Consistency in Hindsight:
A Fully Decentralized STM Algorithm

Annette Bieniusa
Department of Computer Science
University of Freiburg, Germany
bieniusa@informatik.uni-freiburg.de

Thomas Fuhrmann
Department of Computer Science
Technical University of Munich, Germany
fuhrmann@net.in.tum.de

Abstract—Software transactional memory (STM) algorithms often rely on centralized components to achieve atomicity, isolation and consistency. In a distributed setting, centralized components are undesirable as they impair scalability. This paper presents Decent STM, a fully decentralized object-based STM algorithm. It relies on mostly immutable data structures, which are well-suited for replication and migration. It is the first decentralized STM implementing snapshot isolation semantics. A novel randomized consensus protocol guarantees consistency of the mutable parts. Transactions may proceed tentatively before consensus has been reached. Object versioning ensures consistency in hindsight. Thus, atomic code sections never block during execution. The evaluation of benchmarks shows that the guaranteed success of reads more than compensates for the higher conflict rate during commit.

Keywords—software transactional memory; distributed computing

I. INTRODUCTION

Developing software for distributed systems has proven to be a highly non-trivial task. The complexity and the lack of composability of lock-based synchronization, as offered by APIs such as OpenMP or MPI, causes well-known software-engineering problems. Software transactional memory (STM) is emerging as an alternative to simplify this task with clear and composable semantics of synchronization primitives.

In this paper, we transfer the STM paradigm to a distributed and fully decentralized setting. We propose Decent STM, a fully decentralized object-based STM algorithm. It entirely avoids locking and centralized components. Shared-memory objects are versioned; hence, delayed communication, e.g. caused by retransmissions in the transport layer, may only affect performance, not consistency.

Transactional read and write accesses create dependencies on the shared data items. The traditional correctness criterion for transactions, 1-copy-serializability, imposes strong restrictions on the ordering of transactions.

Decent STM implements (as default) the slightly weaker snapshot isolation (SI) semantics. Snapshot isolation is a popular isolation level in replicated database systems because it allows longer read-only transactions to safely coexist with short update transactions.

Our algorithm obtains lazily a consistent memory snapshot during a transaction’s execution. It keeps a limited list of committed versions of all shared data. We call this list version history. By choosing a version upon read, a transaction determines on which versions it depends. If the version history was unlimited, a transaction would never have to abort on a read operation, because it could always read a previous version that does not conflict with the data read so far (cf. Section IV-B).

The price for using the version history is a higher rate of failed commits and an increased memory overhead. Limiting the version history reduces both, the likeliness of aborts at commit time caused by intermediate writes and the overhead to store the version history. As we show in Section VI, the concept of the history list pays off, because the effect of avoiding the read conflicts more than compensates for the increased aborts.

The Decent STM algorithm does not require any global synchronization operations. All operations (read, write, and commit) involve only communication between the transaction itself and data versions which the corresponding transaction depends on. Our algorithm is thus well suited for large scale distributed systems.

The remaining problem in such a setting are coincidental commits: Two transactions may try to commit at the same time and thus create a conflict if their write sets overlap. We resolve this conflict with a novel distributed randomized consensus protocol. Hence, commits do not take effect immediately but only after all affected processors have reached consensus.

During that time, transactions may read the tentative commits, but they have to abort if the consensus protocol does not elect the corresponding tentative version.

This paper focuses on the basic ideas only. Accord-
ingly, we describe our algorithm in a plain manner without optimizations. Nevertheless, we designed it with such optimizations in mind, e.g., the possibility to cache, replicate and migrate the data structures so that they match the available memory and processor resources more efficiently.

Outline: We start with discussing related work in Section II. Section III introduces the versioned data management via globally accessible objects. We then describe the Decent STM algorithm in Section IV and the randomized consensus protocol in Section V. Section VI presents benchmark evaluations of Decent STM. Finally, we conclude and give an outline on future work in Section VII.

II. RELATED WORK

Many popular STMs, such as TL2 [1], SkySTM [2], LSA [3], or McRSTM [4], use blocking synchronization barriers; or they rely on centralized components such as version counters. Both make them unsuitable to be transferred to a (truly) distributed setting.

Riegel et al. [5] replace version counters with real-time clocks. Unfortunately, this requires both hardware support and a synchronization protocol to ensure bounded deviation of the timers.

There is also a variety of non-blocking implementations, e.g., [6] or [7], which propose techniques to reduce contention and to avoid serialization due to bottlenecks. These systems provide higher scalability than the blocking systems but introduce a higher runtime overhead. Decent STM is also a non-blocking system. It reduces the entailed runtime overhead by offering non-conflicting reads. Moreover, its underlying data structures are specifically designed for a fully decentralized setting.

ClusterSTM [8] is an STM design for high performance computing on very large-scale commodity clusters. In contrast to our system, they provide a low-level API which is supposed to get integrated into some domain specific language for high productivity computer systems, and thus poses a great burden on the programmer. Following their reasoning about the design space, we also try to minimize communication overhead and aim for both appropriate placement of shared memory and execution frames on appropriate nodes in the future.

Manassiev et al. [9] apply STM to a distributed setting. Unlike our system, Distributed Multiversioning uses replicas of the shared memory on each network node in combination with a distributed shared memory consistency protocol. In our system, we rely on one physical copy of each shared object, though caching and redundancy copies may lead to multiple instances. This data is immutable for the major parts, as described in Section III.

Kotselidis et al. [10] have implemented an STM framework for clusters where a master node serves as a global data store. They evaluated several coherence protocols, one of them decentralized (Transactional Coherence and Consistency, TCC). Since all worker threads are involved in this protocol, it leads to substantial overhead. They also show that leases provide a bottleneck when the application entails high contention. Our protocol involves only objects in the write set. Thus non-interfering transactions can commit concurrently.

Reed [11] proposes the use of multi-versioning for handling decentralized data. In line with us, he retrieves a version corresponding to the memory snapshot taken so far when reading a variable. In contrast to our approach, he utilizes synchronized timers to obtain a consistent memory snapshot. Cachopo et al. [12] implement this idea with versioned boxes to store the value history of a variable in combination with a global commit counter. Riegel et al. [13] transfer snapshot isolation semantics from the database domain to STM. In their implementation, a transaction’s read set is taken from a consistent memory snapshot based on a time stamp given a priori. In contrast, Decent STM incorporates updates into a transaction’s memory snapshot if this does not violate the snapshot’s consistency, i.e., Decent STM gives always the latest consistent versions.

Aydonat et al. [14] incorporate multi-versioning for read-only access into an online schedule generation to reduce the number of conflicts. Ramadan et al. [15] show that their dependence-aware transactional memory system accepts all conflict-serializable schedules. We do not need to integrate such a scheduling system into Decent STM as transactions register their writes only at commit time. Similar to their approach, we do allow forwarding of tentative writes when reading transactional objects.

Guerraoui et al. [16] compare different one-, two- and three-phase commit protocols in terms of underlying assumptions, given guarantees and message overhead. They also show the close relation to consensus protocols. Further, they propose a decentralized one-round three-phase commit protocol [17]. Contrary to their investigation, we propose a (potentially multi-round) randomized consensus protocol as we want the flexibility to introduce some bias towards certain transactions.

Distributed systems like Telex [18] introduce the notion of optimistic execution and rollback to high-level software development. These systems often resemble...
or incorporate databases and offer application-specific support whereas we focus on a generic approach to be incorporated into the memory management of virtual machines, for example.

III. GLOBALLY ACCESSIBLE OBJECTS

The Decent STM algorithm is an object-based STM algorithm. We enforce strong atomicity on all objects by distinguishing between (non-shared) thread-local data and globally accessible objects (GAOs). This approach is usually referred to as partitioning.

A GAO is a distributed data structure for shared-memory objects. It consists of a list containing one or more subsequent successfully committed versions of the GAO (version history), and a tentative version tree (cf. Figure 1).

The representation of a GAO version consists of
- a unique GAO version identifier (GVID),
- the GAO identifier (GID) of the GAO it belongs to,
- the actual data items to be stored, i.e. the member fields according to the respective class,
- a reference to the version’s predecessor,
- the transaction identifier (TID) of the transaction that wrote the GAO version, and
- a list of the TIDs of ongoing transactions that have read this GAO version.

It suffices for the GVID to be unique among all GVIDs of the corresponding GAO. In combination with the GID we then obtain globally unique identifiers. The reference to the previous GAO version builds the committed versions history list of the corresponding GAO.

An underlying routing protocol [19] routes requests to a GAO version based on its GVID. For the purpose of this paper, we assume that all communication operations are performed reliably, i.e. all requests will be answered eventually. However, we do not assume a finite upper bound for the transmission delay. Thus, in practice, an underlying protocol can guarantee reliability with the help of retransmissions.

Routing to a GAO means routing to the processor node that currently handles that GAO. For increased reliability and faster read access, multiple nodes might handle copies of the same GAO. In this paper, we assume that objects are not replicated. We believe that replication can be added without any fundamental changes by using a separate protocol.

Furthermore, except for the TID list, the data structure of a GAO version is immutable. Thus it may be replicated, e.g. by caching. After a successful commit or an abort, the TID is removed from all GAO versions in the read set of the respective transaction. As the TID lists are only required for pruning the version history (cf. Section IV-C), using delayed updates for synchronizing the TID lists does not interfere with the STM mechanism. At worst, this could introduce a performance degradation.

To track the dependence relation of GAO versions and check for conflicts, we also require meta data associated with a transaction. A transaction record consists of
- the transaction’s globally unique identifier (TID),
- the read set,
- the write set, and
- the create set of the transaction.

The read, write, and create set are sets of GVIDs of the GAO versions that the transaction has read, written, or instantiated.

Once a transaction enters the commit phase, the data structure of the transaction becomes immutable. It can then also be easily replicated.

IV. DECENTRALIZED STM

A. Thread-local Operations

Transactions operate on local copies of GAO versions and other thread-local data. Reading a GAO creates a local object copy (LOC) that corresponds to a particular version of that GAO. A successful commit then turns each LOC that the transaction has modified into a new version of that GAO.

The following operations constitute the basic work flow on a node that executes a transaction:

Start a transaction: Create a transaction record with a new unique TID and empty read set, write set, and create set.

Instantiate an object: Create a LOC and initialize it to its default value. Note that committing such a LOC will both create the GAO and its initial version.

Read an object: If a LOC for the GAO to be read exists, read the data from the LOC. Otherwise, send a
fetch request to the GAO and wait for the response. Then create the LOC from the received GAO version by cloning it, enter the GVID into the read set, and read the data from the LOC.

Write an object: Write the data in the LOC and enter the corresponding GVID into the write set if not done before. We assume that a GAO has been read or created before it is modified.

End a transaction: Send a commit request to all GAOs that are about to be written (cf. Section IV-D). The transaction record needs to be kept, first for a potential roll-back, later for providing the read and write sets that link the GAO versions. It may be disposed when the corresponding GAO versions are disposed.

A transaction may encounter unresolvable conflicts either when fetching a GAO or upon commit. In both cases, the node that executes the transaction must perform a roll back, i.e. the corresponding entries in the TID lists of the GAO versions read so far are deleted and the local heap and stack frame are restored to the pre-transaction state.

Figure 2 gives the pseudo code for the thread-local operations. The procedures SEND and RECEIVE represent the asynchronous communication operations between a GAO and a transaction. A SEND procedure takes as arguments the receiver of the message, a flag which denotes the type of the message, and possibly further parameters according to the message type. Similarly, a RECEIVE procedure takes the sender of the message, the message type, and further parameters.

In comparison to other STMs our Decent STM algorithm treats the fetching and commit phase substantially differently. We therefore will have a closer look at these operations.

B. Fetching a GAO version

We use a lazy snapshot algorithm to detect and resolve conflicts during transactional reads. Figure 3 motivates this approach by means of a short example: Assume a transaction $T_{90}$ attempts to read GAO version $v_{12}^1$, i.e. GAO $G_1$ in the version that was written by transaction $T_{42}$. Transaction $T_{42}$ has also written $v_{22}^2$, and it has $v_{38}^3$ in its read set. If $T_{90}$ has previously read $v_{36}^2$ or $v_{12}^1$, the attempt to read $v_{12}^1$ must fail because otherwise transaction $T_{90}$ would operate on an inconsistent memory snapshot. These zombie transactions could produce all kind of unexpected effects, and should be avoided.

Our algorithm establishes a partial order $\preceq$ on the versions of the GAOs according to their relative position in the version history list or the tentative version tree. $v_{m}^i \prec v_{j}^l$ means $v_{m}^i$ is a predecessor of $v_{j}^l$.

We will now develop a more formal description of the fetching operation. When a transaction $T_j$ needs to fetch a version from a GAO’s history list, say for GAO $G_i$, it picks one version, say $v_{m}^i$. In general, this should be the latest version, but not necessarily, as we will see.

```plaintext
procedure stmStart
    tid ← generate unique id
    readset ← ⌀
    writset ← ⌀
    createset ← ⌀
end procedure

procedure stmRead(GAO g)
    loc ← LOCALREAD(g)
    if loc = null then
        SEND(g, fetch, readset)
        RECEIVE(g, deliver, gaov)
        loc ← clone gaov
        readset.add(g, gaov)
    end if
    return loc
end procedure

procedure localRead(GAO g)
    loc ← writset.get(g)
    if loc = null then loc ← createset.get(g)
    end if
    if loc = null then loc ← readset.get(g)
    end if
    return loc
end procedure

procedure stmWrite(LOC loc)
    if writset.get(loc.g) = null & createset.get(loc.g) = null then
        writset.put(loc.g, loc)
    end if
end procedure

procedure STMCreate(Class c)
    loc ← create new loc of class c
    createset.put(loc.g, loc)
return loc
end procedure
```

Figure 2. Thread-local operations of a transaction.

**Figure 3. Illustration of read and write dependencies.**
$T_m$ is then the transaction that committed $v_m^i$. Let $R_m$ be the read set of $T_m$, and $W_m$ its write set. We call the union of the read and write set the check set of version $v_m^i$ and denote it by $C(v_m^i) = R_m \cup W_m$. Note that all versions written by the same transaction share the same check set. We further define for a set $S$ of versions $C(S) = \bigcup_{v \in S} C(v)$ to be their check set.

The transitive closure $C^*(v)$ of a version’s check set reflects all read and write dependencies of $v$. It is formally defined as $C^*(v) = \bigcup_{k=1}^{\infty} C^k(v)$ where $C^1(v) = C(v)$ and $C^n(v) = C(C^{n-1}(v))$.

Transactions must not read GAO versions where some versions date before and some after the elements of $C^*(v)$ because this would violate consistency of the memory snapshot taken. The possibility to read such GAO versions only arises if other transactions committed in the mean time and induced the transitivity of check sets a dependency to the still running transaction.

To avoid this problem, we can use for all GAOs $G$ corresponding to versions in $C^*(v)$ the order relation on GAO versions to check if $C^*(v)$ contains a more recent version of this GAO than the read set. This means we check that $\forall v_h^i \in C^*(v_m^i)$, if there exists a $v_l^i \in R_j$, it holds that $v_h^i \preceq v_l^i$.

If the check holds, the fetch operation was successful and will return $v_m^i$. In this case, the operation will also add $T_j$ to $v_m^i$’s TID list.

Otherwise, the fetch operation failed for $v_m^i$. It retries and picks another version $v'_m = v_m^i \sim v_m^i$ from the GAO’s history list. If no such earlier version exists, the fetch operation fails and the transaction $T_j$ must abort.

Note that it suffices to keep the most recent version of each GAO in the transaction’s check set. Also, the check set does not change after committing the transaction. To obtain better performance they can be cached so that the depth of the recursive calculation will be small.

\section*{C. Limiting the Committed Version History List}

When reading a GAO, going back in history can temporarily avoid a conflict. But it causes the reading transaction to depend on an older version. This can potentially lead to a conflict at commit time as another transaction with an overlapping write set might have already updated variables. Hence, it is not always reasonable to do this.

Furthermore, the longer the version history, the more effort is required to iteratively validate the check sets in the transitive closure. This holds even in the case that the check sets are cached, because the number of involved GAOs could grow. For both reasons, the committed versions history list should be kept short.

The TID list in a GAOV contains all ongoing transactions which read this GAOV. They provide us with the necessary information to decide when a transaction record and the related GAO versions may be discarded. We show its effect with the help of two lemmata (cf. [20] for proofs).

\textbf{Lemma IV.1 (Stability).} Let $v$ be some GAO version. If the TID lists of the predecessors of all $v' \in C^*(v)$ are empty, they will always remain empty.

\textbf{Lemma IV.2 (Disposability).} Let $v$ be some GAO version. If the TID lists of the predecessors of all $v' \in C^*(v)$ are empty, $v$ will not cause a (read) conflict.

The disposability lemma gives a sufficient condition when a transaction record may be discarded, namely when all versions in the transitive closure of the transaction’s check set have only predecessors with empty TID lists. The stability lemma shows that discarding transaction records does not need to be synchronized because this condition is stable.

Finally, we can dispose of all GAO versions that are neither in the check set of any transaction nor the most recent version. This entire process – i.e. checking the predecessors of the check set and disposing the transaction records and GAO versions – is very similar to concurrent mark-and-sweep garbage collection. It can thus be implemented in a similar manner.

Clearly, a long lasting transaction may prevent the application of this rule, because the TID list of some GAO version might not become empty. However, in that case the transaction is likely to fail anyway. It will only succeed if it did not read or write any conflicting GAO version. So, we put such a transaction on probation and clear the history list as described above. If the transaction then performs a conflicting read or commit, it fails immediately.

\section*{D. Committing a transaction}

We now discuss (write) conflicts which might occur in the commit phase of a transaction. If for any GAO in $\{G_i \mid v_i^i \in W_n\}$ the corresponding read version $v_k^i \in R_n$ is not the latest version in the history list, the commit must fail. Otherwise, the new GAO version can be inserted into the tentative version tree.

Checking for intermediate writes and inserting the tentative version needs to be performed atomically for all $v_n^i \in W_n$. To handle coincidental commits, we use a distributed consensus protocol. Similar to the standard two-phase commit protocol, it atomically performs the check and insertion of the tentative versions. It then gradually confirms the versions until they are moved.
from the tentative version tree to the committed versions history list. We describe the protocol in detail in the next section.

V. DISTRIBUTED CONSENSUS PROTOCOL

In general, to solve a consensus problem, a group of participants needs to decide in unison about the value of a shared variable. In the setting of our STM algorithm, we have \( m \) globally accessible objects (GAOs) that have to decide in unison which of the \( n \) pending transactions can commit successfully. Similar to our protocol, many approaches to solve this problem are based on phased commit protocols [16]. These protocols usually reduce and fix the number of participants in each round to one transaction and \( m \) GAOs. Typically, there is a central component which is responsible for managing the committing participants and serializes the commit requests.

In contrast to these approaches, we use a randomized distributed consensus protocol which leaves both the number of transactions and GAOs variable. It avoids centralized managing components. Instead, the background routine that is running a transaction needs to communicate with the routines managing the GAOs. We will for now directly associate these managing routines with the managed counterparts and call them transactions and GAOs, respectively.

Using a randomized consensus protocol also gives us the freedom to prefer some transactions over others. For example, for certain applications, we might favor longer transactions or those using specific resources. In this paper we do not discuss such options, but assume equal chances for all transactions.

A. Example

Before we explain the protocol in detail, we first sketch its design rationale by means of an example. Consider a case where two transactions, \( T_1 \) and \( T_2 \), try to commit on overlapping sets of GAOs, namely on \( G_1, G_2, G_3 \) and \( G_2, G_3 \), respectively. Table I shows the GAOs’ internal state as the protocol progresses in rounds.

Assume that the commit requests are issued at the same time. Here, \( v_j^i \) denotes the GAO version that is committed by transaction \( T_j \) on the GAO \( G_i \). Further, assume for now that due to delays in the network the request for \( v_1^3 \) arrives before the one for \( v_2^2 \) at \( G_2 \), and the one for \( v_1^2 \) arrives after the commit request for \( v_2^3 \) at \( G_3 \). Each GAO answers the first incoming commit request with a positive vote, all following ones with a negative vote. Hence, both transactions receive positive and negative votes in the initial round: Transaction \( T_1 \) received positive votes from \( G_1 \) and \( G_2 \) and one negative vote from \( G_3 \). Transaction \( T_2 \) got one negative vote from \( G_2 \) and one positive vote from \( G_3 \). Hence, the success rate in the first round for \( T_1 \) is \( \frac{2}{3} \), and for \( T_2 \) it is \( \frac{1}{2} \).

The GAOs are then notified with the success rates (cf. \( s_j(0) \) in Table I). They take these values as success probabilities for the respective tentative versions, and normalize them within each GAO (cf. \( p_j(1) \) in Table I). Using these probabilities ensures that transactions that were successful in the previous round, are more likely to win the next round.

Now each GAO randomly chooses one of its tentative versions according to the just calculated probabilities. As GAO \( G_1 \) still has only one tentative version, it has no choice. GAO \( G_2 \) chooses (against the odds) \( v_2^3 \), GAO \( G_3 \) chooses \( v_1^2 \). The transactions are again notified with the results from these second round. In our example, their success rates do not change.

Only in the third round, when all GAOs happen to vote for the same transaction. The probability that all GAOs happen to vote consistently (cf. \( s_j(0) \) in Table I). They take these values as success probabilities for the respective tentative versions, and normalize them within each GAO (cf. \( p_j(1) \) in Table I). Using these probabilities ensures that transactions that were successful in the previous round, are more likely to win the next round.

In this example, we see that the initial vote has in general a strong influence on the final result. We exploit this fact and let subsequent transactions read the tentative version that has the highest probability. As we have explained, this does not violate consistency, because subsequent transactions can only be considered by the consensus protocol when all the versions in its read set have succeeded.

Furthermore, we see that the protocol is guaranteed to eventually reach consensus (i.e. asymptotically almost surely): In each round, there is a non-zero probability that all GAOs happen to vote consistently. In practice, consensus is reached within the first two rounds. In our example, consensus is reached when \( G_2 \) and \( G_3 \) happen to vote for the same transaction. The probability is \( (\frac{1}{2})^2 + (\frac{1}{2})^2 = (\frac{1}{2})^2 \). Thus, in our example, we have a Bernoulli process with \( p = (\frac{1}{2})^2 \), which leads to a Poisson distribution of expected runtimes for the consensus protocol.

B. Algorithm Overview

Let \( T_j \) be a transaction, and \( W_j = \{v_j^i\} \) be its write set, i.e. the versions it wants to publish. \( M_j \) denotes the corresponding set of GAOs that the transaction wants to commit to.

\[ T_1 \text{ received positive votes from } G_1 \text{ and } G_2 \text{ and one negative vote from } G_3. \]
\[ T_2 \text{ got one negative vote from } G_2 \text{ and one positive vote from } G_3. \]
\[ \text{Hence, the success rate in the first round for } T_1 \text{ is } \frac{2}{3}, \text{ and for } T_2 \text{ it is } \frac{1}{2}. \]

The GAOs are then notified with the success rates (cf. \( s_j(0) \) in Table I). They take these values as success probabilities for the respective tentative versions, and normalize them within each GAO (cf. \( p_j(1) \) in Table I). Using these probabilities ensures that transactions that were successful in the previous round, are more likely to win the next round.

Now each GAO randomly chooses one of its tentative versions according to the just calculated probabilities. As GAO \( G_1 \) still has only one tentative version, it has no choice. GAO \( G_2 \) chooses (against the odds) \( v_2^3 \), GAO \( G_3 \) chooses \( v_1^2 \). The transactions are again notified with the results from these second round. In our example, their success rates do not change.

Only in the third round, when all GAOs happen to vote for the same transaction. The probability that all GAOs happen to vote consistently (cf. \( s_j(0) \) in Table I). They take these values as success probabilities for the respective tentative versions, and normalize them within each GAO (cf. \( p_j(1) \) in Table I). Using these probabilities ensures that transactions that were successful in the previous round, are more likely to win the next round.

In this example, we see that the initial vote has in general a strong influence on the final result. We exploit this fact and let subsequent transactions read the tentative version that has the highest probability. As we have explained, this does not violate consistency, because subsequent transactions can only be considered by the consensus protocol when all the versions in its read set have succeeded.

Furthermore, we see that the protocol is guaranteed to eventually reach consensus (i.e. asymptotically almost surely): In each round, there is a non-zero probability that all GAOs happen to vote consistently. In practice, consensus is reached within the first two rounds. In our example, consensus is reached when \( G_2 \) and \( G_3 \) happen to vote for the same transaction. The probability is \( (\frac{1}{2})^2 + (\frac{1}{2})^2 = (\frac{1}{2})^2 \). Thus, in our example, we have a Bernoulli process with \( p = (\frac{1}{2})^2 \), which leads to a Poisson distribution of expected runtimes for the consensus protocol.

B. Algorithm Overview

Let \( T_j \) be a transaction, and \( W_j = \{v_j^i\} \) be its write set, i.e. the versions it wants to publish. \( M_j \) denotes the corresponding set of GAOs that the transaction wants to commit to.
Once a commit request messages reaches the GAO, it checks whether the requested GAO version is eligible, i.e. no intermediate write to the GAO has taken place since the transaction read it. If a GAO signals a write conflict, the transaction is doomed to fail and will send failed messages to all other involved GAOs. If no write conflict was detected, the transaction is notified with a vote message from each GAO. The first transaction that announces a commit on a GAO receives a positive vote. While the GAO is then waiting for the transaction’s reply, other commit requests that come in are also checked for write conflicts, but in the conflict free case they are answered with a negative vote. (Read requests are answered at any time, i.e. also while the protocol is being executed.)

Let \( m^+_j(t) \) denote the vote that the GAO \( G_i \) is sending to transaction \( T_j \) in round \( t \). For a positive vote, \( m^+_j(t) = + \), and for a negative note, \( m^+_j(t) = - \). Further, let \( m_j = |M_j| \) be the total number of votes transaction \( T_j \) receives, and

\[
m^-_j(t) = \{|m^+_j(t) \mid m^+_j(t) = + \quad \text{for} \quad i \in M_j\}
\]

be the number of positive votes a transaction received in round \( t \). Similarly, \( m^-_j(t) \) denotes the number of negative votes.

When all votes have arrived, the transaction decides on its success, failure or continuation.

A transaction commits successfully if it only receives positive votes, i.e. \( s_j(t) = 1 \). It fails either if it receives only negative votes, i.e. \( s_j(t) = 0 \), or another conflicting transaction committed successfully at one of the GAOs. In this case, the GAO signals a write conflict. When a transaction receives both positive and negative votes, it sends a continue message to the GAOs to indicate its wish of participation in another round. This message contains the transaction’s success rate of the previous round, \( s_j(t) = \frac{m^+_j(t)}{m_j} \).

After all transactions that have tentative versions for a GAO have replied, the GAO randomly chooses a winner (for that round) according to the scaled probability

\[
p_j^i(t + 1) = s_j(t) / \sum_{0 \leq j < n} \frac{m^+_j(t)}{m_j}
\]

for a version \( v^i_j \) in round \( t + 1 \). All other probabilities are scaled accordingly.

As before, the winner is notified with a positive vote, all other tentative versions with a negative vote. A GAO stops its participation in an instance of a consensus protocol either when it gets notified with a success message or all involved transactions have withdrawn their tentative versions.

The algorithm for the involved transactions and GAOs is given in Figures 4 and 5. Figure 5 shows the complete execution loop for a GAO. Whenever it receives a message from a transaction, it dispatches on the message type and executes the respective actions.

If another transaction reads the GAO during this commit phase, it shall be given one of the tentative versions according to their respective current probability. When such a transaction commits while the election is still going on, its version is inserted below the respective tentative version (hence the term tentative version tree). These tentative versions are handled after their parent version has been elected. If however such a parent version is not elected, all depending transactions must abort without being considered for consensus. If a GAO responds for each depth level always with the same

<table>
<thead>
<tr>
<th></th>
<th>( G_1 )</th>
<th>( G_2 )</th>
<th>( G_3 )</th>
<th>GAOs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( v^1_1 )</td>
<td>( v^2_1 )</td>
<td>( v^3_1 )</td>
<td>( v^4_1 )</td>
</tr>
<tr>
<td>Initial votes</td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1st round</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( \frac{4}{7} )</td>
<td>( \frac{4}{7} )</td>
<td>( \frac{4}{7} )</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>-</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>2nd round</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
<td>( \frac{2}{3} )</td>
</tr>
<tr>
<td></td>
<td>1</td>
<td>( \frac{4}{7} )</td>
<td>( \frac{4}{7} )</td>
<td>( \frac{4}{7} )</td>
</tr>
<tr>
<td></td>
<td>+</td>
<td>+</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3rd round</td>
<td>success</td>
<td>success</td>
<td>failed</td>
<td>success</td>
</tr>
</tbody>
</table>

Table I

Probabilities and votes at the GAOs.
procedure STMCommit
  for all (g,loc) ∈ writeset do
    SEND(g, commit, loc)
  end for
status ← open
while status = open do
  COLLECTVOTES
  if status = abort then
    for all (g, loc) ∈ writeset do
      SEND(g, abort, loc)
    end for
    for all (g, gaov) ∈ readset do
      SEND(g, clear, gaov)
    end for
  else if status = success then
    for all (g, loc) ∈ writeset do
      SEND(g, success, loc)
    end for
    for all (g, gaov) ∈ readset do
      SEND(g, clear, gaov)
    end for
  else
    abort transaction
  end if
end while
end procedure

procedure COLLECTVOTES
  pos,neg ← 0
  for all (g,loc) ∈ writeset do
    if RECEIVE(g, posVote) then
      pos++
    else if RECEIVE(g, negVote) then
      neg++
    else if RECEIVE(g, conflict) then
      status ← abort
    end if
  end for
  if pos = writeset.size then
    status ← success
  else if neg = writeset.size then
    status ← abort
  end if
end procedure

Figure 4. Committing a transaction.

tentative version, the tree will have on each depth level exactly one branch with further children.

VI. PERFORMANCE EVALUATION

The key feature of Decent STM is its ability to operate in a fully decentralized setting. To assess this feature and evaluate the algorithm’s performance, we have created a simple reference implementation of the Decent STM algorithm in Java. It simulates a distributed setting with the help of a message based runtime system. Processing nodes handle transactions (application threads) and GAOs (runtime threads). The runtime system automatically creates request messages when an application thread reads a GAO or when it commits its write set. The runtime threads generate the respective reply messages and send them back to the application threads. This guarantees that each processing node handles its messages in a serialized order. But it does of course not establish synchronization across the processing nodes.

We evaluated our implementation with several applications on a Dual-Quad-Core AMD Opteron running Java 1.6. The runtime system, which handles the GAOs, consisted of eight background threads. Each thread handled an equal set of GAOs. The assignment of the

Figure 5. GAO execution loop.
Each of the figures shows the average of 10 runs. Figure 6 presents the results of benchmarks with a red-black tree and an AVL tree. In these benchmarks, the tree itself and each node is represented by a GAO. We fill the tree initially with 100 elements in a setup.

GAOs was random and did not reflect the relations between the GAOs. (The development of appropriate placement and migration algorithms is ongoing work.) Each of the figures shows the average of 10 runs.
phase. (This phase is not represented in the figures.) We perform a total of 8000 operations on the tree, of which

- in Figure 6(a) and 6(c), 10% insert, 10% delete, and 80% look up an element;
- in Figure 6(b) and 6(d), 40% insert, 40% delete, and 20% look up an element.

These elements were randomly chosen integers ranging from 0 to 1024, so that we created sufficient contention to stress our STM algorithm.

We show both, the system time and the time spent in the benchmark itself. For the latter, we differentiate between the time spent in the application threads, the commit phase, and the runtime overhead:

- **Runtime** is the absolute time spent in the GAO management for handling read request, commit request, and voting result messages;
- **Commit** is the absolute time that the transactions spent in the commit protocol;
- **Application** is the absolute time spent in the application, including the overhead for transactional reads and writes at the transaction’s side.

The system time corresponds to the time that elapsed between start and end of the benchmark run. We attribute the difference between the system time and the benchmark time to the operating system overhead, including the IO activities that were caused by writing log files during the benchmark.

The time spent in the application dominates by far the runtime and commit time overhead, because our implementation employs intensive checks, which contribute to the application overhead. But as these checks affect the evaluation equally for all degrees of parallelization, we do not consider them harmful for our evaluation.

Figure 7 presents the results from running the vacation benchmark from the STAMP benchmark suite [21] with the parameters `-n2 -q90 -u98 -r2024 -t4096`. The customers and reservations are represented by GAOs, all other data is thread-local or read-only.

The figures show that Decent STM obtains good scalability when increasing the number of application threads. As expected the performance decreases when we deploy more than eight threads, due to a disproportion of available processing cores and threads. We believe that the scalability will be preserved when using Decent STM in a truly distributed setting with a higher total number of cores.

Furthermore, we explored the influence of the committed versions history list length on the performance of the application. To this end, we analyzed the run times and conflicts for the tree benchmarks in the 80% update case (as described above). Figures 8(a) and 8(b) show the system times for version histories of length 0 (no version history), 1, 2, or max (all versions are used when resolving read requests). The system time differs only slightly between the different strategies.

Figures 9(a) and 9(b) show the corresponding read and write conflicts. When using just one thread, unsurprisingly, no conflicts occur. When increasing the number of threads, the number of conflicts increases significantly. Almost all read conflicts occur when no version history is provided. In most cases, the version history is only used up to a depth of one. All further entries are apparently not needed for responding to read requests.

The RB tree benchmark in Figure 9(a) shows a very high abort rate when compared to the AVL tree benchmark, although the RB tree application is in the presented case almost twice as fast. The high abort rate induced by write conflicts is mainly due to the performance of the RB tree application. A closer look at the data revealed that it suffers from multiple aborts in a row as the time needed for restart and re-calculation is less than the time for resolving write conflicts in this case.
VII. CONCLUSION AND FUTURE WORK

In this paper, we presented Decent STM, a fully decentralized STM algorithm. It is based on immutable versions of globally accessible objects (GAOs). The successfully committed versions of a GAO form the GAO’s committed versions history list. GAO versions from transactions that are trying to commit, are kept in a tentative version tree. A randomized consensus protocol operates on that tentative version tree: It accepts non-conflicting GAO versions by moving them to the committed versions history list; and it discards conflicting GAO versions thereby causing the respective transactions to roll-back.

This concept decouples the process of obtaining consistency from the application threads. Thereby, Decent STM can hide the communication latency in a distributed system. In particular, it allows non-blocking reads in the presence of ongoing commits. The Decent STM algorithm also ensures lock freedom in case of competing transactions. Moreover, mainly immutable data structures allow caching, replication, and migration of shared memory objects in an easy way.

An evaluation of Decent STM with three typical benchmarks has shown that its features do not impede scalability. On contrary, as far as we could demonstrate on our eight core machine, Decent STM scales well. Thus, we conclude that Decent STM is well-suited for distributed environments.

Currently, we integrate Decent STM into a distributed Java Virtual Machine. This implementation shall form the basis for an extensive performance analysis in a real-world setting. It will also provide object placement algorithms and migration mechanisms, which are required for an efficient use of Decent STM in distributed settings.

Furthermore, we are also working to incorporate fault-tolerance into Decent STM. We believe that Decent STM’s versioning, which already provides memory snapshots, makes check-pointing a straightforward enhancement.

ACKNOWLEDGMENTS

This work was partially funded as part of the JCell project by the Federal Government of Germany under grant number 01IH08011. We also thank the reviewers who provided us with helpful comments.

REFERENCES


