On Closed Nesting and Checkpointing in Fault-Tolerant Distributed Transactional Memory

Aditya Dhoke
ECE Dept.
Virginia Tech.
Email: adityad@vt.edu

Binoy Ravindran
ECE Dept.
Virginia Tech.
Email: binoy@vt.edu

Bo Zhang
ECE Dept.
Virginia Tech.
Email: alexzbzb@vt.edu

***Abstract*—We consider the closed nesting and checkpointing model for transactions in fault-tolerant distributed transactional memory (DTM). The closed nested model allows inner-nested transactions to be aborted (in the event of a transactional conflict) without aborting the parent transaction, while checkpointing allows transactions to rollback to a previous execution state, potentially improving concurrency over flat nesting. We consider a quorum-based replicated model for fault-tolerant DTM, and present algorithms to support closed nesting and checkpointing. The algorithms use incremental validation to avoid communication overhead on commit, and ensure 1-copy equivalence. Our experimental studies using a Java DTM implementation of the algorithms on micro and macro benchmarks reveal the conditions when they improve transactional throughput over flat nesting, and also their relative advantages and disadvantages.**

***Keywords*-software transactional memory; distributed systems; replication; closed nesting ; checkpointing;**

I. Introduction

Lock-based synchronization suffers from programmability, scalability, and composability challenges [1]. These difficulties are exacerbated in distributed systems due to the challenges of multi-computer concurrency–e.g., distributed race conditions; distributed versions of deadlocks, livelocks, lock convoying, priority inversion; distributed composability.

Transactional memory (TM) [2] is an alternative synchronization abstraction that promises to alleviate these difficulties. With TM, code that read/write shared memory objects is organized as *memory transactions*, which speculatively execute, while logging changes made to objects. Transactions are monitored for read/write and write/write conflicts, usually by keeping track of their read-sets and write-sets. When two transactions conflict, one of them is aborted, and the other is committed, yielding (the illusion of) atomicity. Aborted transactions are re-started, after rolling-back the changes. Besides a simple programming model, TM provides performance comparable to lock-based/lock-free synchronization, especially for read-dominated workloads, and is composable [3]. TM for multiprocessors was first proposed in hardware (called HTM), later in software (called STM), and subsequently in combination (called HyTM).

Distributed TM (or DTM) [4]–[6] is an alternative to lock-based distributed concurrency control, and can be supported in any distributed execution model, including a) control flow [7], where objects are immobile and transactions invoke object operations through RMIs/RPCs; b) dataflow [8], where transactions are immobile, and objects are migrated to invoking transactions; and c) a hybrid model [9] where transactions or objects are migrated, based on access profiles, object size, or locality. DTM can also be classified based on the system architecture: cache-coherent DTM (cc DTM) [10]–[12], in which a set of nodes communicate by message-passing links over a communication network, and a cluster model (cluster DTM) [13]–[15], in which a group of linked computers works closely together to form a single computer. The most important difference between the two is communication cost. cc DTM assumes a *metric-space* network (i.e., the communication cost between nodes form a metric), whereas cluster DTM differentiates between local cluster memory and remote memory at other clusters. cc DTM uses a cache-coherence protocol (e.g., Ballistic [10], Relay [11]) to locate and move objects in the network, satisfying object consistency properties. Similar to multiprocessor TM, DTM provides a simple distributed programming model (e.g., locks are entirely precluded in the interface), and performance comparable or superior to distributed lock-based concurrency control [12]–[15].

With a single object copy, node/link failures cannot be tolerated. If a node fails, the objects held by the failed node will be simply lost and all following transactions requesting such objects would never commit. Additionally, read concurrency cannot be effectively exploited. Thus, an array of DTM works ([5], [6],

[9], [13], [16], [17]) – all of which are cluster DTM – consider object replication. These works provide fault-tolerance properties by inheriting fault-tolerance protocols from database replication schemes, which rely on broadcast primitives (e.g., atomic broadcast, uniform reliable broadcast) [5], [6], [9], [13], [15], [16]. Broadcasting transactional read/write sets or memory differences in metric-space networks is inherently non-scalable, as messages transmitted grow quadratically with the number of nodes. Thus, directly applying cluster DTM replication solutions to cc DTM may not yield similar performance.

A. Nesting and Checkpointing : Mechanisms for partial abort

Nesting: A transaction is called *nested* when it is enclosed within another transaction. Three types of nesting models have been previously studied [18]: flat, closed and open. They differ based on whether the parent and children transactions can independently abort:

Flat nesting. This is the simplest type of nesting, where the existence of transactions in inner code is simply ignored. All operations are executed in the context of the outermost enclosing transaction. Aborting the inner transaction causes the parent to abort as well (i.e., partial rollback is not possible).

Closed nesting. Here, each transaction attempts to commit individually, but the commit of inner transactions is not visible outside the enclosing transaction. Inner transactions can abort independently of their parent (i.e., partial rollback).

Open nesting. Here, the commit of inner nested transactions are visible to the entire system, though the parent transaction has not yet committed. In case of abort of a parent transaction, the changes committed by inner nested transactions need to be compensated.

Checkpointing model: Transactions in checkpointing model create checkpoints by saving their execution state. Checkpoints provide a way to resume from a consistent transaction state. In case of conflict, transaction can partially rollback to a checkpoint to resolve conflict and resume execution. Checkpointing can be thought of as generalization of closed nesting, where transaction can rollback to any of the checkpoints, whereas, in closed nesting, it is limited one of the enclosing transactions. Let us look at an example of transaction using checkpointing. A transaction has successfully added an element *A* to a hashmap, created checkpoint *Chk* and is attempting to delete element *B*. However, a conflict is encountered while deleting *B*. Instead of aborting the transaction, we can partially rollback to *Chk* and retry deleting *B*.

Flat nested transactions, typically, gain access to objects through read and write requests, perform computations on them, and finally attempt to commit the modifications via a commit request. If the commit request does not succeed, the work done so far is rolled back and retried. This optimistic approach ensures that the transaction accesses the latest copy of objects the next time. However, this can be conservative in many cases, and we could end up doing redundant work.

```
T_flat
m1 = getRemote(m1_Obj);
m2 = getRemote(m2_Obj);
m3 = getRemote(m3_Obj);

intm = add(m1,m2);
result = add(intm,m3);
if commit()
      return result;
else
      retry T_flat;
```

Figure 1: An Example Flat Nested Transaction.

Nesting and checkpointing provide mechanisms to partially abort or rollback a transaction. We show the benefit of partial abort by illustrating an example of closed nesting. Figure 1 shows the code snippet for a transaction, which takes three matrices, m_1, m_2, and m_3, as arguments and returns their sum on successful commit. The transaction adds two matrices at a time. First, the result of the addition of the matrices m_1 and m_2 is stored in an intermediate matrix. Next, the matrix m_3 is added to the intermediate matrix to obtain the final result, after which T_{flat} attempts to commit.

Assume that by the T_{flat} attempts to commit its changes, a conflicting transaction, say T_c, has successfully made write modification to m_3. This will cause T_{flat} to abort, and start again from the beginning (line corresponding to label T_flat in Figure 1). In the next attempt, T_{flat} again reads m_1 and m_2, though they were unchanged, and thus incurring additional remote calls.

Figure 2 shows closed nested transaction, T_{closed}, enclosed inside parent transaction, T_{parent}. Here, T_{parent} adds m_1 and m_2, while T_{closed} adds the intermediate matrix and m_3. Similar to previous example, assume that by the T_{closed} attempts to commit, a conflicting transaction, T_c, has made changes to m_3. As a result, the commit attempt of T_{closed} fails, and it restarts from line corresponding to label T_closed in Figure 2. In the subsequent attempt, T_{closed} will only read m_3. After commit of T_{closed}, T_{parent} will attempt to commit changes to shared memory.

In the closed nesting scenario, we did not repeat the first add operation on m_1 and m_2, thus avoiding extra

```
T_parent
m1 = getRemote(m1_Obj);
m2 = getRemote(m2_Obj);
intm = add(m1,m2);

T_closed
m3 = getRemote(m3_Obj);
result = add(intm,m3);
if commit()
      return result;
else
      retry T_closed;

if commit()
      return result;
else
      retry T_parent;
```

Figure 2: An Example Closed Nested Transaction.

computation and remote calls. In the DTM context, partial abort of transactions can therefore potentially save computation time and communication messages for requesting remote object copies. In the replicated DTM context, the commit request can incur remote messages equal to the number of nodes in the system. Therefore, it becomes important to reduce commit requests (e.g., by minimizing the abort rate) to reduce network traffic.

B. Contributions

In the example in Figure 2, we considered a simple scenario where we knew beforehand that the conflicting transaction had modified the matrix m_3. However, in general, we need to answer the following questions:

(1) What application/workload will benefit from partial abort, as compared to flat nesting?

(2) What is the potential performance improvement or degradation due to partial abort?

(3) Which parameters of a transaction will affect the partial abort performance?

We answer these questions by developing support for closed nesting and checkpointing in replicated cc DTM. We consider Zhang and Ravindran's quorum-based replication model [19] (Section II). In this model, a *quorum system* is used to manage transactional meta data (i.e., read-set, write-set). Transactions communicate with a *read quorum* for obtaining the latest copy of an object for reading and writing, and communicate with a *write quorum* for committing their changes. The intersection property of read and write quorums is used for concurrency control: they ensure consistent state of the replicas and thus, 1-copy equivalence. We support closed nesting and checkpointing by developing protocols called, *QR-CN* (Section III) and , *QR-CHK* (Section IV), respectively. To reduce commit overhead in quorum-based replicated DTM, we develop a protocol for incremental validation called, *Read Quorum Validation* or *Rqv* (Section III-B). We show that Rqv ensures 1-copy equivalence (Section V).

We construct a Java implementation of quorum-based replicated DTM, called QR-DTM, and implement our proposed protocols in QR-DTM. We conduct experimental studies using macro-benchmarks including distributed versions of applications from the STAMP benchmark suite and micro-benchmarks including distributed data structures (Section VI). Our studies reveal that closed nesting improves throughput by as much as 101% over flat nesting in specific cases, with an average improvement of 53% across all benchmarks. To the best of our knowledge, ours is the first work on supporting closed nesting and checkpointing in fault-tolerant DTM (Section VII), and constitutes the paper's contribution.

II. Quorum-Based Replication

Zhang and Ravindran's quorum-based replication protocol [19] (QR for short) provides concurrency control for objects via STM and fault-tolerance by maintaining copies of an object at multiple nodes. Each node is designated a read quorum and a write quorum, where a quorum is a set of nodes having specific properties. A read quorum services read and write requests of objects, while a write quorum is used to commit changes to objects. A transaction executing on a node uses the read and write quorum designated to that node. (From here on, when we say a node's or transaction's quorum we will refer to these designated quorums).

The QR protocol ensures 1-copy equivalence [20], meaning that when a transaction reads an object, it will use the latest copy of the object. This property is maintained by the system, because any write quorum and read quorum always intersect [21]. Thus, the latest changes committed to a write quorum will be visible to at least one node in the read quorum. Therefore, any read quorum can provide the latest version of the object. (Note that the rest of the nodes in a read quorum may have stale versions of an object.) Thus, the QR protocol ensures a consistent view of the most recently committed changes.

A transaction uses its read quorum and write quorum for reading from, or writing to objects and for propagating updates, respectively. For reading or acquiring a writable copy of an object, a transaction sends a request to its read quorum. The transaction selects the object copy with the latest version from all the copies received from the read quorum. This object copy is the most recent one in the system, at that point of time.

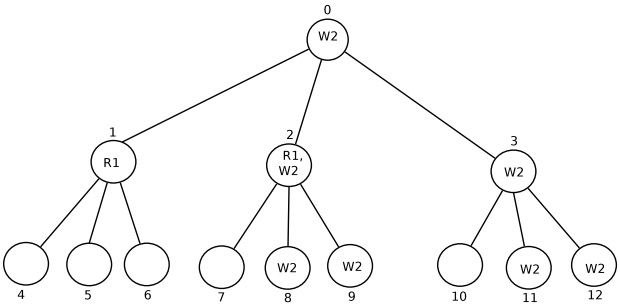


Figure 3: Ternary tree with 13 nodes.

For committing writes, a transaction uses a two-phase commit protocol to obtain consensus for commit from its write quorum. Initially, the transaction sends a commit request message to its write quorum. On every node of the write quorum, a decision for commit or abort is made based on the state of objects. If the node decides to commit the requesting transaction, it will lock the objects in write-set for the transaction by setting the object field $protected$ to $true$. If the node decides to abort, the object state remains unchanged. The decision is then sent back as reply to the requesting transaction. The transaction collects all the replies and commits only when it receives a *commit* message from all the nodes; otherwise, the transaction is aborted.

Quorums maintain *potential readers list* (PR) and *potential writers list* (PW) for every object. Whenever a read or a write request is processed for an object, the requesting transaction is added to the PW or PR, accordingly. These lists are used by contention managers to decide which transaction needs to be aborted or committed.

The nodes in QR form a logical ternary tree. Agrawal *et. al* [21] have defined the precedure for creating read and write quorum. A read quorum can be viewed as majority of children at a level, while write quorum can be viewed as majority of children at every level.

Figure 3 illustrates the process. The figure shows a tree with 13 nodes with read quorum as $R1 = \{n_1, n_2\}$ and write quorum as $W2 = \{n_0, n_2, n_3, n_8, n_9, n_{11}, n_{12}\}$. A transaction T_w writes to an object o_1 and commits the changes at time t using $W2$. All the nodes of $W2$ have the latest version of o_1. Now, another transaction T_r reads o_1 by requesting to $R1$ after time t. Since the intersection of $R1$ and $W1$ is n_2, n_2 has the latest version of o_1. T_r collects copies of objects from n_1 and n_2, and chooses the one sent by n_2.

A. System Model

We consider a distributed system which consists of a set of nodes that communicate with each other by message-passing links. We consider a set of *distributed transactions* $\mathcal{T} := \{T_1, T_2, \ldots\}$ sharing a set of objects $\mathcal{O} := \{o_1, o_2, \ldots\}$ distributed on the network. A transaction contains a sequence of requests, each of which is a read or a write operation request for an individual object, followed by a commit operation. We define the transactions under closed nesting and checkpointing below.

Closed Nesting Model: Following are the different kinds of transactions in the system [18].

- **Root transaction.** This transaction has a behavior similar to that of a flat nested transaction. The commit of a root transaction is globally visible – i.e., any transaction accessing objects after root transaction's commit will be able to view the changes. The abort of a root transaction will retry the transaction from beginning.
- **Closed Nested Transaction.** A closed nested transaction (CT) executes on behalf of the parent transaction. Commit of a closed nested transaction is not globally visible. Successful commit of CT moves the execution back to the parent, while an abort either retries the CT or its parent transaction.
- **Parent transaction.** A transaction is a parent transaction when it encloses atleast one CT. Upon successful commit of the CT, the parent transaction continues its execution.

Checkpointing Model: This model requires transactions to support checkpoint creation and rollback.

- **Checkpointing transaction.** A checkpointing transaction (CPT) creates checkpoints based on specific criterion. Note, this criterion is predefined in the system, as opposed to programmer created manual checkpoints [22]. Commit of CPT is globally visible.

III. QR-CN Protocol

A. QR-CN: Overview

A closed nested transaction obtains object copies from its read quorum. For the commit of a CT, it needs to validate objects in its read-set and write-set, and then merge these objects to the respective sets of the parent. In QR [19], validation is performed by sending a request to a write quorum. However, such a validation would increase message overhead for closed nested transactions, when compared to flat nested transactions.

We add an incremental validation mechanism to the read operation. This mechanism validates a transaction's

read-set and write-set objects on every read operation. This means that, when a read request is completed, the transaction's read-set is valid at that point of time. Further, when a transaction completes reading all the objects, its read-set and write-set objects are valid. As a result, a CT does not need to send a validation request to its write quorum and can commit without incurring any remote communications. For commit of CT, it only has to merge its read-set and write-set with its parent. Similarly, a read-only operation can commit without sending a commit request.

In the subsequent sections, we describe QR-CN protocol to support closed nested transactions. We describe the $read/write$ operation at local and remote nodes in Section III-B, and $commitCT$, commit operation for CT, in Section 3. The operations for $commit-request$, $commit$, and $abort$ are same as in QR, which we summarized in Section II.

B. Read/Write Operation

Read quorum validation (Rqv). Rqv is an incremental validation mechanism to detect early abort of a transaction. It helps CT and read-only transactions to commit locally.

Recall the following two properties of QR:

1) Every node in QR has copies of all the objects.
2) In QR, any read and write quorum intersect. Therefore, a read quorum can provide the latest version of every object.

From these properties, we can infer that a read quorum is aware of the latest version of all the objects in the system. It follows that validation can be performed on a read quorum for any set of objects. This observation is the basis of Rqv.

A read/write operation proceeds as follows. A transaction sends a read request for an object to its read quorum. A node in the read quorum first validates objects that are currently in the transaction's read-set and write-set. In the validation procedure, the versions of read-set and write-set objects are checked against the versions of objects present at that node. Validation is successful if the transaction's objects have versions equal to the object versions on that node; else it fails. On successful validation, the node proceeds to retrieve the copy of the requested object. However, if the validation fails for any of the objects, an abort message is sent back to the transaction.

For a flat nested transaction, an abort message implies abort of that transaction. For a CT, an abort message could mean abort of the CT or any of its parents. This is decided by the objects on which validation fails.

Consider again the example in Figure 3, with a read quorum R_1 and a write quorum W_2 intersecting at n_2. A transaction T_1 has read objects $O = \{o_1, o_2, o_3\}$ from its read quorum R_1. At this point, a conflicting transaction T_2 commits via write quorum W_2 and increments the version of object o_2. Next, T_1 requests object o_4 from R_1. n_1 will successfully validate T_1's read-set ($\{o_1, o_2, o_3\}$). However, n_2 will find that the version of o_2 has increased. Therefore, validation will fail and n_2 will send an abort message to T_1.

Algorithm 1: QR-CN: Read Quorum Validation for transaction.

```
   procedure VALIDATION (T)
1   Remote:
2   dataSet = getDataSet(T);
3   abortTxn = null;
4   foreach o in dataSet do
5       protected = getObj(o.id).isProtected;
6       ownerTxn = getObj(o.id).ownerTxn;
7       if o.version < getObj(o.id).version ||
          protected then
8           remove ownerTxn from PW,PR;
9           if isParent( ownerTxn, abortClosed) then
10              abortClosed = ownerTxn;
11  return abortClosed;
```

Algorithm 1 shows the read quorum validation procedure for a transaction T. $getDataSet$ traverses the parents of T and stores the objects read so far by them in $dataSet$. Each of the object copies have an $ownerTxn$, which refers to the transaction that reads the object. Each object o is checked for its validity (line 7). If the object is not valid, then $ownerTxn$ is removed from PR and PW lists (line 8). Then, we check whether $ownerTxn$ is higher in transaction heirarchy than $abortClosed$ (line 9). If it is, then $ownerTxn$ becomes the new value of $abortClosed$; else it remains unchanged. At the end of iteration, $abortClosed$ is the transaction highest in the hierarchy whose object is invalid and which needs to be aborted. However, if the value of $abortClosed$ is $null$, it means that validation is successful.

Algorithm 2 shows the read operation procedure, which uses the validation procedure in Algorithm 1.

A read request of a CT first recursively checks for the object in the read-set and write-set of the parents (line 2). If the object is found, the request is completed locally without incurring any remote call. If the object does not exist locally, a request for that object is sent to its read quorum. The remote node records the relationship between CT and the parent transaction, and performs validation for the transaction. If the validation succeeds (i.e., if the return value is $null$), the node

Algorithm 2: QR-CN: Read for CT.

```
   procedure READ (T, objId)
1  Local:
2  o = checkParent(objId);
3  if o == null then
4      objSet, abortClosed = READQUORUM (T, objId));
5      if abortClosed != null then
6          abort(abortClosed);
7          return;
8      o = latestVersion(objSet);
9  add o to T.readset;
10 Remote:
11 setChild(parent(T), T);
12 abortClosed = validate(T);
13 if abortT != null then
14     respond(T, abortTxn);
15     return;
16 o = getObj(objId);
17 if T is root then
18     add T to PR(o);
19 respond(T, o);
```

sends back the copy of the object. The remote node adds the objects to the PR/PW list only when it is a root transaction. It is necessary that we do not create any metadata for CT on the remote node. This ensures that the commit of CT happens locally.

The write procedure is the similar to the read procedure, except that $T.readset$ is replaced with $T.writeset$ in line 9 and PR is replaced with PW in line 18.

The local node, on receiving objects from read quorum, selects the object with the highest version number. If an abort message is received, either the CT or its parent transaction aborts, depending on the value of $abortClosed$.

C. QR-CN : Commit operation.

Algorithm 3 shows the procedure for commit of a CT. The local node merges the read-set and write-set of a CT with that of its parent.

Algorithm 3: QR-CN: Commit of CT.

```
   procedure COMMITCT (T)
1  Local:
2  parent = T.parent;
3  foreach o ← T.readSet do
4      add o to parent.readset;
5  foreach o ← T.writeSet do
6      add o to parent.writeSet;
```

IV. QR-CHK Protocol

A. QR-CHK: Overview

A CPT has the ability to rollback to previous exection state in case of transactional conflict. It has read and write operations similar to those defined in QR-CN, while the request-commit and commit operation are exactly the same as flat nested transaction. Transaction creates checkpoints whenever a pre-defined criterion is satisfied. These checkpoints are the points to which transaction can be rolled back to whenever a transactional conflict is detected. When a conflict is detected during request commit, the entire transaction is aborted and retried. In case of a conflict detected during read/write of a remote object, the transaction is partially aborted by rolling back to an appropriate checkpoint.

In DTM context, a checkpoint is defined as the state of the transaction at a specific point in time. The state consists of transaction's read-set, write-set and program state. A checkpoint is created whenever the number of objects in transactions's read-set and write-set crosses a threshold. Every checkpoint has a checkpoint ID representing the time at which it was created.

B. QR-CHK: Read/Write Operation

The read/write operation performs validation of objects which can result in partial abort of transaction, described in Algorithm 4. This process is similar to the read quorum validation and read operation described in Algorithm 1 and Algorithm 2, respectively. We record the latest checkpoint ID ($ownerChkpnt$) in the object copy whenever it is requested from remote node, similar to $ownerTxn$ in Rqv. The objects in read-set and write-set are scanned to find out the invalid objects and the least of the $ownerChkpnt$ among them, which is assigned to $abortChk$ (line 7-10 in Algorithm 4). The read-set and write-set corresponding to $abortChk$ will have valid objects, similar to $abortClosed$ in Rqv. An abort message is sent back to the transaction along with $abortChk$. The requesting transaction on receving $abortChk$ retrieves the corresponding checkpoint and resumes execution from the execution state associated with $abortChk$.

V. Analysis

Theorem V.1. *Rqv preserves* 1*-copy equivalence for all objects.*

Proof: Let T_1 be a transaction in either closed nesting (parent and child transaction) or checkpointing model. T_1 reads an object o at time t_1. At a later time t_2, T_1 sends a request for object o'. Let O be the set of objects in the read-set and write-set of transaction T_1 at t_2. Let T_c be any transaction that has started propagating changes to the object o at time t_3 such that it conflicts with T_1. Note that T_c can be a root, parent, or a child transaction. We will now analyze all possible

Algorithm 4: QR-CN: Read Quorum Validation for checkpointing.

```
   procedure VALIDATIONCHK (T)
1  Remote:
2  dataSet = getDataSet(T);
3  abortClosed = null;
4   foreach o in dataSet do
5       protected = getObj(o.id).isProtected;
6       ownerChk = getObj(o.id).ownerChk;
7       if o.version < getObj(o.id).version ||
        protected then
8           remove ownerChk from PW,PR;
9           if ownerChk < abortChk then
10              abortChk = ownerChk;
11 return abortChk;
```

cases based on the relationship between t_1, t_2, and t_3, and show that T_1 does not violate 1-copy equivalence for o.

T_c has committed changes to o before t_1. In this case, T_1 uses the latest version of o. This is because of the following property:

Let $data(o, v)$ be the object copy of o on node v. There exists a write quorum q_w such that $\{\forall v \in q_w\} \wedge \{\forall v' \notin q_w\}$, $data(o, v).version > data(o, v').version$. If any transaction T accesses o at time t, it collects a set of copies from a read quorum q_r. We know that $\exists v \in \{q_w \cap q_r\}$ such that $data(o, v)$ is collected by T. Note that read and write operations select the object copy with the highest version number. Hence, for any transaction T, $data(o, v)$ is selected as the latest copy.

Since T_c has committed changes on o before t_1, T_1 uses the latest version of o and does not find any conflict. The read request for o' succeeds.

T_c is attempting to commit changes after t_1 and before t_2. In this state, T_c has received the commit decision from its write quorum. Thus, any node in T_c's write quorum will either have applied the changes of T_c on o or would have set $o.protected = true$. While T_c is in this state, T_1 sends a read request for o' to its read quorum. Let q_r and q_w be the read quorum and write quorum of T_1 and T_c, respectively. Then $\exists v \in \{q_r \cap q_w\}$ such that the changes committed by T_c are applied on o or $o.protected = true$. Therefore, the read request of T_1 will be denied by such a node v, and the $abort$ message will be sent back. For closed nesting, the transaction $abortClosed$ will be aborted. For checkpointing, the transaction will be rolled back upto $abortChk$.

T_c is attempting to commit changes after t_2 and before the request commit of T_1. At t_2, the read quorum validation for T_1 succeeded as there was no conflict. Next, T_1 has completed reading all the remote objects and its next request will be commit request. Furthermore, T_c has received the commit decision from its write quorum, and any node in the write quorum will either have applied the changes of T_c on o or would have set $o.protected = true$. Let q_w and q'_w be the write quorums of T_1 and T_c, respectively. Then $\exists v \in \{q_w \cap q'_w\}$ such that the changes committed by T_c are applied on o or has $o.protected = true$. When the request commit for T_1 is sent to its write quorum, such a node v will send an abort message to T_1. Note that, this case is exactly the same as in QR [19].

T_1 reads from its data-set or parent data-set. If o has been read before by T_1 or any of the parent transaction, then T_1 will read the local copy of the object. In this case, the read quorum validation will not be performed. Instead, the object validation is performed whenever T_1 sends the next remote read request. If this was the last remote request, then validation is performed as part of request commit.

From all these cases, we see that transactions observing an inconsistent state of an object will never commit. Theorem follows. ■

From the above proof, we can easily prove that QR-CN and QR-CHK also preserve 1-copy equivalence. For a transaction in QR-CN and QR-CHK, the read/write cost is equal to transaction node's distance from read quorum (distance from farthest node in read quorum), while request commit and commit confirm cost is equal to its distance from write quorum. We show that QR-CN and QR-CHK guarantee opacity. More details can be seen in [23].

VI. Implementation and Experimental Evaluation

A. Implementation

We implemented QR-DTM, a Java based DTM. QR-DTM uses the QR protocol [19] as the basic algorithm for providing replication and cache coherency.

Figure 4 shows the architecture diagram for QR-DTM. It shows interaction of *Transaction Manager*, *Cluster Manager*, the nodes and their quorums in QR-DTM. A Transaction Manager performs two tasks i) It provides an interface for remote requests through *TM Proxy*. ii) *Context Delegator*, a sub-module in Transaction Manager, maintains the metadata regarding PR and PW lists of objects, relation between parent, child and root transactions.

The TM Proxy forwards the message request to Cluster Manager. The designated quorums for a node are

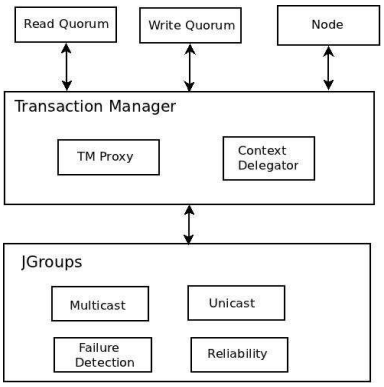


Figure 4: QR-DTM's architecture.

tracked by the Cluster Manager. The message received by Cluster Manager is multicast to the read quorum or write quorum depending on the type of the message. JGroups API handles the group communication between the nodes and their quorums. JGroups sends multicast and unicast messages in a reliable manner.

QR-DTM uses Java Exceptions for partial rollback in closed nesting. On abort of a transaction, an exception encapsulating the transaction $abortClosed$ is thrown. Transaction catching this exception compares $abortClosed$ with its ID. If it does not match, it throws another exception which is caught by its parent. This process continues until transaction Id matches $abortClosed$. At this point, the transaction discards the previous read and write sets, and starts afresh.

QR-DTM uses Java Continuations for partial rollback in checkpointing. Continuations provide a mechanism to save the current execution state, and resume from this saved state at a later time. During checkpoint creation, Continuation objects are saved along with a copy of transaction information. For restoring from a given checkpoint ID, the execution resumes from the state saved in corresponding Continuation object using transaction copy. Java Continuations require a customized JVM available at [24].

B. Experimental Settings

We conducted the experiments using a 40-node testbed. Each node is an AMD Opteron processor clocked at 1.9GHz and running the Ubuntu Linux 10.04 server operating system. Each node was assigned the same read and write quorums. The average round-trip network latency for a remote request was observed to be $\approx$30 milliseconds.

In section VI-C, we compare flat nesting (i.e. QR), closed nesting (i.e. QR-CN) and checkpointing (i.e. QR-CHK) for three micro-benchmarks including Hashmap, Red Black Tree (RBTree), Skiplist (SList), and macro-benchmarks including Bank (monetary) application (similar to the one in [4]) and STAMP benchmark [25] Vacation. In Section VI-D, we compare existing DTM implementations with QR-DTM for Bank benchmark. In Section VI-D, we show graceful degardation in failure scenario for three benchmarks.

C. Comparison between flat and closed nesting, and checkpoint

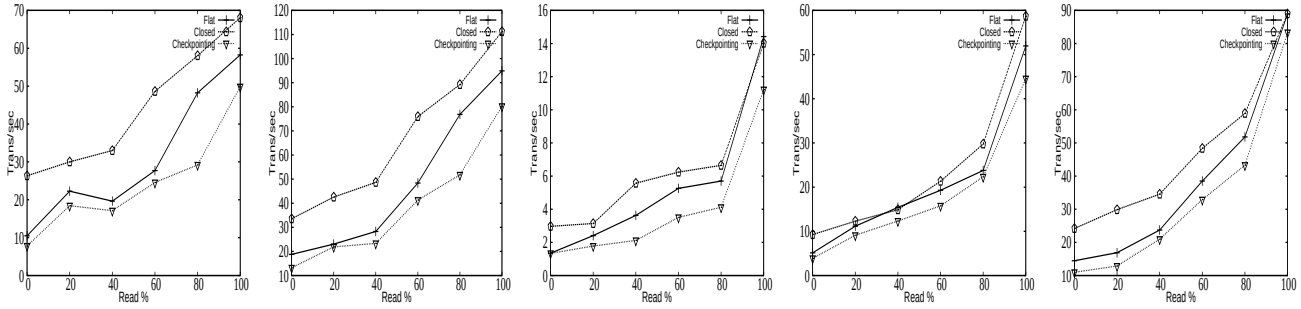
We compared closed nesting (QR-CN) and checkpointing (QR-CHK) with flat nesting (QR) under Bank, Hashmap, RBTree, SList and Vacation. Every root transaction consists of one or more CTs, where each CT is an operation on data structure. For example in hashmap, operation for adding or removing an element is a CT, and multiple such CTs are enclosed inside a root transaction. For vacation, each of the reservations for car, hotel and flight forms a CT.

In these experiments, we measured the throughput (transactions committed per second) by varying the following parameters: 1) read workload i.e. the percentage of read operations 2) the number of nested calls, which affects the number of objects read within a transaction, thereby controlling its length, 3) the number of objects, which could increase or decrease contention depending on the application.

Figures 5a, 5b, 5c, 5d and 5e show the variation with read workload for five benchmarks, varied from 0 to 100 %. Figures 6a, 6b, 6c, 6d and 6e show the variation with number of calls, varied from 1 to 5. Figures 7a, 7b, 7c, 7d and 7e show the variation with number of objects. Contention increases for SList and Hashmap benchmarks with increase in objects, while for remaining benchmarks contention reduces.

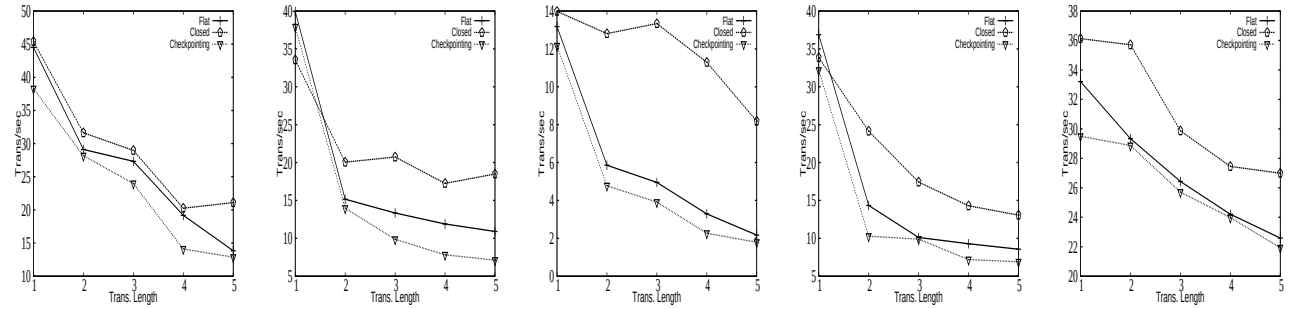
In all these runs, we observed that closed nesting outperforms flat nesting and checkpointing, and that checkpointing suffers from performance degradation over flat nesting. The best speedup obtained by closed nesting over flat nesting is for SList (101%) and the least speedup is for Bank (9%). We observed that checkpointing has 16% degradation over flat nesting across all benchmarks.

For read workload variation, we observed that the throughput improvement of closed nesting over flat nesting and checkpointing is greater for higher percentage of writes, while the gap reduces as the read workload increases. Similarly, for transaction length variation, throughput improvement for closed nesting increases as the length increases. For object variation, with increase in contention for benchmarks, we observed closed nesting performing better than other two models.



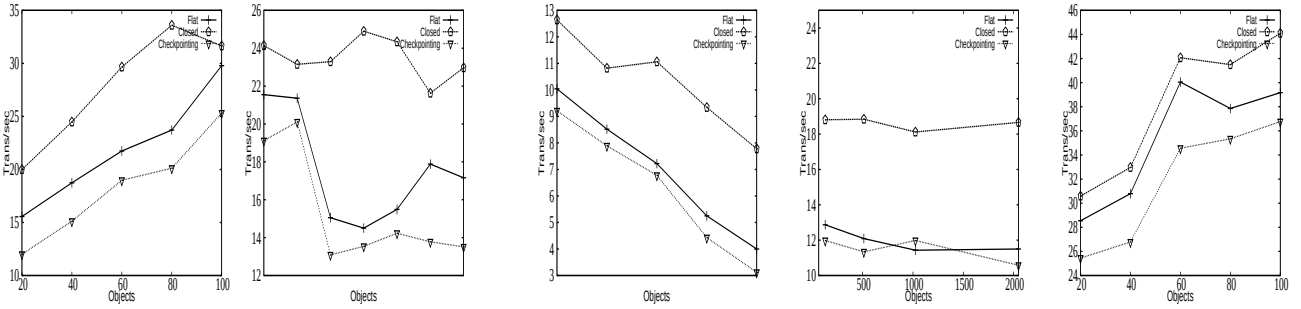
(a) Bank. (b) Hashmap. (c) Skiplist. (d) RBTree (e) Vacation.

Figure 5: Flat nesting, closed nesting and checkpointing for different read workload.



(a) Bank. (b) Hashmap. (c) Skiplist. (d) RBTree (e) Vacation.

Figure 6: Flat nesting, closed nesting and checkpointing for different transaction length.



(a) Bank. (b) Hashmap. (c) Skiplist. (d) RBTree (e) Vacation.

Figure 7: Flat nesting, closed nesting and checkpointing for different number of objects.

We also measured the transaction abort rates (i.e., root and child transaction aborts) and the number of messages exchanged (i.e., read and commit requests) for all benchmarks. Table 8 shows the percentage change in abort rate and messages exchanged with closed nesting and checkpointing compared to flat nesting. For QR-CN, there was decrease in the abort rate and message (denoted by negative values), while for QR-CHK it has increased. The independent evaluation to find the overhead for checkpoint creation shows that it has only 6 % overhead compared to flat nesting. Thus, the reason for overall drop in througput for QR-CHK is the fine granularity of checkpoints which results in large number of unnecessary partial aborts as can be seen in table 8. There is almost a direct correlation between the decrease in the number of messages and the total number of aborts and the throughput improvement. The least reduction in abort rate is observed for Bank and highest reduction is for SList.

From the results so far, we also observe that the improvement obtained by closed nesting increases with increase in contention. We also note that the length of

Bench.	QR-CN Abort %	QR-CHK Abort %	QR-CN Msg. %	QR-CHK Msg. %
Bank	-18	14	-22	15
Hashmap	-45	19	-51	22
SList	-56	23	-52	26
RBTree	-21	15	-23	16
Vacation	-33	11	-41	17

Figure 8: Abort rate and message % for QR-CN and QR-CHK compared to flat nesting

transactions has a significant effect in determining the throughput gain for closed nesting. The SList benchmark, which have large lengths, have high throughput gain (as high as 122 %). In contrast, the average gain for rest of the benchmarks with shorter lengths is 40%.

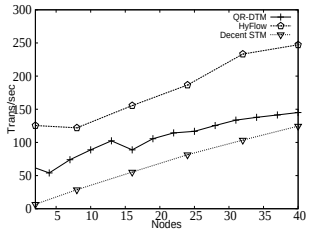
D. Comparison with DTM implementations

We compared QR-DTM with other DTM implementations including HyFlow [4] and Decent STM [6]. Decent STM uses a fully decentralized snapshot algorithm that relies on multiversion concurrency control: using a history of object states, conflicting transactions are allowed to proceed as long as they can see a consistent snapshot of memory. Thus, Decent STM is a fair competitor to QR-DTM. On the other hand, HyFlow uses an algorithm called Transaction Forwarding Algorithm (TFA), which is based on the single object copy model and therefore cannot cope with failures. TFA ensures transactional properties using an asynchronous clock-based validation technique. We still include HyFlow in our comparison because, for the no-failure case, TFA is shown to outperform Decent-STM in [26], and therefore serves as a good baseline for us.

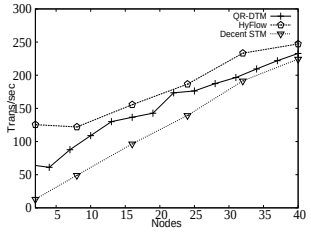
Figure 9 shows the comparison of the three DTM implementations under high and low contention for the Bank benchmark. We observe that QR-DTM consistently outperforms Decent STM. HyFlow performs the best.

These results indicate that Decent-STM's snapshot isolation algorithm has higher overhead than QR-DTM. The reason for the lower performance of QR-DTM than HyFlow is that any remote request for QR-DTM takes $30ms$, on average, in our testbed, compared to HyFlow's $5ms$. This is because, QR-DTM uses multicast, while HyFlow uses unicast for message passing. However, HyFlow cannot cope with failures.

Throughput under Node Failures We now show fault-tolerance of QR-DTM by measuring throughput under



(a) Bank : 10 % Read, 90 % Write.



(b) Bank : 90 % Read, 10 % Write.

Figure 9: Throughput of QR-DTM, HyFlow, and Decent-STM for the bank application

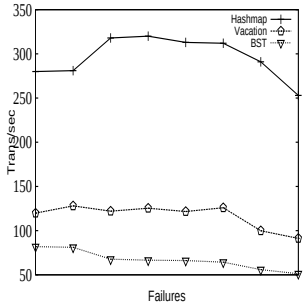


Figure 10: Throughput under increasing node failures.

node failures. Figure 10 shows the throughput for the Hashmap, Binary Search Tree (BST) and Vacation benchmark under increasing number of node failures. We consider a system with 28 nodes, where initially, a read quorum, consisting of a single node, is assigned to all the nodes. With each failed node, the size of the read quorum increases by one. The number of failed nodes (n) ranges from 1 to 8. Initially, we observe that the throughput increases for certain number of failure. This is because, the workload is balanced across the read quorum nodes. However, for n greater than 4, we observe that the throughput degrades gracefully as the messages exchanged increases due to the larger size of the read quorum.

VII. Related Work

Replication has been studied in DTM for improving concurrency and for coping with failures, largely in the context of cluster DTM [5], [6], [9], [13], [16],

[17]. These works provide fault-tolerance properties by inheriting fault-tolerance protocols from database replication schemes, which rely on broadcast primitives. D2STM [13], is a replicated DTM that provides strong consistency through a distributed certification scheme. Decent STM [6] implements a decentralized snapshot isolation protocol for guaranteeing consistency. GenRSTM [5] is a generic framework for replicated DTM, and supports replication via a replication manager, which is notified of updates made by local STMs. Zhang and Ravindran's QR protocol [19] uses a replicated DTM model that relies on quorums for managing transactional metadata, and ensures consistency using multicast among the replicas. QR does not consider transactional nesting or checkpointing. (QR is the foundation of our work.)

Transactional nesting has been studied for TM, but largely in the multiprocessor context. Earlier multiprocessor TMs either did not support nesting or simply flattened nested transactions into a single top-level transaction. Harris *et. al.* [27] argued that closed nested transactions, supporting partial rollback, are important for implementing composable transactions, and presented an orElse construct that relies on closed nesting. In [28], Adl-Tabatabai *et. al.* presented an STM that provides both nested atomic regions and orElse, and introduced the notion of mementos to support efficient partial rollback.

Recently, a number of researchers have proposed the use of open nesting in (multiprocessor) TM. Moss described the use of open nesting to implement highly concurrent data structures in a transactional setting [29]. In contrast to the database setting, the different levels of nesting are not well-defined; thus different levels may conflict. For example, a parent and a child transaction may both access the same memory location and conflict.

Atomos [30], TCC [31], and LogTM [32] describe HTM implementations of closed and open nesting, with commit and abort handlers for open nesting. Agrawal *et al.* [33] study the memory model semantics of open-nested TM. They describe ownership-aware transactions, which provide a disciplined methodology for open nesting, while guaranteeing abstract serializability.

Herlihy and Koskinen [22] proposed checkpointing and partial aborts (in multiprocessor TM), as an alternate to nesting. They argued that fine grained checkpointing can be achieved and closed nesting is a more rigid alternative.

None of the DTM efforts [4]–[6], [9], [13]–[15], [34] consider transactional nesting or checkpointing. The nested DTM works that we are aware of include the N-TFA protocol [35], which supports closed nesting, and the TFA-ON protocol [36], which supports open nesting. N-TFA extends Saad and Ravindran's TFA algorithm, which uses an asynchronous clock-based validation technique to ensure DTM transactional properties, to support closed nesting. The work [35] reports 2% average performance benefit for closed nesting compared to flat nesting (and 84% speedup in certain cases). In TFA-ON [36], abstract locks are used to guarantee that no data conflicts occur. The average speedup for open nesting is 30% compared to flat nesting [36]. However, N-TFA and TFA-ON use a single copy DTM model and therefore are not fault-tolerant. In contrast, we consider nesting (QR-CN) and checkpointing (QR-CHK) in replicated (and thus fault-tolerant) DTM, and is the first work to do so.

VIII. Conclusions

We presented the QR-CN and QR-CHK protocols that supports closed nesting and checkpointing in quorum-based replicated distributed TM. We showed that Rqv ensures 1-copy equivalence. Our implementation and experimental evaluation shows that closed nesting (with QR-CN) improves throughput over flat nesting: the average performance gain is 53% across all the benchmarks, while the highest speedup is 101%. The reason for performance gain is the average reduction of 33% in abort rate. The lower abort rates, in turn, are responsible for reducing the communication overhead by 34 %. We observed performance degradation of 16% for checkpointing over flat nesting across benchmarks, owing to message overhead of 19%.

We determined that closed nesting best applies for applications with high contention. We also found that the length of transactions is an important factor in the performance of closed nesting. We observed that the performance of closed nesting increases with increase in the level of contention and transaction length.

References

[1] M. Herlihy, "The art of multiprocessor programming," in *PODC*, 2006, pp. 1–2.

[2] J. R. Larus and R. Rajwar, *Transactional Memory*, 2006.

[3] B. Saha, A.-R. Adl-Tabatabai *et al.*, "McRT-STM: a high performance software transactional memory system for a multi-core runtime," in *PPoPP*, 2006, pp. 187–197.

[4] M. M. Saad and B. Ravindran, "Distributed Hybrid-Flow STM : Technical Report," ECE Dept., Virginia Tech, Tech. Rep., 2010. [Online]. Available: http://hyflow.org/trac/hyflow/wiki/Publications

[5] P. R. N. Carvalho and L. Rodrigues, "A generic framework for replicated software transactional memories," in *IEEE NCA11*, 2011, pp. 271–274.

[6] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized stm algorithm," in *IPDPS*, April 2010, pp. 1 –12.

[7] K. Arnold, R. Scheifler *et al.*, *Jini Specification*, 1999.

[8] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java application partitioning," in *ECOOP*, 2002, pp. 1–3.

[9] R. L. Bocchino *et al.*, "Software transactional memory for large scale clusters," in *PPoPP*, 2008, pp. 247–258.

[10] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," *Distributed Computing*, vol. 20, no. 3, pp. 195–208, 2007.

[11] B. Zhang *et al.*, "Relay: A cache-coherence protocol for distributed transactional memory," in *OPODIS*, 2009, pp. 48–53.

[12] M. M. Saad and B. Ravindran, "Supporting STM in distributed systems: Mechanisms and a Java framework," in *TRANSACT*. ACM, 2011. [Online]. Available: hyflow.org

[13] M. Couceiro *et al.*, "D2STM: Dependable distributed software transactional memory," in *PRDC*, 2009, pp. 307–313.

[14] P. Romano, N. Carvalho, M. Couceiro, L. Rodrigues, and J. Cachopo, "Towards the integration of distributed transactional memories in application servers clusters," in *QoSHN*, 2009, pp. 755–769.

[15] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, "Cloud-TM: harnessing the cloud with distributed transactional memories," *SIGOPS*, pp. 1–6, 2010.

[16] C. Kotselidis *et al.*, "DiSTM: A software transactional memory framework for clusters," in *ICPP*, 2008, pp. 51–58.

[17] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho, "Evaluating database-oriented replication schemes in software transactional memory systems," in *IPDPSW*, 2010, pp. 1–8.

[18] J. E. B. Moss and A. L. Hosking, "Nested transactional memory: model and architecture sketches," *SCP*, pp. 186–201, 2006.

[19] B. Zhang and B. Ravindran, "A quorum-based replication framework for distributed software transactional memory," *PoDS*, pp. 18–33, 2011.

[20] P. Bernstein and N. Goodman, "Multiversion concurrency controltheory and algorithms," *TODS*, pp. 465–483, 1983.

[21] D. Agrawal and A. El Abbadi, "The tree quorum protocol: An efficient approach for managing replicated data," in *VLDB*, 1990, pp. 243–254.

[22] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *SPAA*, 2008, pp. 160–168.

[23] A. Dhoke and B. Ravindran, "On Closed Nesting in Replicated Distributed Transactional Memory," ECE Dept., Virginia Tech, Tech. Rep., Sep 2012. [Online]. Available: http://www.hyflow.org/hyflow/raw-attachment/wiki/Publications/qr-cn-tech-report.pdf

[24] J. Rose. Multi-language virtual machine. [Online]. Available: http://hg.openjdk.java.net/mlvm/mlvm/summary

[25] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," in *IISWC*, 2008, pp. 35–46.

[26] M. M. Saad and B. Ravindran, "Transactional forwarding algorithm," ECE Dept., Virginia Tech, Tech. Rep., January 2011. [Online]. Available: hyflow.org

[27] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy, "Composable memory transactions," in *PPoPP*, 2005, pp. 48–60.

[28] A.-R. Adl-Tabatabai, B. T. Lewis, V. Menon, B. R. Murphy, B. Saha, and T. Shpeisman, "Compiler and runtime support for efficient software transactional memory," in *PLDI*, 2006, pp. 26–37.

[29] J. E. B. Moss, "Open nested transactions: Semantics and support," in *WMPI*, 2005.

[30] B. D. Carlstrom, A. McDonald, H. Chafi, J. Chung, C. C. Minh, C. Kozyrakis, and K. Olukotun, "The Atomos transactional programming language," in *ACM SIGPLAN*, 2006, pp. 1–13.

[31] A. McDonald, J. Chung, B. D. Carlstrom, C. C. Minh, H. Chafi, C. Kozyrakis, and K. Olukotun, "Architectural semantics for practical transactional memory," *SIGARCH*, pp. 53–65, 2006.

[32] M. J. Moravan *et al.*, "Supporting nested transactional memory in logTM," in *ASPLOS*, 2006, pp. 359–370.

[33] K. Agrawal, I.-T. A. Lee, and J. Sukha, "Safe open-nested transactions through ownership," in *SPAA*, 2008, pp. 110–112.

[34] K. Manassiev *et al.*, "Exploiting distributed version concurrency in a transactional memory cluster," in *PPoPP*, 2006, pp. 198–208.

[35] A. Turcu, B. Ravindran, and M. Saad, "On closed nesting in distributed transactional memory," in *ACM SIGPLAN*, 2012.

[36] A. Turcu and B. Ravindran, "On open nesting in distributed transactional memory," in *SYSTOR*, 2012.