On Enhancing Concurrency in Distributed Software Transactional Memory

Bo Zhang
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
alexzbzb@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

Abstract

Distributed software transactional memory (STM) promises to alleviate difficulties with lock-based (distributed) synchronization and object performance bottlenecks in distributed systems. The existing distributed STM model based on globally-consistent contention management policies may abort many transactions that could potentially commit without violating correctness. To reduce unnecessary aborts and increase concurrency, we propose the distributed dependency-aware (DDA) model for distributed STM, which adopts different conflict resolution strategies based on the types of transactions.In the DDA model, read-only transactions never abort by keeping a set of versions for each object. Each transaction only keeps precedence relations based on its local knowledge of precedence relations.The DDA model that, when a transaction reads from or writes to an object based on its local knowledge, the underlying precedence graph remains acyclic. We propose starvation-free multi-version (SF-MV)-permissiveness, which ensures that: 1) read-only transactions never abort; and 2) every transaction eventually commits. The DDA model satisfies SF-MV-permissiveness with high probability. We present a set of algorithms to support the DDA model, prove its correctness and permissiveness, and show that is supports invisible reads and efficiently garbage collects useless object versions. Our experimental results show that the DDA model outperforms existing contention management policies by 30%-40% in average in high contention environments.

1. Introduction

Software transactional memory (STM) is an alternative synchronization model for shared in-memory data objects that promises to alleviate the pitfalls of traditional lock-based synchronization schemes, where are often non-scalable, non-composable, and inherently error-prone. Similar to database transactions, a transaction, in STM context, is an explicitly delimited sequence of steps executed atomically by a single thread. A transaction terminates by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). If a transaction aborts, it is typically retried until it commits.

A distributed STM model supports the STM API in a distributed system consisting of a network of nodes that communicate by message passing links. Supporting STM in distributed systems is motivated by the similar difficulties of lock-based synchronization methods employed by existing distributed control-flow programming models such as remote procedure calls (RPCs). For example, RPC calls, while holding locks, can become remotely blocked on other calls for locks, causing distributed deadlocks. Livelocks and lock convoying similarly occur. In addition, in the RPC model, an object can become a "hot spot," and thus a performance bottleneck. As suggested in [10], in a data-flow distributed STM model, object performance bottlenecks can be reduced by exploiting locality: move the object to nodes. Moreover, if an object is shared by a group of geographically-close clients that are far from the object's home, moving the object to the clients can reduce communication costs.

We consider Herlihy and Sun's data-flow model [10]. In this model, transactions are immobile (running at a single node), but objects move from node to node. Transactional synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access. Each node has a *TM proxy* that provides interfaces to the STM application and to proxies of other nodes. When a node v_A initiates a transaction A that requests a read from or write to object o, its TM proxy first checks whether o is in the local cache; if not, the TM proxy invokes a *distributed cache-coherence protocol* (CC) to locate o in the network. Assume that o is in use by a transaction B initiated by node v_B. When v_B receives request $CC.locate(o)$ from v_A, its TM proxy checks whether o is in use by an active local transaction; if so, the TM proxy invokes a *conflict resolution* module to compare the priorities of transaction A and B. Based on the result of the conflict resolution module, v_B's TM proxy decides whether to abort B immediately, or postpone A's request and let B proceed to commit. Eventually, v_B invokes CC to move o to v_A.

Therefore, a distributed STM is primarily composed of two elements. The first element is the *conflict resolution strategy*. Two transactions *conflict* if they access the same object and one access is a write. Most existing STM implementations adopt a conflict resolution strategy that aborts one transaction whenever a conflict occurs—e.g., a contention management module [8]. The second element is the *distributed cache-coherence protocol*. When a transaction attempts to access an object in the network, the distributed cache-coherence protocol must locate the latest cached copy of the object, and move a read-only or writable copy to the requesting transaction.

Most of the past works on STM in distributed systems [3][10][17] focus on the design of cache-coherence protocols, while assuming a contention-management-based conflict resolution strategy. While easy to implement, such a contention management approach may lead to significant number of unnecessary aborts, especially when high concurrency is preferred—e.g., for read-dominated workloads [2]. On the other hand, none of the past works consider the design of conflict resolution strategies to increase concurrency under a general cache-coherence protocol.

In this paper, we approach this problem by exploring how we can increase concurrency in a distributed STM. Our work is motivated by the past works on enhancing concurrency by establishing precedence relations among transactions in multiprocessor systems [7] [12] [14]. A transaction can commit as long as the correctness criterion is not violated through its established precedence relations with other transactions. Generally, the precedence relations among all transactions form a *global precedence graph*. By computing the precedence graph and ensuring that it is acyclic, an STM can efficiently avoid unnecessary aborts [12].

Our contributions are as follows:

1) We propose the *distributed dependency-aware* (or DDA) STM model, which leverages the advantages of the aforementioned two strategies. DDA model utilizes different conflict resolution strategies targeting different types of transactions: read-only, write-only and update (involving both read and write operations) transactions. We identify the two inherent limitations of establishing precedence relations in distributed STM. First, there is no centralized unit to monitor precedence relations among transactions in distributed systems, which are scattered in the network. Each transaction should first observe the status of the precedence graph before the next operation. Hence, a large amount of communication cost between transactions is unavoidable. In the DDA model, we design a set of algorithms to avoid frequent inter-transaction communications. In the DDA model, read-only transactions never abort by keeping a set of versions for each object. Each transaction only keeps precedence relations based on its local knowledge of precedence relations. Our algorithms guarantee that, when a transaction reads from or writes to an object based on its local knowledge, the underlying precedence graph remains acyclic. On the other hand, we adopt a randomized algorithm to assign priorities to update/write-only transactions. This strategy ensures that an update transaction is efficiently processed when it potentially conflicts with another transaction, and ensures system progress.

2) We prove that the DDA model satisfies some desirable properties. It satisfies the *opacity* correctness criterion [7]. We define *starvation-free multi-versioned (SF-MV)-permissiveness*, which ensures that: 1)read-only transactions never abort; and 2) every transaction eventually commits. The DDA model satisfies SF-MV-permissiveness with high probability. The DDA model uses a *real-time useless-prefix* (RT-UP)-garbage-collection (GC) mechanism, which enables it to only keep the shortest suffix of versions that might be needed by live read-only transactions. The DDA model also supports *invisible reads*, which is a desirable property for STM.

3) We implement the DDA model in our *HyFlow* distributed STM project [1] and compare it with existing contention management policies with given cache-coherence protocols. Our results show that in the high contention environments, the DDA model outperforms selected contention management policies by 30%-40% in average; in low contention environments, the DDA model still exhibits an approximately same performance compared with selected contention management policies.

The rest of the paper is organized as follows. We present the preliminaries and system model in Section 2. We formally present the DDA model and propose a set of expected properties in Section 3. We present algorithms of the DDA model and analyze them in Section 4. Experimental results are presented in Section 5. The paper concludes in Section 6.

2. Preliminaries and System Model

Distributed transactions. We consider a set of *distributed transactions* $\mathcal{T} := \{T_1, T_2, \ldots, T_n\}$ sharing up to s objects $\mathcal{O} := \{o_1, o_2, \ldots, o_s\}$ distributed on a network of m nodes $\{v_1, v_2, \ldots, v_m\}$, where nodes communicate by message passing links. For simplicity of the analysis, we consider the objects as *read/write registers* and each node runs a single thread only, i.e., in total there are at most m threads running concurrently. A transaction is invoked by a certain *node* (or *process*) in the distributed system. When there is no ambiguity, the notation of T_i may indicate either a transaction or the node that invokes the transaction. The status of a transaction may be one of the following three: *live*, *aborted*, or *committed*. Retrying an aborted transaction is interpreted as creating a new transaction with a new id. However, when a transaction retries, it preserves the original starting timestamp as its starting time.

An *execution* of a transaction is a sequence of *timed operations*. Generally, there are four action types that may be taken by a single transaction: *write*, *read*, *commit*, and *abort*. A transaction T_i's *type* is defined as: 1) *read-only* if T_i does not contain write operations; 2) *write-only* if T_i does not contain read operations; 3) *update* if T_i consists of both read and write operations. The *duration* of a transaction T_i is denoted by τ_i and refers to the time that T_i executes *locally* until commit without contention, i.e., τ_i

excludes the time needed to locate and move objects in the network.

When a transaction attempts to read from or write to a remote object, the cache-coherence protocol is invoked by the transaction proxy to locate the current cached copy of the object, move a read-only or writable copy to the requesting transaction's local cache. A distributed cache-coherence protocol moves each object via some specific path (e.g., the path in a spanning tree for Ballistic protocol [10]) or the shortest path (e.g., Relay protocol [17]). In this paper we assume an existing underlying distributed cache-coherence protocol CC which moves the object to the requester in a finite time period.

Correctness criterion. We consider the formal model in [9] to reason about concurrent transaction executions. A *transaction history* is the sequence of all operations performed by transactions in a given STM execution, ordered by the time they are issued. Two histories H_1 and H_2 are *equivalent* if they contain the same transaction operations in the same order. A history is *complete* if it does not contain live transactions (the status of any transaction is either committed or aborted). If a history H is not complete, we can obtain $Complete(H)$ by adding a number of abort operations for live transactions in H.

The real-time order of transactions is defined as follows: for any two transactions $\{T_i, T_j\} \in H$, if the first operation of T_j is issued after the last operation of T_i (a commit operation or an abort operation of T_i), then we denote $T_i \prec_H T_j$. Transactions T_i and T_j are *concurrent* if $T_i \not\prec T_j$ and $T_j \not\prec T_i$. A history H is *sequential* if no two transactions in H are concurrent [7]. A sequential history H is *legal* if it respects the sequential specification of each object accessed in H. Intuitively, a sequential history is legal if every read operation returns the value given as an argument to the latest preceding write operation of a committed transaction. For a sequential history H, a transaction $T_i \in H$ is *legal* in H if the largest subsequence H' of H is a legal history, where for every legal transaction $T_k \in H'$, either 1) $k = i$, or 2) T_k is committed and $T_k \prec_H T_i$.

We adopt the *opacity* correctness criterion proposed by Guerraoui and Kapalka [7], which defines the class of histories that are acceptable for any STM. Specifically, a history H is opaque if there exists a sequential history S, such that: 1) S is equivalent to $Complete(H)$; 2) S preserves the real-time order of H; and 3) every transaction $T_i \in S$ is legal in S.

Precedence Graph. In our proposed model, the basic idea to guarantee correctness is to maintain a *precedence graph* of transactions and keep it acyclic, which has been adopted by some recent STM efforts in multiprocessor systems [7], [12], [14]. Generally, transactions form a directed labeled precedence graph, PG, based on the dependencies created during the transaction history. The vertices of PG are transactions. There exists a directed edge $T_i \rightarrow T_j$ in PG due to following cases:

1) Real-time order: $T_i \prec_H T_j$;
2) Read after Write ($W \rightarrow R$): T_j reads the value written by T_i;
3) Write after Read ($R \rightarrow W$): T_j writes to object o, while T_i reads the version overwritten by T_j; or
4) Write after Write ($W \rightarrow W$): T_j writes to object o, which was previously written to by T_i.

3. Distributed dependency-aware model

3.1. Motivation

Most current distributed STM proposals inherit the globally-consistent contention management strategy (or *CM* model in short) from multiprocessor STM, for resolving read/write conflicts on shared objects. In the CM model, a *contention manager* module is responsible for mediating between conflicting accesses to shared objects. For example, in the data-flow distributed STM model proposed by Herlihy and Sun [10], the Greedy contention manager [6] assigns priorities based on transactions' starting timestamps. Each transaction is assigned a unique timestamp when it starts, and it remains unchanged until commit. A running transaction could only be aborted by another transaction with an older timestamp.

Although easy to implement, the CM model sometimes is too conservative in achieving high throughput. Given a specific transactional workload, the CM model may execute all transactions almost entirely sequentially even if a large number of them could run concurrently. For example, consider a workload

where n transactions with duration $1-\delta$ each, share $n+1$ objects in a network, under a cache-coherence protocol that ensures that the duration for any transaction to acquire a remote object is δ. Assume that transaction T_i requests to write to object o_i at time 0 and object o_{i+1} at time $1-\delta-i\cdot\epsilon$. Assume that the Greedy manager is used to resolve conflicts. Let T_i has the i^{th} oldest timestamp. We have $T_1 \prec_G T_2 \prec_G \ldots \prec_G T_n$, where "$T_i \prec_G T_j$" means that T_i's priority is higher than T_j's under the Greedy manager.

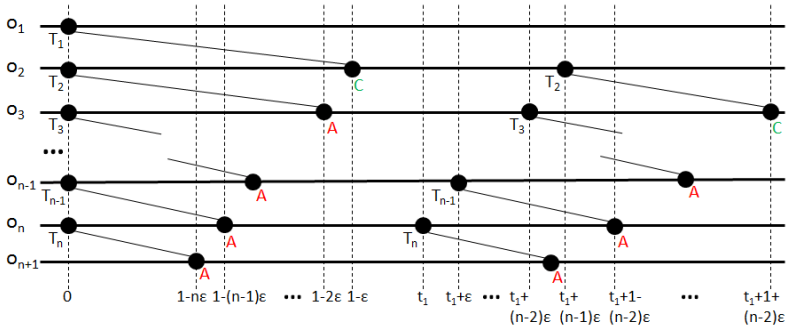

Figure 1. Example 1: the Greedy manager is adopted and transaction T_i has the i^{th} oldest timestamp. T_i can only commit at its i^{th} execution.

We depict this example in Figure 1. We follow the style of [15] to depict transaction histories, and extend it to distributed STM environments. Filled circles correspond to write operations and empty circles represent read operations. Transactions are represented as polylines with circles (write or read operations). Each object o_i's state in the time domain corresponds to a horizontal line from left to right. A commit or an abort operation is indicated by the letter **C** or **A**, respectively. An object moves between transactions which write to it. For example, in Figure 1, object o_2 moves from T_2 to T_1 and T_1 acquires it at $1-\epsilon$. Similarly, o_2 moves from T_1 to T_2 and T_2 acquires it at $t_1+(n-1)\epsilon$.

We assume that initially each object o_i is located at transaction $T_{i \mod n}$. Hence, transaction T_i starts without acquiring o_i remotely and writes to o_i at time 0. At time $1-\delta-i\cdot\epsilon$, T_i requires a write operation to o_i and invokes the cache-coherence protocol to request the object. Hence, T_i acquires o_{i+1} at time $1-i\cdot\epsilon$ for $i\geq 2$, and is aborted by T_{i-1} before $1-(i-1)\epsilon$. Only T_1 commits in its first execution.

After the first execution of each transaction, object o_i is located at T_{i-1} for $i\geq 2$. Transaction T_i ($i\geq 2$) restarts and requests o_i remotely such that T_i acquires o_i ϵ time units before T_{i-1} acquires o_{i-1}. Note that transaction T_{i-1} acquires o_i $(i-1)\epsilon$ time units before its termination. Hence, similarly to its first execution, T_i is aborted by T_{i-1} for $i\geq 3$ and only T_2 commits in its second execution. By repeating this procedure, T_{i-1} aborts T_i at least $i-1$ times and T_i commits at its i^{th} execution. The total time duration to commit the set of n transactions is $\Sigma_{i=1}^{n}(1-i\cdot\epsilon+\delta) = \Omega(n+n\cdot\delta)$.

Obviously, the schedule produced by the Greedy manager is not optimal. By first executing all even transactions and then executing all odd transactions, an optimal schedule finishes in constant time $O(1+\delta)$. Can we find a more efficient conflict resolution strategy to achieve high concurrency? For this specific example, the answer is trivial: all transactions can proceed even when a conflict occurs. Without assigning priorities to transactions, when transaction T_i receives a request from T_{i-1} for object o_i, which is currently in use, it simply sends o_i to T_{i-1} with its initial value (the value before T_i writes to it), denoted by o_i^0. When T_i commits, it sends a request to T_{i-1} to write its value to o_i. In this way, each transaction commits at its first execution. Object o_i $(2\leq i\leq n)$ is first written by T_{i-1} and then by T_i. Let the value written to o_i by the j^{th} write be denoted as o_i^j. Then we know that T_{i-1} writes o_i^1 and T_{i-1} writes o_i^2. As a result, all transactions can be serialized in the order from T_1 to T_n and the time duration to commit all transactions is $O(1+\delta)$, which is optimal.

The example in Figure 1 suggests that the CM model may incur a large amount of unnecessary aborts. On the other hand, instead of aborting a transaction when a conflict occurs, letting conflicting transactions

to proceed in parallel can enhance concurrency efficiently as long as the correctness criterion (i.e., opacity) is not violated. This observation motivates us to propose the *distributed dependency-aware* (DDA) STM model, which differs from past distributed STM models in the way that it resolves read/write conflicts over shared objects.

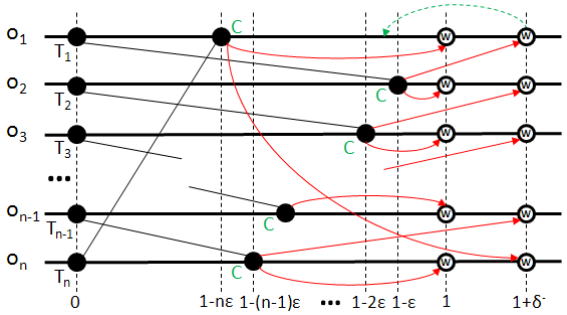

Figure 2. Dining philosophers problem: transactions' write version requests may be interleaved.

In the DDA model, a write-only transaction commits by writing a new version to each object that it requests. Adding a new version to each object in a greedy way in some cases is the simplest and correct solution (e.g., Example 1). However, this method is problematic as it violates opacity under certain workloads. Consider the dining philosophers problem, which is similar to Example 1, except that T_n writes to object o_1 (instead of o_{n+1}) at time $1 - \delta - n \cdot \epsilon$, as shown in Figure 2. In this scenario, transaction T_i needs to write two new versions to o_i and $o_{(i+1) \mod n}$, respectively. Note that T_i only holds $o_{(i+1) \mod n}$ locally (acquired at time $1 - i \cdot \epsilon$) when it commits. Hence, T_i writes a version to $o_{(i+1) \mod n}$ locally at time 1 and writes to o_i remotely at time $1 + \delta^-$, where $\delta^- < \delta$. If objects versions are added in a greedy way, then for object o_i, o_i^1 is written by $T_{(i-1) \mod n}$ and o_i^2 is written to by T_i. Hence, T_i can only be serialized after $T_{(i-1) \mod n}$, since T_i writes after $T_{(i-1) \mod n}$ over object o_i. Obviously, a cycle forms when serializing all transactions, and opacity is violated.

This phenomenon is unique for distributed STM, where objects' versions may be written in an interleaved way. Simply adding versions in a greedy way may violate correctness. To guarantee correctness, the DDA model allows a transaction to insert an object version preceding an older version. How does a transaction decide the proper place to insert an object version? One simplest way is to to assign priorities to transactions based on starting timestamps and insert and the writer's priority, as shown in Figure 2. A circled letter "**W**" represents that an object version is inserted. When transaction T_i tries to insert a version (red lines) to object o_i , it first checks the priorities of the older versions of o_i; if it finds a version o_i^k which is written by a lower-priority transaction (e.g., T_1 finds that o_1 has a version written by T_n at time $1 + \delta^-$), T_i inserts its version preceding o_i^k (the green dotted arc). As a result, the time duration to commit all transactions is $O(1 + \delta)$ and the history is opaque (transactions can be serialized in the priority order). We will study how transactions of different types insert the object versions in detail in Section 4.

3.2. Multi-versioning

Each object o maintains two object version lists: a *pending version list* $o.v_p$ and a *committed version list* $o.v_c$ based on the status of a version's writer. At any given time, the versions of each list is numbered in increasing order, e.g., $o.v_p[1], o.v_p[2], ...$, etc. An object version `Version` includes the data `Version`.*data*, the writer transaction `Version`.*writer*, and `Version`.*sucSet*, a set of detected successors serialized after `Version`. Generally, a transaction is added into `Version`.*sucSet* if: 1) it is an update transaction and reads `Version`; or 2) it writes to the object and `Version`.*writer* precedes it in real-time order. A read operation on object o returns the value of one of o's committed version list. When transaction T_i accesses o to write a value $v(T_i)$, it appends $v(T_i)$ to the tail of $o.v_p$ (note that before this operation, T_i must guarantee that writing to o does not violate correctness), e.g., $v(T_i) = o.v_p[max]$.

When T_i tries to commit, $v(T_i)$ is removed from $o.v_p$ and inserted into $o.v_c$.

Each transaction data structure `TxnDsc` keeps a $readList$ and $writeList$. An entry in a $readList$ points to the version that has been read by the transaction. An entry in a $writeList$ points to the version written by the transaction. `TxnDsc` also keeps a $status$ field which records its status (live, committed or aborted) and a $timestamp$ field which is initialized with its starting timestamp.

3.3. Expected properties

To evaluate the effectiveness of the DDA model, we propose a set of desirable properties for an effective distributed STM supporting multi-versioned objects.

Permissiveness. For multi-versioned STM, the key advantage compared with the CM model is its ability to reduce the number of aborts. The criterion of transaction histories accepted by an STM is captured by the notion of *permissiveness*[5], which restricts the set of aborted transactions by defining such criterion. Informally, an STM satisfies π-permissiveness for a correctness criterion π, if every history that does not violate π is accepted by the STM. However, π-permissiveness is proposed based on an STM model only supporting single-versioned objects and is not sufficient for multi-versioned STM. Keidar and Perelman proposed *online π-permissiveness* [12] which does not allow aborting any transaction if there is a way to continue the run without violating π. Later, Perelman *et. al.* proposed *multi-versioned (MV)-permissiveness* in [13]. In an STM that satisfies MV-permissiveness, read-only transactions never abort and an update transaction is only aborted when it conflicts with another update transaction.

π-permissiveness is argued to be too strong [13]: it aims to avoid all spurious aborts, but is too complicated to achieve and requires keeping a large amount of object versions. On the other hand, we argue that MV-permissiveness may not be strong enough since it does not guarantee that each transaction eventually commits (after a finite number of aborts). For example, an update transaction T may be aborted by infinite times if for every time it restarts, one object in its readset has been overwritten by another update transaction after being read by T and before T commits. In other words, under a certain workload a transaction may starve in an MV-permissive STM. Therefore, our permissive condition captures the starvation-free property in addition to MV-permissiveness.

Definition 1: An STM satisfies starvation-free multi-versioned (SF-MV)-permissiveness if: 1) a transaction aborts only when it is an update transaction that conflicts with another update or write-only transaction; 2) every transaction commits after a finite number of aborts.

Informally, in an STM that satisfies SF-MV-permissiveness, read-only transactions never abort and never cause other transactions' aborts. Furthermore, transactions never starve in SF-MV-permissive STM.

Garbage collection. For multi-versioned STM, old object versions have to be efficiently garbage collected (GC) to save as much space as possible. Perelman *et. al.* [13] argued that no STM can be online space optimal and proposed *useless-prefix (UP)-GC*, a GC mechanism which removes the longest possible prefix of versions for each object at any point of time and keeps the shortest suffix of versions that might be needed by read-only transactions. However, for distributed STM, while transactions may insert its versions before an old version, it may not be safe to always remove the longest possible suffix of versions, since a transaction may not be able to find the proper place to insert its versions. Hence, we define a more practical GC mechanism for SF-MV-permissive STM.

Definition 2: A SF-MV-permissive STM satisfies real-time useless-prefix (RT-UP)-GC if at any point in a transactional history H, an object version o_j^k is kept only if there exists an extension of H with a live transaction T_i, such that o_j^k is the latest version of o_j satisfying $o_j^k.writer \prec_H T_i$.

Read visibility. The STM implementation use *invisible reads* if no shared object is modified when a transaction performs a read-only operation on a shared object, i.e., a read-only transaction leaves no trace the external system about its execution. If a distributed STM supports invisible reads, a traffic between nodes can be greatly reduced for read-dominated workloads and the overall throughput of operations is potentially larger.

4. Algorithms

4.1. Description

Before a transaction performs each read/write operation, it must guarantee that the correctness criterion is not violated. Applying the precedence graph in distributed STM introduces some unique challenges. The key challenge is that, in distributed systems, each transaction has to make decisions based on its local knowledge. A centralized algorithm (e.g., assigning a coordinating node to maintain the precedence graph and make decisions whenever a conflict occurs) involves frequent interactions between different nodes, and is impractical due to the underlying high communication cost. For the same reason, it is also impractical to maintain a global precedence graph on each individual node. Thus, we propose a set of policies to handle read/write operations such that the acyclicity of the underlying precedence graph is not violated and without frequent inter-transaction communications for each individual transaction.

Algorithm 1: Algorithms for read operations

```
1  procedure READ(o) for read-only transaction T_i
2  for Version ← o.v_c[max] to o.v_c[min] do
       // scan the committed version list of o from the latest one
3      if Version.writer ≺_H then
4          return Version.data
5          break
6  procedure Priority Assignment
7  On (re)start of update/write-only transaction T_i:
8  x_{T_i} ← random integer in [1, m];
9  procedure READ(o) for update transaction T_i
10 abortList ← ∅
11 foreach suc ∈ o.v_c[max].sucSet do
12     if suc.status == live then
13         if x_suc < x_{T_i} then
14             ABORT;
15         else
16             add suc to abortList;
17 if T_i.status = live then
18     if o.v_c[max].writer.timestamp ≥ T_i.timestamp then
19         T_i.timestamp ← o.v_c[max].writer.timestamp + ϵ
20     foreach abortWriter ∈ abortList do
21         send abort message to abortWriter
22     add T_i to o.v_c[max].sucSet
23     return o.v_c[max].data // return the latest version
```

Principle 1: In the DDA model, a read-only transaction never aborts.

The pseudo code of read operation for read-only transactions is shown in Algorithm 1. Consider a transaction T_i reading object o. If T_i is a read-only transaction, it reads the latest committed version $o.v_c[j]$ where $o.v_c[j].writer \prec_H T_i$, i.e., the writer of $o.v_c[j]$ precedes T_i in real-time order (lines 2-5). In this way, a read-only transaction is always serialized before other concurrent update/write-only transactions. On the other hand, each object must keep proper object versions to satisfy that each read-only transaction can find the latest committed object version which precedes it in real-time order.

Principle 2: In the DDA model, a transaction aborts if: 1) it is not a read-only transaction; and 2) it conflicts with another transaction and at least one of the conflicting transaction is an update transaction.

Therefore, a transaction aborts in two cases: 1) two update transactions read the same object; and 2) one update transaction and another update/write-only transaction writes to the same object. We discuss them case by case.

Case 1. The pseudo code of read operation for update transactions is shown in Algorithm 1. If T_i is an update transaction, it checks the known successors (which must be serialized after the versions's writer)

of the latest committed version $o.v_c[max]$ and applies a randomized algorithm to make the decision. To assign the priority randomly, each update/write-only transaction T selects an integer $x_T \in [1, m]$ uniformly, independently and randomly when starts or restarts (lines 6-8).

If there exists a live update/write-only transaction $T_j \in o.v_c[max].sucSet$ (line 12), then either one of T_i and T_j can proceed. The transaction with smaller x_T has the higher priority (lines 13-17). After examines all transactions in $o.v_c[max].sucSet$, if T_i is still alive, it sends an abort message to each transaction aborted by T_i (lines 20-21). T_i reads $o.v_c[max]$ and adds itself to $o.v_c[max].sucSet$ (lines 22-23).

An update transaction's timestamp is updated when it reads an object. When an update transaction T_i reads an object version $o.v_c[max]$, it checks the timestamp of its writer ($o.v_c[max].writer.timestamp$). If it has the larger timestamp than T_i, then T_i's timestamp is increased to $o.v_c[max].writer.timestamp+\epsilon$, which is slightly greater than $o.v_c[max].writer.timestamp$.

For example, in the scenario depicted in Figure 3, the sequence of versions read by T_2 is $\{o_1^1, o_2^1\}$. Update transaction T_4 checks the successors of o_2^2 (written by T_5) when reads o_2. Hence, T_4 compares x_{T_4} with x_{T_6}. Assume that $x_{T_4} < x_{T_6}$, T_4 aborts T_6 by sending it an abort message (the dotted line). Now the set of transactions can be serialized in order $T_1T_3T_2T_5T_4T_6$, where T_6 aborts. Similar analysis also applies if $x_{T_6} < x_{T_5}$ and T_5 aborts. Note that after reads o_2, T_4's timestamp is updated from t_1 to $t_2+\epsilon$. Later in this section, we will show that by doing this, the versions written by T_4 (e.g., T_4's version of o_3) can be correctly after the versions written by T_5 (e.g., T_5's version of o_3) by simply comparing their timestamps.

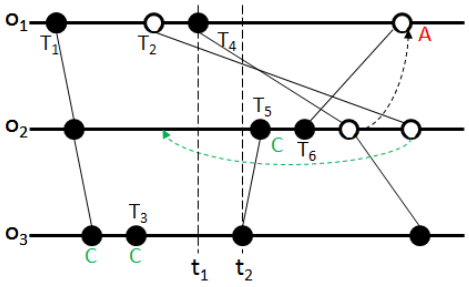

Figure 3. Transactions are serialized in order $T_1T_3T_2T_5T_4T_6$, where T_6 aborts.

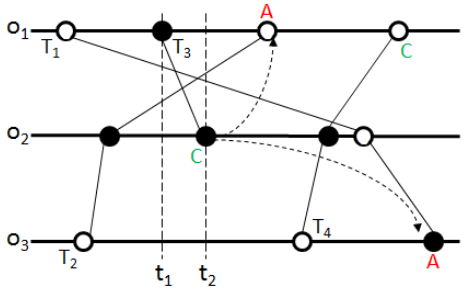

Figure 4. Transactions are serialized in order $T_1T_2T_3T_4$, where T_1 and T_2 abort.

Case 2. We present the pseudo code of write operations in Algorithm 2. When requests to write value $v(T_i)$ to an object o, an update/write-only transaction T_i checks the latest committed version $o.v_c[j]$ such that the writer of $o.v_c[j]$ precedes T_i in real-time order (lines 6-7). For each live transaction $\texttt{suc} \in o.v_c[j].sucSec$, if one transaction in the pair $\langle suc, T_i\rangle$ is an update transaction, then either one of T_i and suc can proceed. The similar random algorithm as the read operations is applied to compare priorities (lines 8-13). If T_i eventually proceeds, it sends an message to each aborted transactions (lines 15-16). T_i writes to o by appending $v(T_i)$ to the end of the pending committed list $o.v_p$ and adds itself to $o.v_c[j].sucSet$ (lines 17-18).

For example, consider the scenario depicted in Figure 4. T_3 conflicts with T_1 and T_2 at t_1 and T_2, respectively. Assume that $x_{T_3} < \{x_{T_1}, x_{T_2}\}$, then T_3 commits and sends an abort message to T_1 and T_2, respectively (the dotted lines). T_4 does not conflict with T_3 since $T_3 \prec_H T_4$. The set of transactions can be serialized in order $T_1T_2T_3T_4$, where T_1 and T_2 abort.

Principle 3: An object's versions are inserted in a greedy way according to writer's timestamps: the version written by a transaction with the older timestamp is always inserted before the version written by a transaction with the newer timestamp.

We present algorithm INSERTVERSION in Algorithm 3. When a transaction tries to commit, it sends a commit message to each object it has requested to write to simultaneously. When receives a commit

Algorithm 2: Algorithms for write operations

```
1  procedure Priority Assignment
2  On (re)start of update/write-only transaction T_i:
3  x_{T_i} ← random integer in [1, m];
4  procedure WRITE(o, v(T_i)) for update/write-only transaction T_i
5  abortList ← ∅;
6  for Version ← o.v_c[max] to o.v_c[min] do
7      if Version.writer ≺_H T_i then
8          foreach suc ∈ Version.sucSet do
9              if suc.status = live & {T_i|suc}.type == update then
10                 if x_suc < x_{T_i} then
11                     ABORT;
12                 else
13                     add suc to abortList;
14         if T_i.status = live then
15             foreach abortSuc ∈ abortList do
16                 send abort message to abortSuc
17             add T_i to Version.sucSet;
18             o.v_p[max + 1] ← v(T_i);
19         break
```

message from T_i, object o removes $v(T_i)$ from pending commit list $o.v_p$ and inserts it to the commit list $o.v_c$ such that the versions in $o.v_c$ is in the order of writers' (updated) timestamps. This strategy guarantees that the versions written by different transactions never interleave at different nodes. By updating the timestamp of each update transaction, a version written by an update transaction is always inserted after the transaction which has written a version the update transaction reads from.

Algorithm 3: Algorithm INSERTVERSION

```
1  procedure INSERTVERSION(o, v(T_i)) when T_i inserts object version v(T_i) to o
2  On receiving a commit message from T_i at object o:
3  remove v(T_i) from o.v_p
4  insert v(T_i) after o.v_c[max]
5  for Version ← o.v_c[max] to o.v_c[min] do
       // scan the committed version list of o from the latest one
6      if Version.writer.timestamp > T_i.timestamp then
7          move v(T_i) before Version
8      else
9          break
```

4.2. Analysis

In this section, we prove that the DDA model satisfies opacity and has following properties: 1) it is SF-MV-permissive with high probability; 2) it supports RT-UP-GC; and 3) it supports invisible reads.

Lemma 1: In the DDA model, a transaction does not generate any cycle in the precedence graph PG before it tries to commit.

Proof: We prove this theorem case by case. Consider an update/write-only transaction T_i. If T_i reads object version o_j^k, then it only adds a $W \rightarrow R$ edge from $o_j^k.writer$ to T_i to PG since o_j^k is the latest committed version of o_j. If T_i writes to object o_j, it first finds the latest committed version $o_j.v_c[k]$ where $o_j.v_c[k].writer \prec_H T_i$, i.e., the writer of $o_j.v_c[k]$ precedes T_i in real-time order. It only adds an $R \rightarrow W$ edge from $T_l \in o_j.v_c[k].sucSet$ to T_i in two cases: 1) T_l is a read-only transaction which reads $o_j.v_c[k]$; 2) T_l is a committed update transaction which reads $o_j.v_c[k]$. Note that the operations of T_i only introduce incoming edges to T_i in PG. Hence, T_i does not generate any outgoing edge before it tries to commit and no cycle forms.

Consider a read-only transaction T_i. From the description of read operations, we know that T_i can always find an object version o_j^k to read for object o_j, where $o_j^k.writer \prec_H T_i$. Hence, for each object o_j^k read by T_i: 1) no new incoming edge to T_i is added to PG; 2) an $R \rightarrow W$ outgoing edge from T_i to T_l is added to PG for each $T_l \in o_j^k.rtSuc$ where T_l writes to o_j.

Suppose a cycle is generated by T_i's operation. Then we can find a cycle $T_{i_1} \rightarrow T_i \rightarrow T_{i_2} \ldots \rightarrow T_{i_1}$ where $T_{i_1} \prec_H T_i$ and $T_i \rightarrow T_{i_2}$ is an $R \rightarrow W$ edge. Then a path exists from T_{i_2} to T_{i_1} before T_i's operation. Note that T_{i_2} is an update transaction. There are two cases based on T_{i_2}'s status. If T_{i_2} is a live transaction, from the first part of the proof we know that no outgoing edge from T_{i_2} exists in PG. If T_{i_2} is a committed transaction, a path forms from T_{i_2} to T_{i_1} if and only if T_{i_1} commits after T_{i_2} commits. In both cases, a contradiction forms. The lemma follows. □

Lemma 1 guarantees the acyclicity of PG from the time a transaction starts to the time it tries to commit. Obviously, the commit of a read-only transaction does not make any change to PG. For transactions with write operations, a new version is inserted in the committed version list for each object in its $writeList$. Such operation brings new edges to PG.

Lemma 2: In the DDA model, the INSERTVERSION operation of a transaction with write operations does not generate any cycle in the precedence graph PG.

Proof: Consider an update transaction T_i which inserts a new version $v(T_i)$ to the committed version list $o_j.v_c$ of object o_j. From Lemma 1, we know that before T_i tries to insert object versions, it does not bring any new outgoing edge to PG. If $v(T_i)$ is inserted to the tail of $o_j.v_c$, then a $W \rightarrow W$ edge from $o_j.v_c[max].writer$ to T_i and a set of $R \rightarrow W$ edges from T_l to T_i for each $T_l \in o_j.v_c[max].readers$ are added to PG. Hence, no new outgoing edge from T_i is added to PG.

If $v(T_i)$ is inserted to the place preceding $o_j.v_c[k]$, then a $W \rightarrow W$ edge from $o_j.v_c[k-1].writer$ to T_i and a set of $R \rightarrow W$ edges from T_l to T_i for each $T_l \in o_j.v_c[k-1].readers$ are added to PG. Additionally, a $W \rightarrow W$ edge from T_i to $o_j.v_c[k].writer$ is added to PG. However, from the description of INSERTVERSION we know that $v(T_i)$ is inserted before $o_j.v_c[k]$ if and only if there preexists an edge from T_i to $o_j.v_c[k]$ in PG. Hence, the INSERTVERSION operation does not introduce new outgoing edge from T_i to PG. The lemma follows. □

We now introduce the following lemma with the help of Lemma 4 from [12]:

Lemma 3: If PG of the execution of a set of transactions is acyclic, then the non-local history H of the execution satisfies opacity.

Then from Lemmas 1, 2 and 3, we have the following theorem.

Theorem 4: In the DDA model, the non-local history H of the execution of any set of transactions satisfies opacity.

Lemma 5: A transaction is aborted at most $O(C \log n)$ times before it commits with probability $1 - \frac{1}{n^2}$, where n is the number of transactions in $\mathcal{T}$ and C is the maximum number transactions concurrently conflicting with a single transaction.

Proof: Let the set of conflicting transaction of transaction T denoted by N_T. T can only be aborted when it chooses a larger x_T than $x_{T'}$, the integer chosen by a conflicting transaction T'. The probability that for transaction T, no transaction $T' \in N_T$ selects the same random number $x_{T'} = x_T$ is

$$\mathbf{Pr}(\nexists T' \in N_T | x_{T'} = x_T) = \prod_{T' \in N_T} (1 - \frac{1}{m}) \geq (1 - \frac{1}{m})^{|N_T|} \geq (1 - \frac{1}{m})^m \geq \frac{1}{e}.$$

Note that $|N_T| \leq C \leq m$. On the other hand, the probability that x_T is at least as small as $x_{T'}$ for any conflicting transaction T' is at least $\frac{1}{(C+1)}$. Thus, the probability that x_T is the smallest among all its neighbors is at least $\frac{1}{e(C+1)}$. We use the following Chernoff bound:

Lemma 6: Let $X_1, X_2, \ldots, X_n$ be independent Poisson trials such that, for $1 \leq i \leq n$, $\mathbf{Pr}(X_i = 1) = p_i$, where $0 \leq p_i \leq 1$. Then, for $X = \sum_{i=1}^n X_i$, $\mu = \mathbf{E}[X] = \sum_{i=1}^n p_i$, and any $\delta \in (0, 1]$, $\mathbf{Pr}(X < (1 - \delta)\mu) < e^{-\delta^2 \mu / 2}$.

By Lemma 6, if we conduct $16e(C+1)\ln n$ trials, each having success probability $\frac{1}{e(C+1)}$, then the probability that the number of successes X is less than $8\ln n$ becomes: $\mathbf{Pr}(X < 8\ln n) < e^{-2\ln n} = \frac{1}{n^2}$. The theorem follows. □

From Lemma 6 we immediately have the following theorem.

Theorem 7: The DDA model satisfies SF-MV-permissiveness with probability $1 - \frac{1}{n^2}$.

Theorem 8: The DDA model satisfies RT-UP-GC.

Proof: The theorem directly follows from the algorithm description. For any object, the earliest version it needs to keep is the latest version that precedes all live read-only transactions in real-time order. All versions earlier than this version can be GCed. The theorem follows. □

Theorem 9: The DDA model supports invisible reads.

Proof: We prove the corollary by contradiction. Suppose the DDA model does not support invisible reads. Then for any history H, we can find a read-only transaction T_i which causes the abort of a read-only transaction or a write-only transaction if T_i is invisible. Note that if T_i is invisible, then the edges added to PG by its read operations are not observed by the STM. From the proof of Lemma 1, we know that T_i only adds outgoing edges from T_i to PG. On the other hand, an update transaction only adds incoming edges to PG. Hence, the only possibility of the cycle formed must be of the form $T_{i_1} \rightarrow T_i \rightarrow \ldots\ldots \rightarrow T_{i_2} \rightarrow T_{i_1}$ where: 1) $T_{i_1} \prec_H T_i$; 2) T_{i_2} is an update transaction; 3) T_{i_1} reads a committed version written by T_{i_2}. Then contradiction forms since T_i and T_{i_2} must be concurrent transactions. The theorem follows. □

5. Experimental Results

To evaluate the practical performance of the DDA model, we implement the DDA model within our *HyFlow* distributed STM project [1]. HyFlow currently focuses on Herlihy and Sun's data flow model [10], and integrates a class of conflict resolution strategies and distributed cache-coherence protocols. In HyFlow, object location is transparent to the application. Objects are located with its IDs using the Directory Manager, which encapsulates a class of cache-coherence protocols. Atomic sections are defined as transactions in which reads and writes to shared objects appear to take effect instantaneously. A transaction maintains its read set and write set, and checks for conflicts on shared objects. If conflicts are detected, the encapsulated conflict resolution strategy decides which transaction to abort in a way that avoids deadlocks, livelock, and ensures system-wide progress.

The experiments were conducted on a 24-node system, with each node running a Java Virtual Machine. Nodes communicate via TCP connections with a link delay of $1ms$. The transactional code length is at least $100ms$. Objects are initially distributed equally on the nodes. Each node run 100 consecutive transactions. We developed and used a distributed version of the Bank application of the STAMP benchmark suite [4], which was rewritten with HyFlow's distributed STM APIs with different cache-coherence protocol/conflict resolution strategy combination.

We use the Home directory protocol [11] and Relay protocol [17] as two underlying cache-coherence protocols. We also implement the Greedy [6] and Karma [16] contention managers for comparison. We control the level of contention by change the percentage of shared objects accessed by a single transaction. Figures 5 and 6 show that in high contention environments (where each transaction accesses 80% of shared objects), the DDA model always outperforms selected contention management policies by 30%-40% in average. When the workload is not read dominated, the DDA model reduces the number of aborts by guaranteeing that a write-only transaction is never aborted by another write-only transaction. The randomized priority assignment also assures the starvation-freedom of a single transaction with high probability. When the workload is read dominated, the number of aborts is significantly reduced since the DDA model never aborts a read-only transaction.

The DDA model does not always outperforms selected contention management policies, as illustrated in Figures 7 and 8. In low contention environments (where each transaction accesses 20% of shared

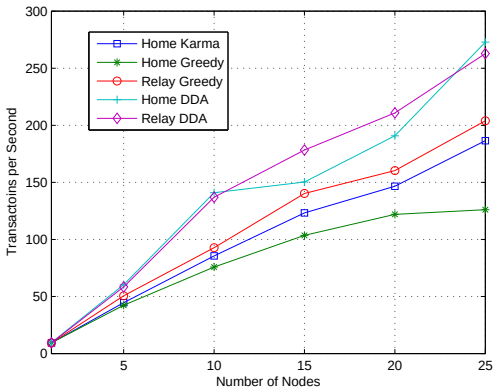

Figure 5. Bank; 50% reads; high contention

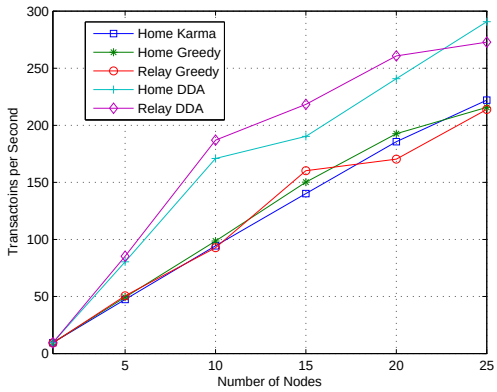

Figure 6. Bank; 90% reads; high contention

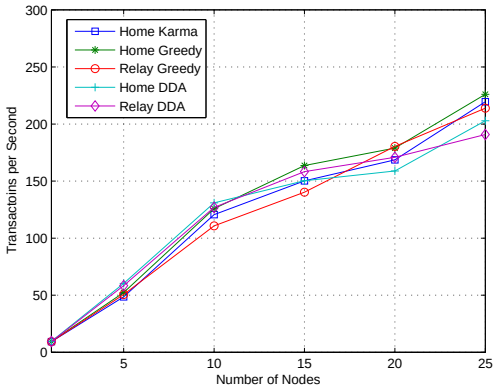

Figure 7. Bank; 50% reads; low contention

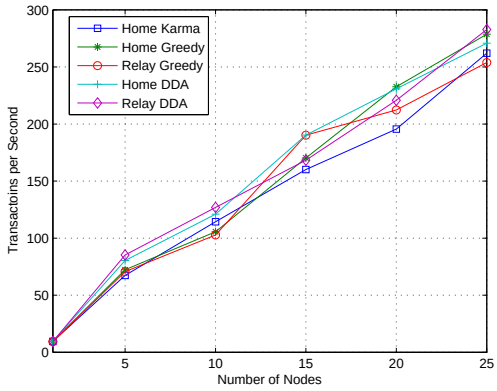

Figure 8. Bank; 90% reads; low contention

objects), the DDA model does not perform as well as in high contention environments. The penalty is that in the DDA model, a transaction has to insert an object versions for each object it requested to write to, which incurs high communication overhead. Yet the DDA model does show an approximately same performance compared with selected contention management policies. This is due to its RT-UP-GC mechanism which discards the useless object versions and reduces the overhead of inserting versions efficiently.

6. Conclusion

This paper takes a step towards enhancing concurrency in distributed STM. We have shown the tradeoff of directly adopting past conflict resolution strategies: the CM model is easy to implement and involves low communication cost in resolving conflicts. However, it may introduce a large number of unnecessary aborts. On the other hand, resolving conflicts by completely relying on establishing precedence relations can effectively reduce aborts. However, it requires frequent message exchanges, which may introduce high communication costs in distributed STM. The DDA model, in some sense, plays a role between these two extremes. It allows maximum concurrency for some transactions (i.e., read-only transactions), and uses randomized priority assignment to treat "dangerous" transactions (i.e., update/write-only transactions), which will likely participate in a cycle in the underlying precedence graph. Moreover, the randomized algorithm ensures the starvation-freedom property with high probability.

Our work suggests a new direction for future research, particular for distributed STM: different conflict resolution strategies can be applied based on different types of transactions. Our work shows that there is a tradeoff between the inter-transaction communication cost and the number of aborts, which is unique for distributed STM. We believe that understanding this tradeoff (as well as others already shown in multiprocessor systems) is important in the design of distributed STM.

References

[1] HyFlow TM, http://www.hyflow.org

[2] Attiya, H., Milani, A.: Transactional scheduling for read-dominated workloads. In: OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems. pp. 3–17. Springer-Verlag, Berlin, Heidelberg (2009)

[3] Bocchino, R.L., Adve, V.S., Chamberlain, B.L.: Software transactional memory for large scale clusters. In: PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming. pp. 247–258. ACM, New York, NY, USA (2008)

[4] Cao Minh, C., Chung, J., Kozyrakis, C., Olukotun, K.: STAMP: Stanford transactional applications for multi-processing. In: IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization (September 2008)

[5] Guerraoui, R., Henzinger, T.A., Singh, V.: Permissiveness in transactional memories. In: DISC '08: Proceedings of the 22nd international symposium on Distributed Computing. pp. 305–319. Springer-Verlag, Berlin, Heidelberg (2008)

[6] Guerraoui, R., Herlihy, M., Pochon, B.: Toward a theory of transactional contention managers. In: PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. pp. 258–264. ACM, New York, NY, USA (2005)

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

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

[9] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann (2008)

[10] Herlihy, M., Sun, Y.: Distributed transactional memory for metric-space networks. Distributed Computing 20(3), 195–208 (2007)

[11] Herlihy, M., Warres, M.P.: A tale of two directories: implementing distributed shared objects in java. In: Proceedings of the ACM 1999 conference on Java Grande. pp. 99–108. JAVA '99, ACM, New York, NY, USA (1999), http://doi.acm.org/10.1145/304065.304107

[12] Keidar, I., Perelman, D.: On avoiding spare aborts in transactional memory. In: SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures. pp. 59–68. ACM, New York, NY, USA (2009)

[13] Perelman, D., Fan, R., Keidar, I.: On maintaining multiple versions in stm. In: PODC '10: Proceeding of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing. pp. 16–25. ACM, New York, NY, USA (2010)

[14] Ramadan, H.E., Roy, I., Herlihy, M., Witchel, E.: Committing conflicting transactions in an stm. In: PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming. pp. 163–172. ACM, New York, NY, USA (2009)

[15] Riegel, T., Fetzer, C., Sturzrehm, H., Felber, P.: From causal to z-linearizable transactional memory. In: PODC '07: Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing. pp. 340–341. ACM, New York, NY, USA (2007)

[16] Scherer, III, W.N., Scott, M.L.: Advanced contention management for dynamic software transactional memory. In: PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing. pp. 240–248. ACM, New York, NY, USA (2005)

[17] Zhang, B., Ravindran, B.: Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In: OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems. pp. 48–53. Springer-Verlag, Berlin, Heidelberg (2009)