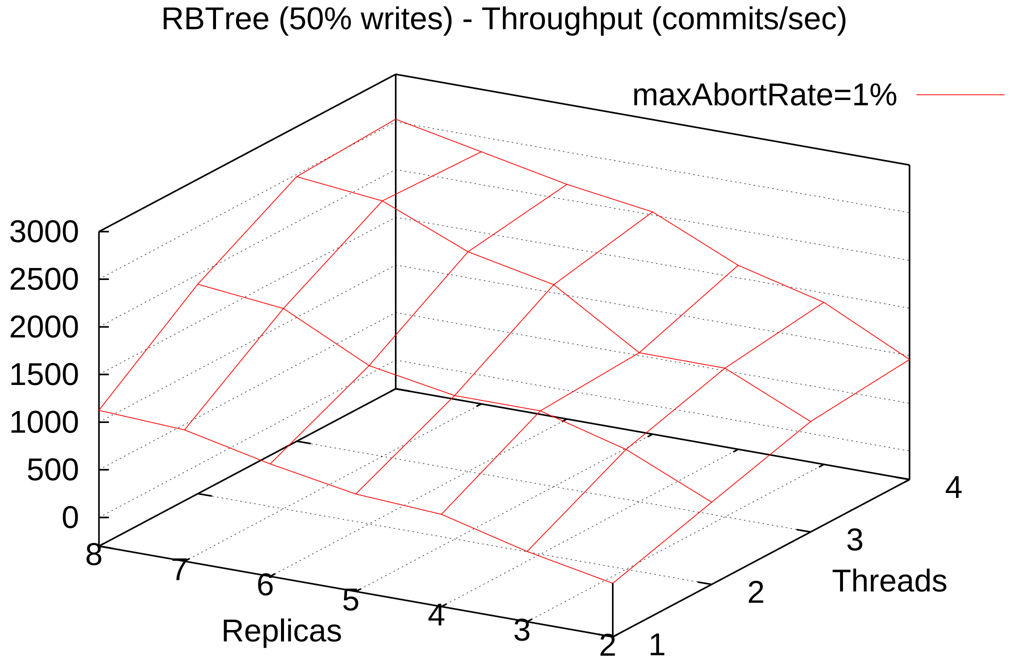


Figure 6: Throughput - Red Black Tree, *maxAbortRate*=1%, 50 % writes

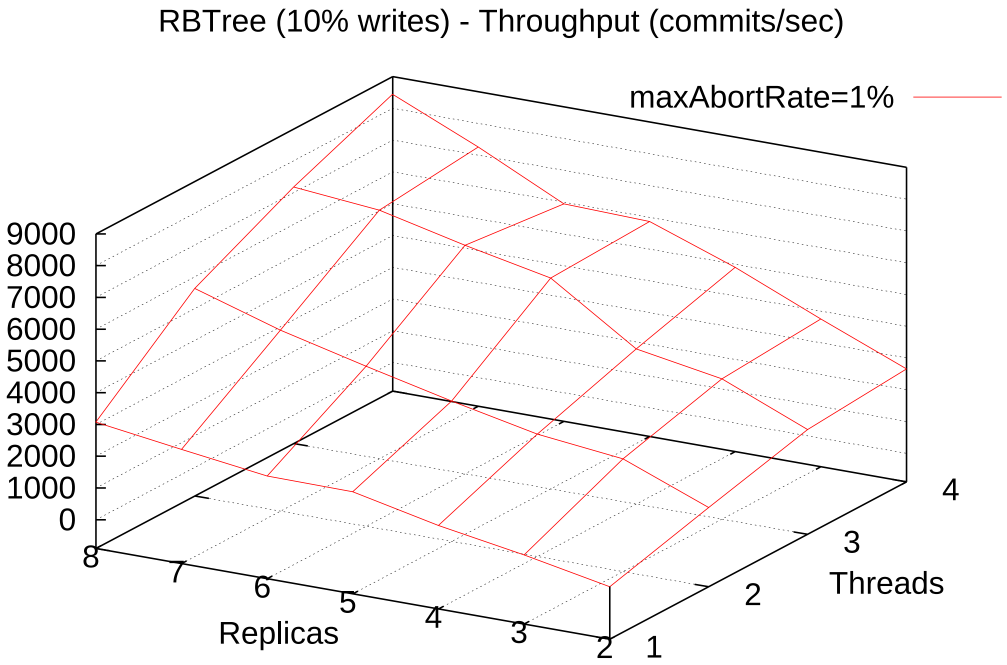


Figure 7: Throughput - Red Black Tree, *maxAbortRate*=1%, 10 % writes

approximately 37% in scenarios with a large number of replicas and threads. This is due to the 10x compression of the messages achieved thanks to the Bloom Filter encoding and to the corresponding reduction of the ABcast latency, which represents a dominant component of the whole transaction's execution time. Note that since the cost of multicast grows with the number of replicas, the reduction also grows proportionally.

We finally show results using the STMBench7 benchmark. This benchmark features a number of operations with different levels of complexity which manipulate an object-graph with a millions of objects heavily interconnected and three types of workload (read dominated, read-write and write dominated). This benchmark can generate very demanding workloads which include, for instance, heavy-weight write transactions performing long traversals of the object graph generating huge readsets. In order to avoid the excessive growth of the size of the messages exchanged when using a standard non-voting certification algorithm (which would lead to the saturation of the network even with a small number of replicas), we found necessary to reduce the size of some of the benchmark's data structures with respect to their default configuration. The exact settings of the benchmark's scale parameters is reported in Table 1 in order to ensure reproducibility of our experiments.

Figure 9 depicts the performance of the system using the "read dominated with long traversals" workload. As before, each plot shows the system throughput for a different combination of number of

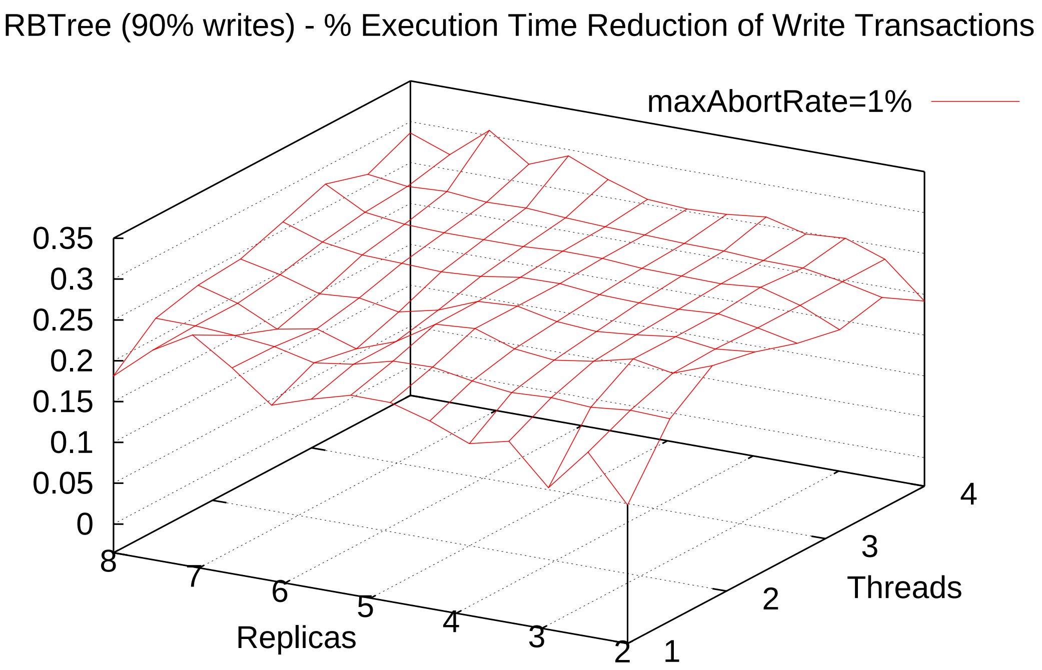


Figure 8: Reduction of the Execution Time of Write Transactions - Red Black Tree, *maxAbortRate*=1%

Parameter	Value
NumAtomicPerComp	100
NumConnPerAtomic	3
DocumentSize	20000
ManualSize	1000000
NumCompPerModule	250
NumAssmPerAssm	3
NumAssmLevels	7
NumCompPerAssm	3
NumModules	1

Table 1: Parameters used to build the initial data structure of STMBench7 benchmark.

replicas (from 2 to 8) and threads per replica (from 1 to 4). The speedup results are consistent with the results obtained with the Red Black tree benchmark. Looking at the throughput numbers in Figure 9(a), we can also observe linear speedups with the increase in the number of replicas. For instance, by moving from 2 to 8 replicas, the system performance increases of a factor 4x independently of the number. Figure 9(b) highlights the performance gains achievable thanks to the usage of Bloom Filter with respect to a classic non voting certification scheme. To this purpose, we report the reduction of execution time for write transactions (namely the only ones to require a distributed certification) which fluctuates in the range from around 20% to around 40%. These gains were achieved, in this case, thanks to the 3x message compression factor permitted by the use of Bloom Filters.

An interesting finding highlighted by our experimental analysis is that, in realistic settings, the BFC scheme achieves significant performance gains even for a negligible (i.e. 1%) additional increase of the transaction's abort rate. This makes the BFC scheme viable, in practice, even in abort-sensitive applications.

In conclusion, the Bloom Filter Certification procedure implemented in D^{2}STM provides fault-tolerance, makes it possible to use additional replicas to improve the throughput of the system (mainly, in the presence of read dominated workloads) and, last but not the least, permits to use (faster) non-voting certification approaches in the presence of workloads with large read sets.

7 Conclusions

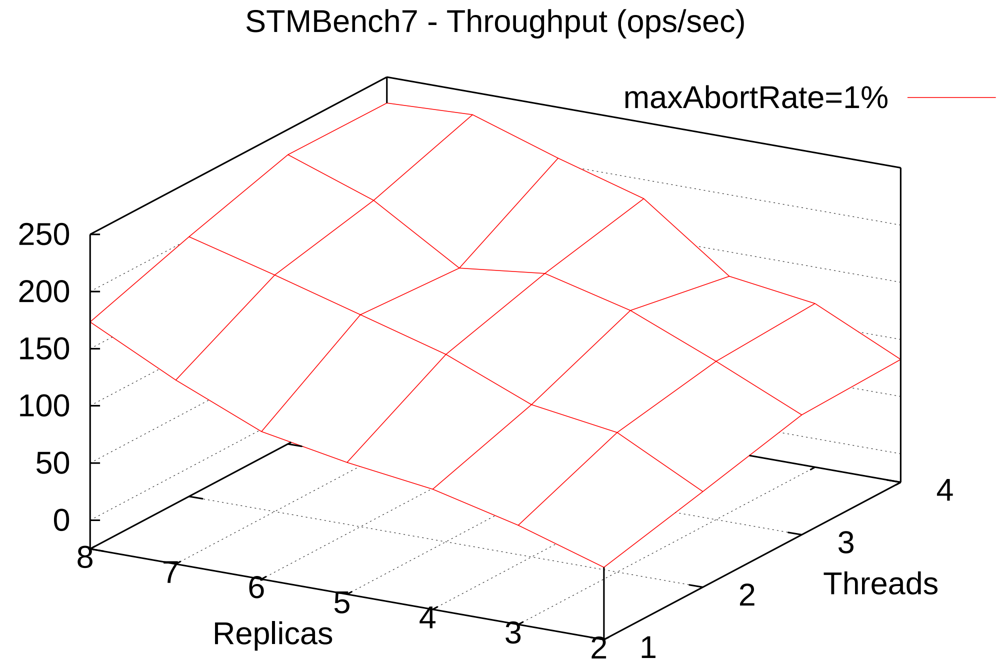
In this work we introduced D^{2}STM, which is, to the best of our knowledge, the first Distributed Software Transactional Memory ensuring both strong consistency and high availability despite the occurrence of (a minority of) replicas' failures.

The replica consistency mechanism at the core of D^{2}STM's, namely the BFC protocol, leverages on a novel Bloom Filter based encoding scheme which allows achieving striking reductions of the overhead associated with the transaction certification phase. Further, thanks to a tight integration with a multi-versioned STM, D^{2}STM can process read-only transactions locally, without incurring in the risk of aborts induced by local or remote conflicts and avoiding any communication overhead.

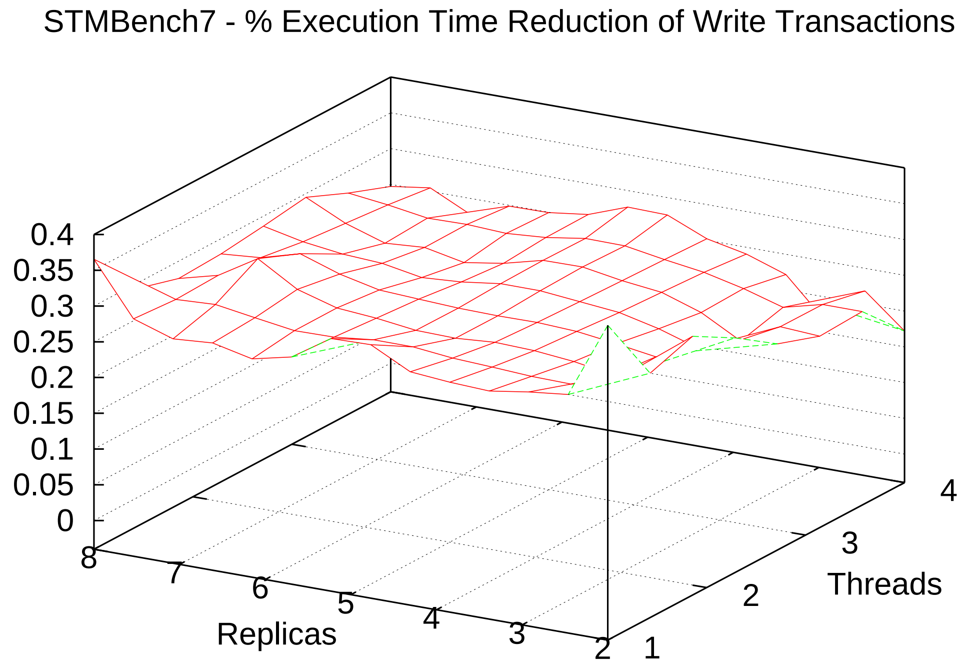
References

[1] K. Adam and M. Michael. Less hashing, same performance: building a better bloom filter. In *ESA'06: Proceedings of the 14th conference on Annual European Symposium*, pages 456–467, London, UK, 2006. Springer-Verlag.

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



(a) Throughput



(b) % Execution Time Reduction

Figure 9: STMBench7, read dominated with long traversals, *maxAbortRate*=1%

[3] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 159–174, New York, NY, USA, 2007. ACM.

[4] Y. Amir, C. Danilov, and J. Stanton. A low latency, loss tolerant architecture and protocol for wide area group communication. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN)*, June 2000.

[5] A. Appleby. Murmurhash 2.0. http://murmurhash.googlepages.com/, 2009.

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

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[8] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the Symposium on Principles and practice of parallel programming*, pages 247–258, New York, NY, USA, 2008. ACM.

[9] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics*, 1(4):485–509, 2003.

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

[11] J. Cachopo and A. Rito-Silva. Combining software transactional memory with a domain modeling language to simplify web application development. In *6th International Conference on Web Engineering*, pages 297–304, July 2006.

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

[13] N. Carvalho, J. Cachopo, L. Rodrigues, and R. S. A. Versioned transactional shared memory for the FenixEDU web application. In *Proc. of the 2nd Workshop on Dependable Distributed Data Management*. ACM, 2008.

[14] N. Carvalho, J. Pereira, and L. Rodrigues. Towards a generic group communication service. In *Proceedings of the 8th International Symposium on Distributed Objects and Applications (DOA)*, Montpellier, France, Oct. 2006.

[15] E. Cecchet, J. Marguerite, and W. Zwaenepole. C-JDBC: flexible database clustering middleware. In *Proc. of the USENIX Annual Technical Conference*, pages 26–26, Berkeley, CA, USA, 2004. USENIX Association.

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

[17] K. Fraser. *Practical lock freedom*. PhD thesis, Cambridge University Computer Laboratory, 2003. Also available as Technical Report UCAM-CL-TR-579.

[18] J. Gray, P. Helland, P. O'Neil, and D. Shasha. The dangers of replication and a solution. In *Proc. of the 25th Conference on the Management of Data (SIGMOD)*, pages 173–182. ACM, 1996.

[19] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 175–184, New York, NY, USA, 2008. ACM.

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

[21] T. Harris and K. Fraser. Language support for lightweight transactions. *SIGPLAN Not.*, 38(11):388–402, 2003.

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

[23] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer, III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101, New York, NY, USA, 2003. ACM.

[24] P. Kankowski. Hash functions: An empirical comparison. http://www.strchr.com/hashfunctions, 2008.

[25] P. Keleher, A. L. Cox, and W. Zwaenepoel. Lazy release consistency for software distributed shared memory. In *ISCA '92: Proceedings of the 19th annual international symposium on Computer architecture*, pages 13–21, New York, NY, USA, 1992. ACM.

[26] B. Kemme and G. Alonso. A suite of database replication protocols based on group communication primitives. In *Proc. of the The 18th International Conference on Distributed Computing Systems*, page 156, Washington, DC, USA, 1998. IEEE Computer Society.

[27] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing transactions over optimistic atomic broadcast protocols. In *Proc. of the 19th IEEE International Conference on Distributed Computing Systems*, page 424, Washington, DC, USA, 1999. IEEE Computer Society.

[28] 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.

[29] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess progranm. *IEEE Trans. Comput.*, 28(9):690–691, 1979.

[30] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. In *Proceedings of the 5th ACM Symposium on Principles of Distributed Computing*, pages 229–239, New York, NY, USA, 1986. ACM.

[31] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the Symposium on Principles and practice of parallel programming*, pages 198–208, New York, NY, USA, 2006. ACM.

[32] M. Martin, C. Blundell, and E. Lewis. Subtleties of transactional memory atomicity semantics. *IEEE Comput. Archit. Lett.*, 5(2):17, 2006.

[33] H. Miranda, A. Pinto, and L. Rodrigues. Appia, a flexible protocol kernel supporting multiple coordinated channels. In *Proc. 21st IEEE International Conference on Distributed Computing Systems*, pages 707–710, Phoenix, Arizona, Apr. 2001. IEEE.

[34] M. Patino-Martínez, R. Jiménez-Peris, B. Kemme, and G. Alonso. Scalable replication in database clusters. In *Proc. of the 14th International Conference on Distributed Computing*, pages 315–329, London, UK, 2000. Springer-Verlag.

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

[36] F. Perez-Sorrosal, M. Pati no-Martinez, R. Jimenez-Peris, and B. Kemme. Consistent and scalable cache replication for multi-tier j2ee applications. In *Middleware '07: Proceedings of the ACM/IFIP/USENIX 2007 International Conference on Middleware*, pages 328–347, New York, NY, USA, 2007. Springer-Verlag New York, Inc.

[37] L. Rodrigues, H. Miranda, R. Almeida, J. Martins, and P. Vicente. The GlobData fault-tolerant replicated distributed object database. In *Proc. of the First EurAsian Conference on Information and Communication Technology*, pages 426–433, London, UK, 2002. Springer-Verlag.

[38] P. Romano, N. Carvalho, and L. Rodrigues. Towards distributed software transactional memory systems. In *Proceedings of the Workshop on Large-Scale Distributed Systems and Middleware (LADIS 2008)*, Watson Research Labs, Yorktown Heights (NY), USA, Sept. 2008. (invited paper).

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