Reducing Aborts in Distributed Transactional Systems through Dependency Detection

Bo Zhang
Virginia Tech
alexzbzb@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

ABSTRACT

Existing distributed transactional system execution 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, which adopts different conflicting resolution strategies for different transactions. In the DDA model, the concurrency of transactions is enhanced by ensuring that read-only and write-only transactions never abort, through established precedence relations with other transactions. Non-write-only update transactions are handled through a contention management policy. We identify the inherent limitations in establishing precedence relations in distributed transactional systems and propose their solutions. We present a set of algorithms to support the DDA model, then we prove the correctness and permissiveness of the DDA model and show that it supports invisible reads and efficiently garbage collects useless object versions.

Categories and Subject Descriptors

D.1.3 [**Software**]: Concurrent Programming; H.2.4 [**Database Management**]: Transaction processing; C.2.4 [**Computer Systems Organization**]: Distributed Systems

General Terms

Algorithms, Transactions, Theory

Keywords

Synchronization, Distributed Transactional Systems, Dependency Aware

1. INTRODUCTION

The distributed transactional system's (DTS) processing model consists of a network of nodes that communicate by message-passing links and cooperate with the purpose of running transactions on common shared data. Those data are scattered among nodes and, according to the specific adopted model, they can be replicated (i.e., at least one more version of the same object is available in the system) or purely distributed (i.e., one version per object is maintained by the entire system).

A common classification divides DTS protocols according to their transaction execution flow. The first is *control-flow* [19, 18, 24, 4], in which shared objects are permanently stored at predefined nodes and transactions start from a node and then move their execution flow onto other nodes depending on the accessed objects. The second is *data-flow* [9, 13], which conversely allows objects to migrate among nodes, depending on the sequence of transaction commits, while transactions remain immobile and contact other nodes (i.e., object owners) for obtaining objects. In the latter approach, when a transaction successfully commits, the ownership of the updated object is transferred to the node that is managing the transaction commit.

Due to the fixed and pre-defined owner of each object, *control-flow* protocols can rely on a deterministic, consistent hash function [11] which allows transactions to locally compute the node responsible for maintaining the object without involving any inter-node communication. However, it is clear from the deterministic nature of this function that it does not allow changing the object ownership or biasing the initial object location. Without this feature, the distributed concurrency control cannot optimize the object location depending on the workload at hand. Other upper layers can provide this feature, at the cost of paying overheads [10]. Conversely, this problem is inherently solved in the *data-flow* approach, where objects can be moved to nodes that more frequently request and update such objects. Moreover, if an object is shared by a group of topologically-close clients that are far from the object's home, moving the object to the clients can reduce future communication costs. In the rest of the paper, we focus on the data-flow execution model.

The core of the design of a DTS based on data-flow is 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 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 DTS based on data-flow [2, 9, 17, 25] 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 [1]. 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 DTS. Our work is motivated by the past works on enhancing concurrency by establishing precedence relations among transactions in multiprocessor systems [7, 12, 21]. A transaction can commit as long as the correctness criterion is not violated by its established precedence relations with other transactions. Generally, the precedence relations among all transactions form a *global precedence graph*. By maintaining the precedence graph in time and keep it acyclic, the DTS efficiently avoids unnecessary aborts.

We propose the *distributed dependency-aware* (DDA) model[1], which absorbs the advantages of aforementioned two strategies. We identify the two inherent limitations of establishing precedence relations in DTSs. At 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. Here, read-only and write-only transactions never abort by keeping proper versions of accessed objects. Each transaction only records precedence relations based on its local knowledge. Our algorithms guarantee that, when a transaction reads or writes an object based on its local knowledge, the underlying precedence is acyclic. On the other hand, we adopt a contention management policy to handle non-write-only update transactions (i.e., those transaction involving both read and write operations). This strategy ensures that an update transaction is efficiently processed when it potentially conflict with another transaction, and keeps the system making progress.

Second, when a transaction commits, it should insert a new version for each object it writes to. Since objects are distributed, the node executing the transaction may not hold all the objects the transaction requires to access. Hence, a transaction cannot insert those versions as in a centralized system because each node has its independent notion of the time and therefore different objects may observe different committing times for the same transaction. Such phenomenon can cause a transaction make wrong decision in deciding precedence relations, and introduce unnecessary aborts or violate correctness. We design a set of algorithms to efficiently detect and update precedence relations and ensures that even if a wrong detection occurs, the operations of related transactions are adjusted to accommodate such mistake without violating correctness.

[1]A preliminary version of this paper appeared as Brief Announcement in [26]

The DDA model satisfies the following desirable properties:
- It satisfies *opacity*, the correctness criterion defined in [7].
- In order to capture its capability of reducing aborts, we define the *strong multi-versioned (MV)-permissivenes* property, which restricts the set of possible aborted transactions.
- It satisfies *real-time prefix (RtP) garbage collection (GC)*, which enables the model to keep only the shortest suffix of versions that might be needed by live read-only transactions.
- It also supports *invisible reads*, which is highly desired in DTS.

The paper makes the following contributions.

1. We present the DDA model for DTSs. This is the first distributed model, which relaxes the restriction of contention-management-based conflict resolution strategies.
2. We reveal the inherent limitations of establishing precedence relations in DTSs, and propose their solutions.
3. We present algorithms of read/write operations for read-only, write-only and non-write-only update transactions in the DDA model.
4. We prove that the DDA model satisfies: *i)* opacity; *ii)* strong MV-permissiveness; *iii)* RtP GC; *iv)* invisible reads.

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 the solutions of its two inherent limitations in Section 3. We present algorithms of read/write operations and analyze them in Section 4. The paper concludes in Section 5.

2. PRELIMINARIES AND SYSTEM MODEL

2.1 Programming Model

For the sake of generality and following the trend of [14], we adopt the programming model of software transactional memory (STM) [23] and its natural extension to distributed systems (i.e., DTM) [3, 15]. STM allows the programmer to simply mark a set of operations with transactional requirements as an "atomic block." The STM framework transparently ensures the block's transactional properties (i.e., atomicity, isolation, consistency). This offloads the complexity of managing concurrent requests from the programmer to the STM framework.

2.2 Distributed transactions

A *distributed transaction* performs operations on a set of *shared objects* in a distributed system, where nodes communicate by message-passing links. Let $O := \{o_1, o_2, \ldots\}$ denote the set of shared objects. Each transaction has a unique identifier (id) from a set of transactions $\mathcal{T} := \{T_1, T_2, \ldots\}$. 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. A transaction is a *read-only* transaction if all its operations are read. Otherwise it is an *update* transaction.

To understand the elements of the design to support the transactional API in a distributed system, we consider Herlihy and Sun's data-flow model [9]. In this model, transactions are immobile (running at a single node), but objects move from node to node. Synchronization is optimistic: a transaction commits only if no other transaction has executed a conflicting access. In the data-flow model, each node has a proxy that provides interfaces used by the transactional application to proxies of other nodes. When a transaction at node A requests a read/write access to an object o, its proxy first checks whether o is in its local cache; if not, the proxy invokes a *cache-coherence protocol* to fetch the object o in the network. The node B, which holds o, checks whether it is in use by an active local transaction when it receives the request for o from A. If not, the proxy of B sends o to A and invalidates its own copy. If so, the proxy invokes some conflicting resolution strategy to mediate conflicting access requests for o.

When a transaction attempts to read/write 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. Specifically, in this paper, we assume an underlying distributed cache-coherence protocol CC which satisfies the following properties:

1. When the proxy of transaction T_i attempts to locate an object in the network, CC is invoked to carry T_i's read/write request to the transaction which holds the exclusive writable copy of the object in a finite time period.
2. When a transaction T_j makes the decision to send a read-only copy or the writable copy of the object to T_i, CC is invoked. CC must guarantee that the requested copy of the object is moved to the requesting transaction in a finite time period[2].
3. At any given time, CC must guarantee that there exists only one writable copy of each object in the network. In other words, an object can only be written by a single transaction at any given time.

2.3 Correctness criterion

A *transaction history* is the sequence of all events issued and received by transactions in a given execution, ordered by the time they are issued. Hence, a transaction history describes a computation by ordering the sequence of all its events. A history H is *well-formed* if no transaction both commits and aborts, and no transaction takes any step after it commits or aborts. Two histories H_1 and H_2 are *equivalent* if they contain the same transaction events in the same order. Formally, let $H|T_i$ denote the longest subsequence of history H that contains only events issued/received by transaction T_i. Then histories H_1 and H_2 are equivalent if for any transaction $T_i \in T$, $H_1|T_i = H_2|T_i$. A history is *complete* if it does not contain live transactions, i.e., the status of each transaction is either committed or aborted. If a history H is not complete, we can build a well-formed complete history $Complete(H)$ by aborting the live transactions in H. Specifically, we can obtain $Complete(H)$ by adding a number of abort events for live transactions in H. In this paper, we assume that all histories are well-formed.

[2]We assume a partially synchronous distributed system [16] where a message sent from one node eventually is delivered to the destination node.

The real-time order of transactions is defined as follows: for any two transactions $\{T_i, T_j\} \in H$, if the first event of T_j is issued after the last event of T_i (a commit event or an abort event of T_i), then we denote $T_i \prec_H T_j$. In other words, relation $\prec_H$ represents a partial order on transactions in H. Transactions T_i and T_j are *concurrent* if $T_i \nprec T_j$ and $T_j \nprec T_i$, i.e., the transaction events of T_i and T_j are interleaved. 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 that belongs to 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 DTS. Specifically, a history H is opaque if, according to the above definitions, there exists a sequential history S, such that: *1)* S is equivalent to $Complete(H)$; *2)* S preserves the real-time order of H; *3)* every transaction $T_i \in S$ is legal in S.

3. DISTRIBUTED DEPENDENCY AWARE MODEL

3.1 Motivation

In this section we propose the *distributed dependency-aware* (DDA) model. The DDA model differs from Herlihy and Sun's execution model [9] based on the globally-consistent contention management (GCCM) model in the way that it resolves read/write conflicts on shared objects. In the GCCM model, a *contention manager* module is responsible for mediating among conflicting accesses to a shared object. A running transaction could only be aborted by another transaction with a higher priority. Whenever two transactions concurrently need exclusive access to the same shared object, only one of these transactions is allowed to continue, and the other is immediately aborted (or at least suspended).

Consider the scenario in Figures 1 and 2. We follow the style of [22] to depict transaction histories, which is also adopted in [12]. 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 time domain corresponds to a horizontal line from left to right. A commit or an abort event is indicated by letter **C** or **A**, respectively. The initial value of object o_i is denoted by o_i^0, and the value written to o_i by the n^{th} write is denoted by o_i^n.

Figure 1 depicts an execution of m transactions under the GCCM model. Specifically, we assume that the *Greedy* contention manager [6] is employed, which assigns priority to a transaction based on the time it begins. A transaction begins earlier has the higher priority. Hence, for m transactions in Figure 1, we have $T_1 \prec_G T_2 \prec_G \ldots \prec_G T_m$, where "$T_i \prec_G T_j$" represents that T_i's priority is higher than T_j's under the Greedy manager. In a DTS, a transaction's request is forwarded by the underlying cache-coherence pro-

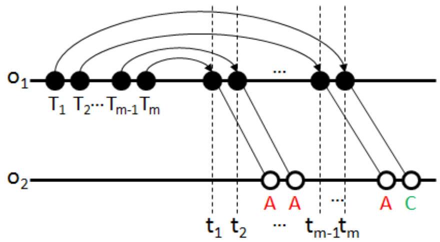

Figure 1: The GCCM model: only T_1 commits under the Greedy contention manager.

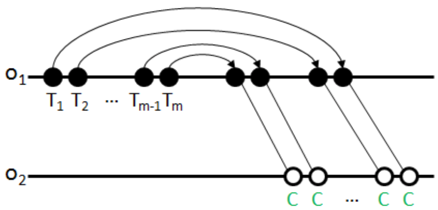

Figure 2: The DDA model: all transactions commit.

tocol. We assume that o_1 first receives T_m's request, although it is the latest transaction that sends the request to o_1 (e.g., it may be the nearest request to o_1). In this scenario, o_1 is first moved to T_1 from its initial place at t_1. In the same way, at time t_k, the DTS knows that T_{m-k+1} requests a write access to o_1. Further, we assume that at time t_k $(2 \leq k \leq m)$, T_{m-k+2} has not committed. Hence, at time t_k, T_{m-k+1} and T_{m-k+2} conflict on o_1 (note that the second operation of each transaction is a read). Since $T_{m-k+1} \prec_G T_{m-k+2}$, T_{m-k+2} is aborted for each $k \in [2, m]$. At last, only T_1 commits.

Figure 2 depicts the execution of the same set of transactions in the DDA model. In this scenario, all transactions can trivially commit in the order that the system receives their requests to o_1. When two transactions conflicts over o_1, the system simply lets them proceed concurrently. Since their second operations do not conflict, the transactions can be serialized in the order that they access o_1. Hence, all transactions can safely commit.

Although somewhat contrived, the examples in Figures 1 and 2 imply the inherent limitation of the GCCM model for DTS. Here, objects are initially scattered in the network, and the locations where transaction are invoked are unpredictable. As the result, it may be impractical to design a globally-consistent policy to assign priorities to transactions which exhibits desirable performance with arbitrarily generated transactions. It may not be a good design choice to simply copy a contention management policy from centralized (e.g., multiprocessor) transactional systems to distributed. Instead of designing a globally-consistent policy to proactively define the priority of a transaction, deciding the priorities of conflicting transactions after the conflict occurs (i.e., as in the DDA model) may leave more space to exploit and increase concurrency of transactions.

3.2 Multi-versioning

The example of Figure 1 and 2 illustrates that the DDA model can avoid unnecessary aborts stemmed from the inherent limitation of the GCCM model. Moreover, past DTS proposals assume that each object only keeps a single version, which may be too conservative and lead to unnecessary aborts. The DDA model allows DTS to manage multiple versions of shared objects.

Each object o maintains two object version lists: a *pending version list*, called $o.v_p$, and a *committed version list*, called $o.v_c$, 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. The data structure of an object version is described in Algorithm 1.

Algorithm 1: Data structure of object version `Version`

Data: data `// actual data written to the object`
id: writer `// transaction ID of the writer`
int: versionNum `// ordered version number`
TxnDsc []: readers `// set of readers`
id []: sucSet `// set of successors detected writing after Version`
id []: preSet `// set of predecessors detected preceding Version`

An object version, called `Version`, includes:
- `Version`.*data*, storing the value;
- `Version`.*writer*, storing the writer transaction's ID;
- `Version`.*readers*, storing a set of readers;
- `Version`.*preSet*, string a set of detected predecessors;
- `Version`.*sucSet*, a set of detected successors (i.e., transaction writing the object after `Version`.

A read operation of 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 commits, $v(T_i)$ is removed from $o.v_p$ and inserted into $o.v_c$. Each transaction keeps two data structures: *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.

3.3 Precedence Graph

In dependence-aware DTS, the basic idea to guarantee correctness is to maintain a *precedence graph* of transactions and keep it acyclic, which has been also adopted by some recent efforts in centralized transactional systems [7, 12, 21]. Generally, transactions form a directed labeled precedence graph, PG, based on the dependencies created during the transaction execution. The vertices of PG are transactions. A directed edge $T_i \rightarrow T_j$ in PG exists in the following cases:
- Real-time order: T_i terminates before T_j starts; or
- Read after Write $(W \rightarrow R)$: T_j reads the value written by T_i; or
- Write after Read $(R \rightarrow W)$: T_j writes to object o, while T_i reads the version overwritten by T_j; or
- Write after Write $(W \rightarrow W)$: T_j writes to object o, which was previously written to by T_i.

3.4 Inherent Limitations

3.4.1 *Distributed Commit Protocol: InsertVersion*

The advantages of the DDA model motivates us to design a framework to support it in DTS. Past similar approaches for centralized transactional systems cannot be directly applied into DTS. In fact, a transaction has to first locate (for read/write) and fetch (only for write) the objects before it performs a read/write operation. Since the DDA model allows multiple conflicting transactions to proceed concurrently, when a transaction attempts to commit all its operations, some objects in its *writeList* may be already moved to other transactions. Intuitively, for each object in its *writeList*, the transaction commits by finding a proper position in the object's version list to insert the new version, without violating correctness. As a result, in DTS, it is unavoidable for a transaction to insert an object version remotely. In this case, directly employing the classic idea from centralized transactional systems by iteratively traversing the written objects to correctly insert all object versions could be extremely expensive in terms of communication costs.

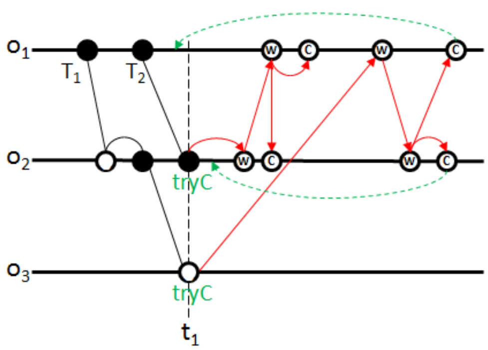

Figure 3: The commit operation which inserts object versions by traversing each object

As an example, consider the scenario depicted in Figure 3. At time t_1, both T_1 and T_2 attempt to commit. Note that at t_1, both o_1 and o_2 are moved to T_2 for its write operations. Hence, T_2's commit operation can be done locally. A circle filled with letter **W** indicates the insertion of a version to the object's version list. In this scenario, T_2 inserts object versions to o_1 and o_2 one after another. Note that T_2 is the first transaction to insert objects versions. Hence, it simply inserts a new version to each object. After the two versions are inserted, T_2 can successfully commit. A circle with letter **C** indicates that the transaction which inserts the new version can commit. Hence, the new version can be safely read by other transactions.

The commit operation performed by T_2 follows the commit protocol in [12]. Since all operations are done locally, no communication cost between transactions is involved. On the other hand, when T_1 conducts similar operations, such cost is induced, as shown in Figure 3. Note that T_1 reads o_2's initial value o_2^0 and T_2 writes to o_2. Hence, T_1 should be serialized before T_2. As a result, the versions written to o_1 and o_2 by T_1 can only be inserted before the versions written by T_2, represented by the dotted arc lines. Since o_1 and o_2 are not located at T_1 when T_1 tries to commit, T_1 can only perform its commit operations remotely. Such operations induce several iterations of communication between T_1 and the object holders until the all object versions can be correctly inserted (commit) or not (abort).

The commit operation illustrated in Figure 3 requires frequent coordinations between object holders. Furthermore,

Algorithm 2: Algorithm INSERTVERSION

1 **procedure** INSERTVERSION$(o, v(T_i))$ when T_i inserts object version $v(T_i)$ to o
2 **remove** $v(T_i)$ **from** $o.v_p$
3 **insert** $v(T_i)$ **after** $o.v_c[max]$
4 **for** `Version` $\leftarrow o.v_c[max]$ **to** $o.v_c[min]$ **do**
    `// scan the committed version list of` o `from the latest one`
5   **if** $T_i \in$ `Version`$.preSet$ **then**
6   **remove** $v(T_i)$ **from** $o.v_p$
7   **move** $v(T_i)$ **before** `Version`
8   **break**
9 **copy** $T_i.preSet$ **to** $v(T_i).preSet$
10 **procedure** UPDATEPRE$(T_i, o.v_c)$ when T_i writes to object version $o.v_c$
11 **for** `Version` $\leftarrow o.v_c[max]$ **to** $o.v_c[min]$ **do**
12   **if** `Version`$.writer \prec_H$ **then**
13   **foreach** *reader* $\in$ `Version`$.readers$ **do**
14   **add** *reader* **to** $T_i.preSet$
15   **add** T_i **to** `Version`$.sucSet$

since a transaction traverses sequentially accessed objects, it may need several iterations of traversing to find a proper position for each object without violating correctness, as suggested in [12]. Apparently, such operation introduces large potential communication cost, which makes it not suited for DTSs. The design of the INSERTVERSION algorithm (Algorithm 2) is motivated by these drawbacks. INSERTVERSION enables each transaction to insert object version in distributed way and avoid inter-transaction communications, as shown in Figure 4.

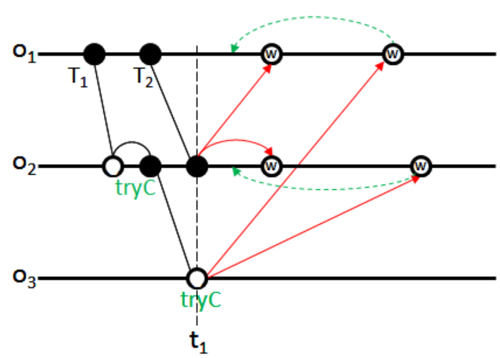

Figure 4: The commit operation implemented by InsertVersion algorithm

At time t_1, T_2 learns that it can only be serialized after T_1 by checking the readers of o_2^0 (lines 10-15). Hence, T_2 can only commit if and only if all its versions are inserted after the versions written by T_1 (if any). Since T_2 inserts its versions before T_1 (by default, a new object version is inserted to the tail of the committed version list, as shown in lines 2-3), the positions of the versions written by T_1 are reserved when T_2 insert its versions. At that time, T_2 just checks each version list to find its reserved positions and inserts its own versions (lines 4-9) there. In this way, no communications between object holders are involved to make each version correctly inserted.

3.4.2 Real-time order detection

The definition of the real-time order inherits the widely-adopted definition in centralized, multiprocessor transactional systems. However, when an update transaction T_i commits in DTSs, it inserts a new version for each object in its *writeList* in a distributed way. As a result, each object in T_i's *writeList* may observe T_i's commit at different time points, thus other transactions may get different information about T_i's commit when accessing different objects. To clarify this, we must define the transaction termination for DTSs.

Definition 1 (Transaction termination). *In DTSs, a transaction T_i terminates if and only if: 1) T_i aborts; or 2) T_i successfully inserts a new version for each object in its $writeList$.*

When a transaction T_i accesses an object o with a version inserted by another transaction T_j, T_i needs to determine its real-time order with T_j. The only information about the time of T_j's commit that T_i can get, is the time when T_j's version for o is inserted. Obviously, T_i may take a wrong decision when it uses this information as T_j's terminating time, since o may not seen the last object version inserted by T_j. Therefore we present the UpdateRt algorithm (Algorithm 3) to let transactions correctly update real-time orders and revise possibly wrong real-time order detections.

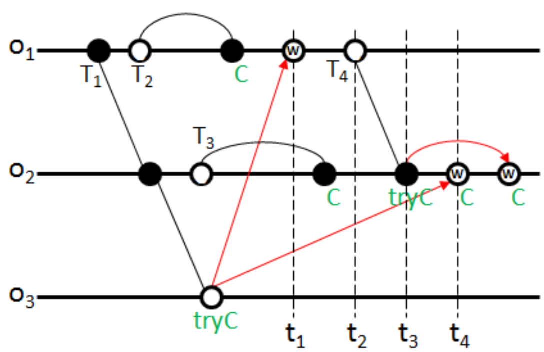

Figure 5: T_4 detects that $T_1 \prec_H T_4$ at t_2. Then at t_3, T_4's commit is postponed after t_4.

Consider the scenario depicted in Figure 5. When T_1 tries to commit, it has to insert new versions to o_1 and o_2, which was moved to T_2 and T_3, respectively. We omit the insert operations of T_2 and T_3 since they are done locally. As a result, when T_1 successfully inserts a new version to o_1 at t_1, o_2 is still waiting for T_1 to insert its new version, which will be done at t_4. When transaction T_4 starts at t_2, it first accesses o_1 to read a value. By comparing its starting time and the insertion time of o_1^2, T_4 wrongly detect that $T_1 \prec_H T_4$, although in fact they are concurrent transactions since T_1 terminates at t_4. When T_4 tries to insert a version to o_2 at t_3, it can only insert a version after the version written by T_1. Furthermore, since T_4 has a real-time order dependency with T_1 , it has to postpone its termination until T_1 commits to comply with the detected real-time order.

The example of Figure 5 illustrates that when a transaction makes a wrong decision about the real-time order, its execution should comply with the real-time order to avoid

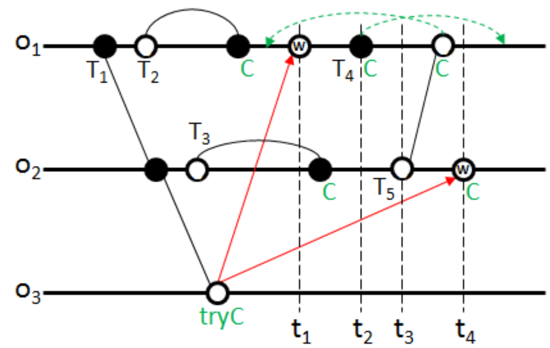

Figure 6: T_5 detects that $T_1 \not\prec_H T_5$ at t_3. Then at t_3, T_5 cannot read the version written by T_4.

unnecessary abort. Moreover, other transactions' execution should also accommodate the established (although wrong) real-time order. Consider, for example, the scenario depicted in Figure 6. When T_4 commits at t_4, it wrongly detects that $T_1 \prec_H T_4$ and inserts the version o_1^3 to o_1. When T_5 starts at t_3, it detects that $T_3 \prec_H T_5$ and reads o_2^1. Thus T_5 establishes a $R \rightarrow W$ order with T_1. When T_5 accesses o_1 to read a value, it detects that $T_1 \prec_H T_4 \prec_H T_5$. Now the contradiction forms since T_5 already knows that T_1 is concurrent with itself. Therefore, T_5 knows that T_4 made a wrong detection. The solution is that T_4 postpones its termination until T_5 commits, thus the real-time order $T_4 \prec_H T_5$ is not held and T_5 can read the value o_1^1.

Algorithm 3: UpdateRt(o) algorithm for T_i to update real-time order when accesses o

foreach `Version` $\in o.v_c$ **do**
  if `Version`$.writer \notin T_i.rtPre$ **then**
    `// for each committed version inserted to` o
    if `Version`$.writeTime < T_i.timeStamp$ & $T_i \nrightarrow$ `Version`$.writer$ **then**
      `// check if` T_i `and Version`$.writer$ `are concurrent`
      add Version$.writer$ to $T_i.rtPre$;
      `// Version`$.writer \prec_H T_i$
foreach `Version` $\in o.v_p$ **do**
  if `Version`$.writer \in T_i.rtPre$ **then**
    `// the detected real-time order Version`$.writer \prec_H T_i$ `is wrong`
    wait until Version$.writerstatus = committed$

4. ALGORITHMS DESCRIPTION AND ANALYSIS

Applying the precedence graph in DTSs 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., assign a coordinator node to maintain the precedence graph and make decisions whenever a conflict occurs), which involves frequent interactions between each individual node and the coordinator node, is impractical due to the underlying communication costs and the limited resources available on a single node. Along this path, it is impractical to maintain a global precedence graph on each individual node. In practice, we propose a set of policies to handle read/write operations such that the acyclicity of the underlying precedence graph is not violated, without frequently inter-transaction communications for each transaction.

4.1 Read

Algorithm 4: Algorithms for read operations

1 **procedure** READ(o) for read-only transaction T_i
2 UPDATERT(o) `// update the real-time order`
3 **for** $Version \leftarrow o.v_c[max]$ **to** $o.v_c[min]$ **do**
  `// scan the committed version list of` o `from the latest one`
4   **if** $Version.writer \prec_H$ **then**
5   **add** T_i **to** Version.$readers$
6   **return** Version.$data$
7   **break**

8 **procedure** READ(o) for update transaction T_i
9 UPDATERT(o)
10 abortList $\leftarrow \emptyset$
11 **foreach** $suc \in o.v_c[max].sucSet \cup o.v_c[max].readers$ **do**
12   **if** $suc.type \neq write\text{-}only$ **then**
13   **if** $suc \notin o.v_c[max].readers$ **then**
14   **if** $suc.timeStamp \leq T_i.timeStamp$ **then**
15   ABORT
16   **break**
17   **else**
18   **add** suc **to** abortList
19   **else**
20   ABORT
21   **break**
22 **if** $T_i.status = live$ **then**
23   **foreach** $abortWriter \in abortList$ **do**
24   send abort message to abortWriter
25   **add** T_i **to** $o.v_c[max].readers$
26   **return** $o.v_c[max].data$ `// return the latest version`

The pseudo code for read operations is shown in Algorithm 4. 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 according to their real-time order relation (lines 3-7).

This way, a read-only transaction guarantees that it can be always serialized before other concurrent transactions. On the other hand, each object must keep proper object versions to satisfy that each read-only transaction can find the latest committed version which precedes it in real-time order.

If T_i is an update transaction, it checks the writing successors (updated by transaction writing the object following Algorithm 2) and readers of the latest committed version $o.v_c[max]$ and applies a contention management policy to make the decision. In the following, we discuss it case by case.

1. If there is no live transaction in $o.v_c[max].rtSuc \cup o.v_c[max].readers$, or for any live transaction $T_j \in o.v_c[max].rtSuc$, T_j just reads $o.v_c[max]$ (line 13), then T_i reads $o.v_c[max]$ (lines 25-26).
2. If there exists a write-only transaction in $o.v_c[max].rtSuc \cup o.v_c[max].readers$ (line 12), then T_i aborts (lines 20-21).
3. If there exists an update transaction $T_j \in o.v_c[max].rtSuc \cup o.v_c[max].readers$ and T_j writes to o (line 13), then only one of T_i and T_j can proceed. We adopt a Greedy contention manager to compare priorities between two transactions based on their timestamps (line 14). The transaction with earlier timestamp has higher priority (lines 14-18). After examines all transactions in $o.v_c[max].rtSuc$, if T_i determines to proceed, it sends an abort message to each transaction which is aborted by T_i (lines 23-24).

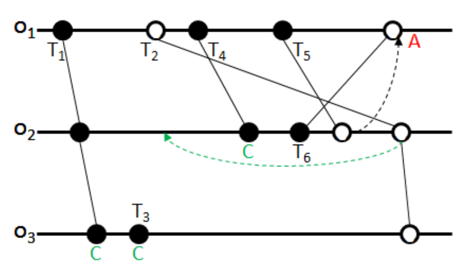

Figure 7: Transactions are serialized in order $T_1T_3T_2T_4T_5T_6$, where T_6 aborts.

In the scenario depicted in Figure 7, the sequence of versions read by T_2 is $\{o_1^1, o_2^1, o_3^2\}$. Note that for object o_2, T_2 does not read o_2^2 written by T_4 since T_4 and T_2 are concurrent. Obviously, if T_2 reads o_2^2, the correctness is violated since T_2 and T_4 cannot be serialized. In this example, T_5 checks the successors of o_2^2 (written by T_4) when reads o_2. Hence, T_5 compares its priority with T_6 and aborts T_6 by sending it an abort message. Now the set of transactions can be serialized in order $T_1T_3T_2T_4T_5T_6$, where T_6 aborts.

4.2 Write

The write operation is managed by the same contention management policy presented above for handling read operation of update transactions. A transaction T_i checks the readers of 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.

1. If there is no live transaction in $o.v_c[j].readers$, or for any live transaction $T_j \in o.v_c[j].readers$, T_j is a read-only transaction, then T_i writes to o by appending $v(T_i)$ to the end of the pending committed list $o.v_p$.
2. If there exists an update transaction $T_j \in o.v_c[j].readers$ which reads $o.v_c[max]$, then only one of T_i and T_j can proceed:
   a. if T_i is a write-only transaction, then T_i has the higher priority;
   b. otherwise, the transaction with earlier timestamp has higher priority.

   After examines all transactions in $o.v_c[j]$, if T_i does not abort, it sends an abort message to each transaction which is aborted by T_i. Then T_i writes to o by appending $v(T_i)$ to the end of the pending committed list $o.v_p$.

An an illustrative example, consider the scenario depicted

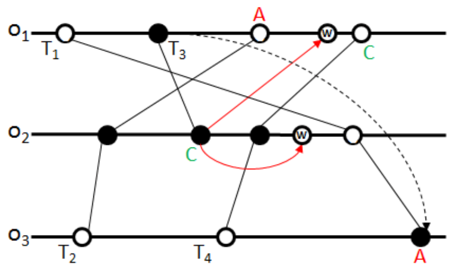

Figure 8: Transactions are serialized in order $T_1T_2T_3T_4$, where T_1 and T_2 abort.

in Figure 8. When T_3 writes to o_1, it aborts T_1 since T_3 is a write-only transaction and T_1 is an update transaction which reads o_1^0. When T_2 reads o_1, it aborts since T_3 is a write-only transaction overwriting o_1^0. On the other hand, T_4 does not abort when writes o_2 since o_2^0 has no readers. Due to the same reason, T_4 does not abort when reads o_1^1 written by T_3. The set of transactions can be serialized in order $T_1T_2T_3T_4$, where T_1 and T_2 abort.

4.3 Correctness

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 transaction T_i. If T_i reads object version o_j^k, then it only adds a $W \to R$ 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 $R \to W$ edge from T_l 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 $R \to W$ 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 $T_{i_1} \to T_i \to T_{i_2} \ldots \to T_{i_1}$ where $T_{i_1} \prec_H T_i$ and $T_i \to T_{i_2}$ is an $R \to W$ 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 update transactions, 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* INSERVVERSION *operation of an update transaction 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 $W \to W$ edge from $o_j.v_c[max].writer$ to T_i and a set of $R \to W$ 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 $W \to W$ edge from $o_j.v_c[k-1].writer$ to T_i and a set of $R \to W$ 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 $W \to W$ 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 relying on 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 Lemma 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.*

4.4 Permissiveness

The key advantage of the DDA model compared with the GCCM model is reducing the number of aborts. Formally, the criterion of transaction histories accepted by a DTS is captured by the notion of *permissiveness* [5], which restricts the set of aborted transactions by defining such criterion. For multi-versioned DTSs, Perelman *et al.* propose *multi-versioned (MV)-permissiveness* in [20]. In a DTS that satisfies MV-permissiveness, read-only transactions never abort and an update transaction is only aborted when it conflicts with another update transaction. Based on MV-permissiveness, we propose the definition of *strong multi-versioned(MV)-permissiveness.*

DEFINITION 2. *A DTS satisfies strong multi-versioned (MV)-permissiveness if a transaction aborts only when it is a non-write-only update transaction that conflicts with another update transaction.*

Informally, in a DTS that satisfies strong MV-permissiveness, read-only and write-only transactions never abort. Furthermore, read-only transactions never cause other transactions' aborts.

THEOREM 5. *The DDA model satisfies strong MV-permissiveness.*

Proof. The proof directly follows the description of read/write operations. In the DDA model, a read-only transaction never conflicts with other transactions. A write-only transaction only conflicts with non-write-only update transactions, and always has higher priority. The theorem follows. □

4.5 Garbage Collection and Read Visibility

We define the following garbage collection (GC) property for strong MV-permissiveness DTSs.

Definition 3. *A strong MV-permissiveness DTS satisfies real-time prefix (RtP) 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 $o_j^k.writer \prec_H T_i$.*

A DTS satisfying RtP GC just keeps the shortest suffix of version that might be needed by live read-only transactions. From the description of read/write operations of the DDA model, we have the following theorem.

Theorem 6. *The DDA model satisfies RtP GC.*

Another desirable property for a DTS is not to update shared memory during read-only transactions, i.e. a read-only transaction leaves no trace the external system about its execution. Such DTSs are said to support *invisible* reads. We can prove the following corollary.

Corollary 7. *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 DTS. 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 corollary follows. □

5. CONCLUSIONS

This paper takes a step towards enhancing concurrency in DTSs. We have shown the tradeoff of directly adopting past conflict resolution strategies: the GCCM model is easy to implement and involves low communication cost in resolving conflicts, while it may introduce large amount of unnecessary aborts; resolving conflicts completely relying on establishing precedence relations can effectively reduce aborts, but it requires frequently message exchanging, which may introduce high communication cost in DTSs. The DDA model, in some sense, plays a role between these two extremes. It allows the maximum concurrency for some transactions (read-only and write-only transactions), and uses contention management policy to treat "dangerous" transactions (non-write-only update transactions), which will likely produce cycles in the underlying precedence graph.

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

6. ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation under grant CNS-1116190.

7. REFERENCES

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

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

[3] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2stm: Dependable distributed software transactional memory. PRDC '09.

[4] N. L. Diegues and P. Romano. Bumper: Sheltering transactions from conflicts. In *IEEE SRDS*, pages 185–194, 2013.

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

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

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

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

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

[10] P. R. J. Paiva, P. Ruivo and L. Rodrigues. Autoplacer: scalable self-tuning data placement in distributed key-value stores. In *ICAC '13*.

[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: distributed caching protocols for

relieving hot spots on the world wide web. In *STOC '97*.

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

[13] J. Kim, R. Palmieri, and B. Ravindran. Enhancing concurrency in distributed transactional memory through commutativity. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, pages 150–161, 2013.

[14] T. Kobus, M. Kokocinski, and P. T. Wojciechowski. Hybrid replication: State-machine-based and deferred-update replication schemes combined. In *ICDCS*, 2013.

[15] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *MULTIPROG*, 2010.

[16] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[17] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 198–208, New York, NY, USA, 2006. ACM.

[18] S. Peluso, P. Romano, and F. Quaglia. SCORe: A scalable one-copy serializable partial replication protocol. In *Middleware*, 2012.

[19] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *ICDCS*, 2012.

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

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

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

[23] N. Shavit and D. Touitou. Software transactional memory. PODC '95.

[24] A. Turcu, S. Peluso, R. Palmieri, and B. Ravindran. Be general and don't give up consistency in geo-replicated transactional systems. In *OPODIS*, 2014.

[25] B. Zhang and B. Ravindran. 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*, pages 48–53, Berlin, Heidelberg, 2009. Springer-Verlag.

[26] B. Zhang and B. Ravindran. Brief announcement: on enhancing concurrency in distributed transactional memory. In *PODC*, pages 73–74, 2010.