On Reducing False Conflicts in Distributed Transactional Data Structures

Aditya Dhoke
Virginia Tech
adityad@vt.edu

Roberto Palmieri
Virginia Tech
robertop@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

ABSTRACT

We present techniques for reducing false conflicts in distributed transactional data structure (DDS). The open nesting transactional model is the common solution because it allows nested transactions to commit independently of their parent transaction, thereby objects in the transaction read-set and write-set are released early, minimizing aborts due to false conflicts and improving concurrency. We present three protocols for avoiding false conflicts in DDS. Our first protocol, QR-ON, incorporates open nesting into the QR protocol that manages concurrency control for distributed transactional memory systems using quorum-based replication. We then introduce Optimistic Open Nesting, QR-OON, in which open-nested transactions commit asynchronously so that subsequent transactions can proceed without waiting for the commit of previous transactions. Finally, we propose an early release methodology, QR-ER, in which objects that do not affect the final state of the shared data are dropped from transaction's read-set, which avoids false conflicts and reduces communication costs. Our implementation and experimental studies revealed that QR-OON outperforms QR-ON by up to 43%, and that QR-ER outperforms QR-ON and QR-OON by up to 10×.

Categories and Subject Descriptors

D.1.3 [**Programming Techniques**]: Concurrent Programming; H.2.4 [**Systems**]: Transaction processing

General Terms

Transactional Memory, Algorithm, Transactions

1. INTRODUCTION

With the advent of multi-core architectures, application software performance can no longer be improved by simply relying on increased clock speeds; performance can only be improved by exposing greater concurrency. Since the presence of sequential code in a concurrent program – often required for concurrency control – significantly limits the program's speedup, sequential code must be minimized as much as possible. As coarse-grained locking increases sequential execution time, and fine-grained locking (and lock-free synchronization) have poor programmability, alternative concurrency control techniques such as software transactional memory (STM) [27] are increasingly gaining traction. With STM, which is inspired by database transactions, programmers write code that access shared memory objects using the abstraction of memory transactions provided by a software framework. The framework transparently ensures transaction properties including atomicity, consistency, and isolation, thereby significantly simplifying the development of concurrent applications. Similar to STM, distributed STM (or DTM) [7, 8], which extends STM to distributed systems (i.e., those based on message passing) is also showing increasing promise.

STM's growing interest has led to the extension of well known concurrent data structures (e.g., concurrent Linked List, Hashmap, Binary Search Tree) with transactional support, in both multiprocessor [14] and distributed [19, 28] contexts. While the benefits of transactional data structures under multiprocessor settings are well-known, they can be equally useful in distributed systems.

Some commercial products are implementing interfaces of well-known data structures. Such interfaces can also be helpful in the integration of legacy applications. For example, Infinispan [19] and Oracle Cache Coherence are in-memory transactional distributed data grids designed from the ground up to be scalable. They extend *java.Map*, offering APIs such as *put* and *get*. Developers can invoke these APIs inside an atomic code block, and execute that block as a transaction. From an user standpoint, the difference between interacting with a local hash-map and its distributed version is minimal, e.g., only the system's configuration file needs modifications.

In classical transaction processing [5], a conflict occurs among transactions when at least one of them is writing to the same object. This can cause a *read/write* or *write/write* conflict. One of the involved transactions must be aborted to resolve the conflict. In addition, transactions also suffer from another type of conflict, called *false conflict* [25, 23], which occurs among transactions performing seemingly independent operations. For example, consider a `set` implemented using a sorted list. Insertion of an element in the set can be viewed as a high level operation, while insertion of an object in the sorted list can be viewed as a low level operation. To insert an object O_1 between objects O_2 (smaller)

.

and O_3 (larger), a transaction T must traverse from the head of the list, and read all objects prior to O_1. Ideally, any invalidation due to concurrent writes on objects prior to O_1 would not compromise T's correctness and should not create any conflicts. However, high level operations, even though semantically independent, traverse the same set of objects during their execution, causing such false conflicts.

False conflicts have a significant impact in systems where network communication costs dominate transaction execution – i.e., distributed systems. In such systems, traversing a list may involve accessing different physical nodes. For example, let a transaction T access objects $\{O_X, O_Y, O_Z\}$, where O_X, O_Y, and O_Z are shared objects in a list of 100 elements that are physically spread on 10 nodes. In the worst case, O_X, O_Y, and O_Z are at the tail of the list. Each time T interacts with an object (read/write), it has to traverse 10 remote nodes to reach that object. In the meanwhile, if other transactions are concurrently updating the list, then they will invalidate objects already accessed by T during its traversal, aborting T (due to a false conflict). Also, when other transactions attempt to access O_X, O_Y, and O_Z, they could be aborted for the same reason. Moreover, when long-running transactions operate on the same shared data structure, false conflicts can prevent those transactions from successfully committing. This scenario is exacerbated in fault-tolerant DTM systems, in which multiple replicas must be updated after a transaction's commit. In this paper, we focus on such systems, where false conflicts can significantly degrade performance. Our target is improving performance of transactional data structures deployed in a distributed system, relying on the same appealing programming abstraction as DTM.

False conflicts have been previously studied. In particular, the DBMS literature has extensively studied the problem of preventing physical memory conflicts in data types that define commutative operations [12, 30]. However, these solutions cannot be used as-is in fault-tolerant DTM systems because of two fundamental reasons. First, the (theoretical) performance gain of such solutions has been studied largely on centralized systems; those gains are negated by the high cost of remote synchronization in DTM. Second, historically in DBMS settings, concurrency control is usually eager (i.e., locks are acquired at encounter time) to mitigate the relatively expensive cost of transaction roll-back (due to interactions with the stable log). In contrast, STM concurrency control protocols are mostly optimistic, acquiring locks only at commit time, to reap the advantage of relatively low-cost transaction roll-back (due to in-memory processing) and short transaction execution time. Classical eager approaches, when deployed in DTM, will block transaction execution due to encounter-time lock acquisition, decreasing performance.

Transaction execution characteristics in distributed systems are significantly different from multiprocessor systems. Unlike in a multiprocessor, in a distributed system, the cost of communication dominates overall transaction execution time. Moreover, replication (often used for fault-tolerance) incurs additional overhead due to the need for multicast or broadcast to the replicas. Any transactional conflict encountered during execution will result in transaction aborts, thereby wasting processor and costly network resources.

Moss [23] and Herlihy *et al.* [14] have proposed solutions to the problem of false conflicts. Moss [23] presented the open nesting model, wherein low-level operations within a transaction form open-nested transactions. The commit of these transactions are globally visible immediately, even though the parent transaction may not have committed yet. The memory release of low-level operations improves concurrency among the high-level operations. Abstract locks are used to prevent accesses on the same elements. The open nesting model was developed in [23] for centralized concurrency control. One of the contributions of our work is applying the same approach to distributed concurrency control, and understanding its effectiveness in the presence of different overheads in DTM (e.g., remote synchronization cost). In [14], Herlihy *et al.* introduced transactional boosting, where transactional support is provided on top of a concurrent list using abstract locks to boost performance. Boosting eagerly modifies shared objects while the transaction is executing. If applied to DTM, it increases the number of remote messages in the transaction's critical path, significantly degrading overall performance.

Motivated by these observations, with the goal of reducing false conflicts in DTM, and thus reduce the likelihood of transaction aborts and improve performance, we present the design and implementation of three protocols. Each protocol targets different transactional and system models. As a baseline DTM concurrency control protocol, we consider [31]'s quorum-based replicated DTM protocol, called QR (Section 2). In QR, transaction execution is divided into two independent phases with orthogonal responsibilities: 1) read/write phase, in which a transaction executes optimistically and obtains the latest copies of objects, and 2) commit phase, in which read objects are validated and write objects are committed. The protocol uses the quorum intersection property for providing the latest copy of objects and detecting conflicts in the presence of node/link failures (Section 2.3). We consider QR as a baseline DTM protocol, as it exhibits good availability, complexity, and scalability properties.

Our first protocol is called QR-ON, for QR with *open nesting* (Section 4). QR-ON incorporates the open nesting model into the QR protocol (both the open nesting model and QR are summarized in Section 2 for completeness). In open nesting, only the objects accessed within the (open) nested transactions are validated and (globally) released after successful commit. This early memory release increases the potential for improving concurrency: two parent transactions that have read or written the same set of objects in their inner transactions will not detect any conflict during their commit. Since each nested transaction directly commits to the shared (or possibly distributed) memory, other transactions can also immediately access the just committed data.

The open nesting model distinguishes between physical serializability (i.e., at the memory-level) and abstract serializability (i.e., at the semantic level). Open nesting breaks transactions' isolation at the memory level, but preserves serializability at the abstract level using abstract locks. If the application checks for the presence of an abstract lock on an accessed object before interacting with its physical memory location, then serializability is preserved. Also, reorganizing transactions into nested transactions enables splitting the validation and commit phases into multiple, smaller pieces that can be more easily successfully committed.

At its core, QR-ON exploits open nesting for speeding

up the validation of parent transactions. This has a drawback. For fault-tolerant protocols with a commit phase that is inherently an order of magnitude slower than the rest of the transactional execution, repeating the commit as many times as there are (open) nested transactions can negate the benefits of open nesting's potential for increased concurrency due to early memory release. Thus, we propose a second protocol called QR-OON, for QR with *optimistic* open nesting (Section 5). QR-OON makes QR-ON's commit phase non-blocking: an open-nested transaction locally commits, speculatively, without blocking, allowing subsequent transactions to start their execution without waiting for its commit. This causes an overlap between the commit of an open-nested transaction and the read/write phase of the subsequent transactions, thereby reducing overall transaction execution time. The approach pays off when the subsequent open-nested transactions are likely to access the data written by the previous, still committing, transaction. In this way, the subsequent transactions will speculatively access the pre-committed version of the data written by the committing transaction. This optimism, in case of successful finalization of a, so called, *asynchronous commit*, allows it to completely overlap the commit phase with transactional execution, thereby mitigating the cost of the expensive commit.

Both QR-ON and QR-OON exploit open-nested transactions for reducing the size of the read-set when the parent transaction commits. Even though in QR-OON, the expensive cost of a commit is alleviated by the asynchronous implementation, fault-tolerant protocols extensively use the network during transaction execution. In such cases, the overlapping time may be limited because few speculative nested transactions can execute concurrently with the commit phase (our experiments revealed up to three). Therefore, we propose a third protocol called QR-ER, for QR with *early release* (Section 6). QR-ER does not rely on open nesting. Instead, false conflicts are avoided by dropping those objects from the read-set that do not need to be validated because, even in case of invalidation, they do not compromise correctness of the execution. This approach is suited for transactional data structures and for protocols that require a significant amount of network communication for executing operations and/or the validation and commit phases. With early release, each transaction locally decides if it wants to exclude objects from the read-set, according to the semantics of the data structure used, thereby saving additional messages. Even though the size of the read-set when the transaction reaches the validation phase is not as small as that in open nesting, it is composed of the minimum number of objects needed for ensuring execution correctness.

We implemented all three protocols in QR-DTM [31, 9], an open source DTM framework, written in Java. We conducted experimental studies using distributed data structures such as Linked List, Hashmap, Binary Search Tree, and a version of the TPC-C benchmark [1] implemented using a distributed hash table (as also used in [26]) (Section 7). Our studies revealed that QR-OON outperforms QR-ON by up to 43%. Additionally, they showed that QR-ER outperforms QR-ON and QR-OON by up to 10×.

2. BACKGROUND

2.1 Open Nesting

Open nesting was introduced by Moss [23] as a solution for avoiding false conflicts. The idea behind open nesting is to commit nested transactions to shared memory, thereby releasing objects in its read-set and write-set, and thus avoid conflicts with transactions working on the same set of objects.

Open-nested transactions optimistically commit changes, which are immediately made globally visible, assuming that the parent transaction will subsequently commit. However, if the parent transaction aborts, those changes must be compensated before the parent transaction can be re-issued. The compensation action does not simply restore the original state of memory, but restores the semantic state of the shared data. For example, the compensation action of adding an element to a set is removing that element from the set (i.e., remove is the *inverse* operation of add). Thus, operations with well defined inverses are appropriate to be open-nested, such as those of collection classes (e.g., set, map).

Globally committing changes of the nested transactions allows other transactions to proceed without encountering conflicts. However, there is a restriction on which transactions can execute concurrently. We illustrate this with an example. Figure 1 shows two parent transactions, *T_1* and *T_2*, each performing *add* and *contains* operations on a set as open-nested transactions. On completion of execution of the two nested transactions, the set should either have the element *e* or *e'*, but not both. If *T_1* and *T_2* complete their *contains* operation, then both will proceed to their *remove* operations, resulting in an incorrect state. This occurs because there is no way for transactions to know what operations are being performed by others. This problem is solved by using an *abstract lock*: an open-nested transaction will acquire an abstract lock corresponding to the operation it has performed to inform other transactions. In the above example, after *T_1* finishes its *contains*, it will acquire an abstract lock on *e*. This will prevent *remove(e)* of *T_2* from committing. Note that *contains(e)* and *contains(e')* can commit at the same time, as each will acquire different abstract locks i.e., on *e* and *e'*, respectively.

```
T_1                              T_2
   if (set.contains(e))             if (set.contains(e'))
      set.remove(e');                  set.remove(e);
```

Figure 1: Concurrent transactions in the open nesting model.

The abstract lock acquired by an open-nested transaction is added to the parent transaction's log. The parent transaction is responsible for releasing the locks acquired by its open-nested transactions. On successful commit, the parent transaction will commit its changes and release the abstract locks in its log. On abort or unsuccessful commit, the parent transaction will run the compensating actions for its committed open-nested transactions, release the corresponding abstract locks, and restart itself from the beginning.

There is no global order that is followed by open-nested transactions in acquiring abstract locks. As a result, deadlocks can occur. If an abstract lock is unavailable for an open-nested transaction, the parent transaction aborts to release all previously acquired abstract locks, and thus avoids deadlocks.

Correctness. In the open nesting model, transactional isolation is violated as partial changes of a transaction are exposed to other concurrent transactions before the commit phase. However abstract locks are acquired on those uncommitted objects such that other transactions are prevented from interact with them. Thus, open-nesting preserves *abstract serializability* [25].

Programmability. The open nesting model affects programmability: the programmer must understand that physical serializability is compromised for the sake of improved performance. Moreover, the programmer must identify operations that are appropriate for open nesting and provide compensating actions for them. For our early release protocol, the programmer must also identify the objects that can be dropped from the read-set.

2.2 Closed Nesting

The closed nesting transactional model has been introduced in [24, 21]. In this subsection we just briefly overview this model because in this paper we do not extensively use it. Only the QR-ER protocol has similarities with it.

Closed nesting allows inner transactions to abort individually. Aborting an inner-transaction does not necessarily lead to also aborting the parent transaction (i.e., partial rollback is possible). However, inner-transactions' commits are not visible outside the parent transaction. An inner-transaction commits its changes only into the private context of its parent transaction, without exposing any intermediate results to other transactions. Only when the parent transaction commits is the shared state modified.

2.3 QR: Quorum-based Replication

The QR protocol [31] provides concurrency control for objects via STM and fault-tolerance by maintaining copies of each object on all the 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 a transaction's read and write requests on objects, while a write quorum is used to commit changes to objects through two-phase commit. A transaction executing on a node uses the read and write quorums designated to that node. (Hereafter, when we say a node's or transaction's quorum, we will refer to these designated quorums.)

The QR protocol ensures 1-copy equivalence [5], meaning that when a transaction reads an object, it will use the latest copy of the object. This is because any write quorum and read quorum always intersect [3]. Thus, the latest changes committed to a write quorum will be visible to at least one node in a 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.

For committing writes, a transaction uses a two-phase commit protocol to lock written objects from its write quorum. Initially, the transaction sends a commit request message to its write quorum. On every node of the write quorum, the read-set and write-set objects of the transaction are validated by comparing their version numbers with those of the objects on the write quorum node. On successful validation, the node decides to commit the transaction and locks objects corresponding to write-set which prevents any further reads or writes to them. On unsuccessful validation, the node decides to abort the transaction and the object state remains unchanged. The decision (commit or abort) is then sent back as reply to the requesting transaction. The transaction collects replies from the write quorum nodes and commits only when it receives *commit* message from all; otherwise, the transaction is aborted. Finally, the requesting transaction sends its decision back to the write quorum nodes, who will release the locks on transaction's write-set objects, if necessary.

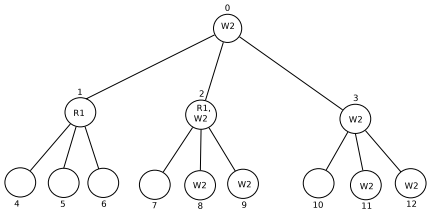


Figure 2: Ternary tree with 13 nodes.

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* [3] have defined the procedure for creating read and write quorums. A read quorum can be viewed as the majority of children at a level, while write quorum can be viewed as majority of children at every level.

Figure 2 illustrates the process through a simple example. 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.

Assume that n_2 fails. As per the protocol for read quorum, $R1$ can be reconfigured to $\{n_1, n_7, n_8\}$, where n_2 is replaced by the majority of its children. As per the protocol for write quorum, $W2$ is reconfigured to $\{n_0, n_1, n_4, n_5, n_3, n_{11}, n_{12}\}$) to form majority at all the levels. In this new configuration, the intersection has changed from n_2 to n_1.

3. SYSTEM MODEL

We consider a distributed system which consists of a set of nodes that communicate by message-passing links. 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 over the network is assumed. A transaction consists of a sequence of requests,

each of which is a read or a write operation request for an object, followed by a commit operation. An object has the following meta-data that are used for QR's operations:

1. *Version number* maintains the object's version number and is used during object validation.
2. *Protected* is a boolean field. When *true*, read or write to the object is disabled until commit is complete, after which it is set to *false*.
3. *Validate* is set to *true* or *false* depending on whether the object needs to be validated or not, respectively (see Section 6).

In addition, objects contain data structure-specific fields: *next* for Linked List, and *left* and *right* for Binary Search Tree.

For the sake of this presentation, we refer to the requesting transaction as the *client*, and the quorum node as the *server*. The term *data-set* refers to the read-set and write-set of a transaction.

Two of the approaches we present leverage on nested transactions. In this paper we limit to only one level of nesting. A transaction T_{N2} cannot start in the context of another transaction T_{N1} if T_{N1} is already nested in its parent transaction. However, due to the complexity of designing multiple nesting levels, usually, only one level is considered in transaction processing [22, 9]. We believe this assumption does not represent a restriction.

4. QR-ON: OPEN NESTING

In QR-ON, the read and write operations of both the nested and parent transactions are exactly the same as that of QR's, described in Section 2.3.

The commits of nested transactions are globally visible. A nested transaction's commit operation will validate the transaction's read-set and write-set objects, and attempt to acquire the abstract lock, corresponding to the transaction operation. On successful commit, the nested transaction will commit its changes, discard the read-set and write-set, acquire the abstract lock, and record the abstract lock in parent transaction's log. The parent transaction can then continue further execution. Once the parent transaction completes execution, its commit operation will validate the parent's read-set and write-set objects, and release the abstract locks acquired by its nested transactions. Note that the parent validates only the objects it has read, and not those read by its nested transactions.

A parent transaction could abort anytime during its execution, either because it has detected a conflict for objects in its data-set or one of its nested transactions could not acquire the needed abstract lock. When a parent transaction aborts, it will perform compensating actions for the operations committed by its nested transactions and release the corresponding abstract locks.

A nested transaction could also abort because of a conflict detected for objects in its data-set. In this case, it will rollback and restart from its beginning. The parent transaction's state is not altered when its nested transaction aborts.

Algorithm 1 describes the methods of QR-ON. A method partly executes on the requesting node (*Local*) and partly on the remote quorum nodes (*Remote*).

We now describe each method of QR-ON:

- *OpenNestedCommit*. This method performs the commit operation of a nested transaction. The remote or write quorum nodes perform object validation for the nested transaction and acquire the abstract lock corresponding to the operation (lines 2-3). On the local node, the result is *commit*, if object validation and abstract lock acquisition succeed for every write quorum node; else it returns *abort*. In case of commit, this method will record the operation details and abstract lock in the parent transaction log (line 14). In case of abort, *onOpenNestedAbort* is invoked.
- *parentCommit*. This method performs the commit operation of a parent transaction. The remote or write quorum nodes perform object validation for the parent transaction (line 24). If validation succeeds, abstract locks are released. On the local node, the result is *commit* if object validation succeeds on every write quorum node; else it returns *abort*. In case of abort, *onParentAbort* is invoked.
- *onOpenNestedAbort*. This method is invoked when a nested transaction aborts. An abort occurs if a conflict was detected for any of the objects in its data-set, or the abstract lock could not be acquired. In the former case, the nested transaction is retried. In the latter case, to avoid a deadlock, the parent transaction must release the previously acquired abstract locks, and invoke *onParentAbort* (to abort the parent transaction).
- *onParentAbort*. This method is invoked when a parent transaction aborts. It performs the compensation actions corresponding to the abstract locks in its log. For each abstract lock in the log, it performs a compensating action, which is another transaction that undoes the effects of the original operation. Additionally, the remote node releases the abstract locks (line 39).

5. QR-OON: OPTIMISTIC OPEN NESTING

In QR-ON, the commit of a nested transaction is a blocking operation i.e., the subsequent transaction has to wait until the commit of the previous transaction completes. In QR-OON, we relax this: a nested transaction commits asynchronously without blocking the next transaction. Additionally, a nested transaction commits its changes locally so that the next transaction can speculatively read those changes. This allows the current transaction's (*curr*) commit phase to overlap with the next transaction's (*next*) read/write phase.

The read phase of a nested transaction always executes in the context of the default thread, i.e., *defThread*. The commit of *curr* involves merging its data-set with the parent's data-set, and delegating the commit phase to a separate thread, *asyncThread*. As a result of this delegation, the execution of *next* can be started by *defThread*, while *asyncThread* is still committing *curr*. During *next*'s read phase, for processing an object request, the object is first locally looked-up in the parent's data-set. If the object is not found, it is fetched from the read quorum nodes. During each operation of *next*, the final outcome of *curr*'s commit operation is checked. If the outcome is a successful commit of *curr*, then *next* can continue its execution without any change and proceed to commit. However, if *curr* failed to commit, then it has to be retried. This involves discarding the speculative work done by *next* and rolling back *defThread* to the start of *curr*.

Rollback is achieved by means of checkpointing. At the start of every nested transaction, a checkpoint is created, which contains the execution state of the *defThread* along with its metadata. When an abort is detected, the execution of *defThread* is restored to the checkpoint corresponding to the aborted transaction.

Algorithm 1: Methods of QR-ON.

```
   procedure OpenNestedCommit (T)
 1  Remote:
 2  valid = validateObjects(T);
 3  lock = acquireAbstractLock(T);
 4  if valid and lock then
 5      return commit;
 6  else
 7      if !lock then
 8          reason = lockUnavailable;
 9      else
10          reason = objectInvalid;
11      return abort, reason;
12  Local:
13  if result == commit then
14      T.parentLogAdd(T.op,
        T.abstractLock);
15      return true;
16  else
17      onOpenNestedAbort(T, reason);

   procedure onOpenNestedAbort
   (T, reason)
18  Local:
19  if reason == objectInvalid then
20      retry T;
21  else
22      onParentAbort(T);
   procedure parentCommit (T)
23  Remote:
24  valid = validateObjects(T);
25  if valid then
26      releaseAbstractLocks(T);
27      return commit;
28  else
29      return abort;
30  Local:
31  if result == commit then
32      return true;
33  else
34      onParentAbort(T);

   procedure onParentAbort (T)
35  Local:
36  foreach op in T.opLog do
37      op.compensateRemote();
38  Remote:
39  op.releaseAbstractLock();
```

Note that, QR-OON needs changes on the client node (with respect to QR-ON), while the processing at server or quorum nodes remains the same as in QR-ON.

Algorithm 2: Methods of QR-OON.

```
   procedure readObject
   (T, objId, validate)
 1  DefThread:
 2  checkState(T.prev);
 3  ob=checkParent(objId);
 4  if ob != null then
 5      return ob;
 6  else
 7      ob =
        getRemoteObject(objId);
 8      return ob;
   procedure checkState
   (curr)
 9  defThread:
10  if state == alive then
11      return;
12  else
13      onStateChange(curr, state);

   procedure onStateChange
   (curr, state)
14  defThread:
15  if state == abort then
16      restoreCheckpoint(curr);
17  else
18      return;
   procedure
   OpenNestedCommit (T)
19  DefThread:
20  mergeParent(T);
21  delegateCommit(T);
22  return;
   procedure asyncCommit
   (T)
23  AsyncThread:
24  OpenNestedCommit(T);
```

Algorithm 2 shows the methods required to support QR-OON. The methods execute either in the context of *defThread* or *asyncThread*, and are summarized as follows:

- *readObject*. This method is invoked for reading objects for a transaction. The parent's data-set is checked for the existence of the requested object. If it is not found locally, the object is fetched from the read quorum nodes. The state of *curr* is checked by invoking *checkState*.
- *OpenNestedCommit*. This method is responsible for the optimistic commit of a nested transaction. It merges the transaction's data-set with its parent's; delegates the work of commit to the *asyncThread*; resumes execution of *next*.
- *asyncCommit*. This method runs in the context of *asyncThread* and invokes the *OpenNestedCommit* method of QR-ON.
- *checkState*. This method is invoked in every read operation of a transaction. The method checks the status of the previous transaction and invokes *onStateChange* when the previous transaction either commits or aborts.
- *onStateChange*. This method is invoked whenever the final outcome of *curr* is decided. On commit, it simply returns to let *next* continue its execution. However, in case of abort of *curr*, *restoreCheckpoint* will restore the execution to the start of *curr* to retry it.

While *curr* is performing its commit asynchronously, only its subsequent transaction, i.e., *next*, can view *curr*'s locally committed changes. Other transactions cannot read these changes. Therefore, the correctness argument for QR-OON is exactly the same as that for QR-ON.

6. QR-ER: EARLY RELEASE

We now present the QR-ER protocol in which objects are dropped from the read-set of a transaction to reduce false conflicts, invalidations and communication costs. For data-structures, certain objects in the read-set of a transaction do not affect the final state of the data structure, and therefore need not be validated during the commit phase. This idea forms the basis of QR-ER. The approach has similarities to [15], but they address the problem in centralized setting.

In QR-ER, each object has a boolean field *validate*. If *validate* is set to *true*, then the object needs to be validated during commit; otherwise, the transaction can commit without validating that object. This field is set when the object copy is added to the read-set during the transaction's read operation. During the transaction's commit operation, objects with false *validate* flags are not validated, and the transaction commits without validating them.

In QR-ER, we retain objects so that subsequent operations can read them locally. In contrast, [15] releases objects from the read-set.

We build QR-ER on top of the QR-CN protocol [9], which extends the QR protocol with the closed nesting model. QR-ER treats nested transactions as in the closed-nesting model. In fact, it inherits all properties of closed nesting such that: a nested transaction's commit merges its data-set with its parent's data-set, and a nested transaction aborts without changing its parent's state. In contrast to open nesting, here the nested transaction's commits are local, while only the parent transaction commits globally. This eliminates the overhead of running a global commit phase for each (open) nested transaction. During the parent transaction's commit, only objects with true *validate* flags are validated.

Algorithm 3: Methods of QR-ER.

```
   procedure readObject
   (T, objId, validate)
1  ob=checkParent(objId);
2  if ob != null then
3      if validate then
4          ob.setValidate(true);
5  else
6      ob =
       getRemoteObject(objId);
7      ob.setValidate(validate);
8  T.addReadSet(ob);
9  return ob;
   procedure childCommit (T)
10 mergeParent(T);
   procedure parentCommit
   (T)
11 foreach ob in
   T.getReadSet() do
12     if ob.validate then
13         validationSet.add(ob);
14 T.commit(validationSet);
```

Algorithm 3 shows the methods required to support QR-ER, which are summarized as follows:

- *Read.* This method is invoked with the *validate* flag for requesting an object by the transaction. The object is looked up locally on the parent's data-set (line 1). If the object is not found, it is retrieved from the quorum nodes over the network, and the method argument *validate* flag is set on the object (lines 6-7). For locally found objects, the *validate* flag is set only when the method invalidates the object (lines 3-4).
- *childCommit.* For the commit operation, the nested transaction merges its data-set with the data-set of its parent. This behavior is similar to the closed nesting model (QR-CN).
- *parentCommit.* For the commit operation, the parent transaction creates the list of objects which need to be validated (lines 11-12), and sends them for validation in the commit message.

Theorem 6.1. *QR-ER's early release protocol guarantees linearizability of the Linked-List data structure.*

Proof. Consider the *add* operation of the linked-list. Let *add* insert the new element between *pred* and *curr* nodes of the linked-list. During linked-list traversal, the objects read are inserted into the read-set, while the objects *pred* and *curr* are inserted into the write-set. Linearizability of linked-list is guaranteed only when the following invariants are maintained for the *add* operation:

- *pred* is reachable from the head of the linked-list;
- *curr* is *pred.next*; and
- *curr.item* is in the linked-list.

These invariants are similar to those of the fine-grained lock implementation of the linked-list [16]. We now establish how QR-ER's early release protocol ensures these invariants during validation.

The scenarios in which the current transaction can be aborted include:

- If *pred* is not reachable from the head, this means that another transaction has deleted *pred*. In this case, *pred*'s version number will increase, aborting the current transaction.
- If *curr* is not *pred.next*, this means that another transaction must have added a new object in between. In this case, the version number of both *pred* and *curr* will increase, aborting the current transaction.
- If *curr* is not in the linked-list, then another transaction must have removed *curr*. This will also increase the version number of *pred*, aborting the current transaction.

It can be seen that the invariants only consider the objects that are in the write-set of the transaction, and are oblivious to the objects in its read-set. Thus, dropping the objects in the read-set will not affect correctness. □

Note that for *find* operation of linked-list, *pred* and *curr* are maintained in the read-set and validated during commit. The proof for other operations of Linked-List, Hashmap and Binary Search Tree are on similar lines, we omit them due to space constraints.

7. EXPERIMENTAL EVALUATION

We implemented QR-ON, QR-OON, and QR-ER on top of QR-DTM [31], the flat nesting implementation of quorum-based replication. We experimentally evaluated the protocols using three distributed data structures: Linked-List, Hashmap, and Binary Search Tree (BST); as well as the well-known TPC-C benchmark [1] implemented storing objects in a distributed hash table (as also used in other works such as [26]). For the benchmarks except TPC-C, each transaction consisted of multiple nested transactions, each of which enclosed a single operation on the data structure. In case of TPC-C, we split the original transaction profiles — e.g., *new order* or *delivery* — into nested transactions. We also included the original QR-DTM in our comparison. This way we can assess also the overhead of our proposals with respect to the classical, flat-nesting, protocol.

Our testbed consisted of 13 nodes in a private cluster, each of which is an 8-core AMD machine running Linux 10.04. We used 30 clients, with each client working on a single core.

QR-ON is particularly effective in scenarios where transactions are composed of a reasonable number of operations and with significant contention level. Our experiments therefore included 20% read-only transactions and with up to 7 nested calls per transaction.

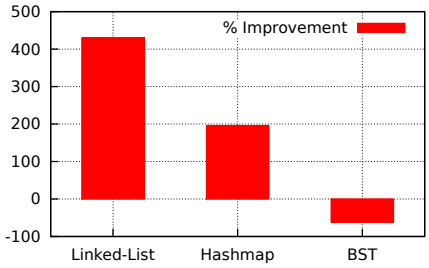
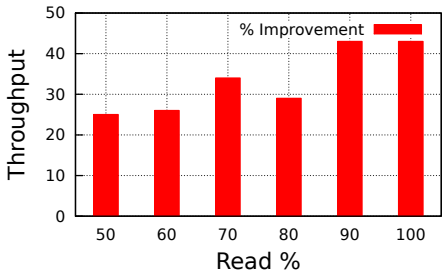


Figure 3: QR-ON vs QR-DTM.

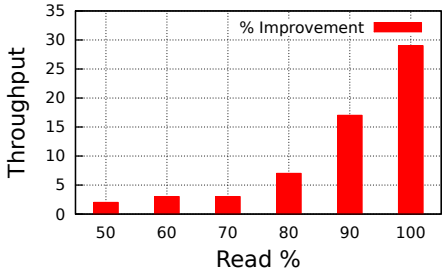
Figure 3 shows the average throughput improvement of QR-ON over QR-DTM. (We report the speed-up instead of absolute numbers because the throughput of Hashmap and BST are two orders of magnitude higher than that of Linked-List.) We observe maximum throughput improvement of 4.2× for Linked-List and 1.95× for Hashmap; a performance degradation of 62% for BST, over QR-DTM. The reason for the throughput improvement in Linked-List and Hashmap is directly due to reduced false conflicts. Avoiding false conflicts decreases the abort rate and the message size. In BST, clients access a small subset of shared objects, and therefore the benchmark does not suffer from false conflicts. As a consequence, QR-ON pays the overhead of committing (open)

nested transactions without actually reaping open nesting benefits.

The impact of QR-ON's overhead is minimal in execution scenarios that are prone to generate several false conflicts. When contention is reduced, the overhead, however, can become significant and degrade performance. QR-OON was indeed designed to overcome this.



(a) Linked-List (#calls per transaction=5, object count=500)



(b) Hashmap (#calls per transaction=10, object count=800)

Figure 4: Speed-up of QR-OON over QR-ON with increasing read %.

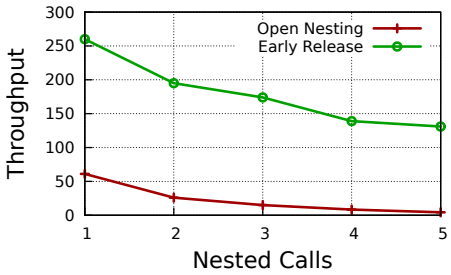
Figure 4 shows the speed-up of QR-OON over QR-ON for varying percentages of read-only transactions. We observe that QR-OON outperforms QR-ON by up to 43% for Linked-List and up to 29% for Hashmap. The improvement occurs because open-nested transactions speculatively read objects from the previous transaction, thus reducing the remote requests for those objects. Indeed, remote requests are reduced by as much as 76% for Linked-List and 25% for Hashmap.

We excluded BSTs in this comparison, because QR-OON does not provide sufficient improvement over QR-ON on BSTs. This is because BST is divided into multiple parts (the paths on the tree), and the probability of a subsequent transaction to access same objects (or part of) accessed by the previous committing transaction is very limited.

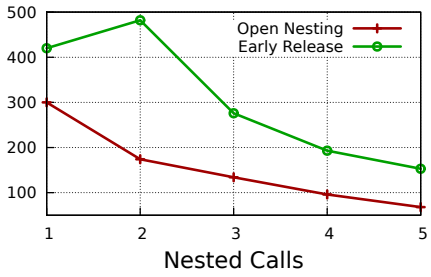
QR-OON is susceptible to workload changes and its performance improvement is bound by the amount of overlap between local execution and the commit phase. This is overcome in QR-ER, which commits nested transactions locally.

To understand QR-ER's effectiveness, we compared it against QR-ON for micro-benchmarks, by varying the number of objects in the data structure (Figure 5) and the number of nested transactions (Figure 6). The workload is configured with 50% of write transactions and we report the actual throughput.

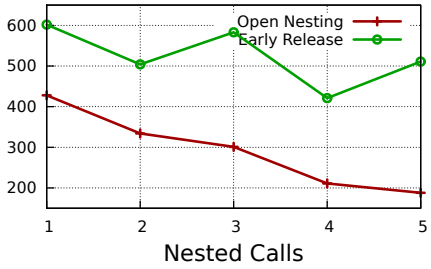
We observe that QR-ER outperforms QR-ON by as much



(a) Linked-List (objs=500)



(b) Hashmap (objs=400)



(c) BST (objs=1024)

Figure 5: QR-ER vs. QR-ON: throughput with increasing nested calls.

as 7× for Linked-List, 82% for Hashmap, and 1.1× for BST. This improvement is due to the reduction in the abort rate and the number and size of messages. Figure 7 shows the average of these parameters for each benchmark.

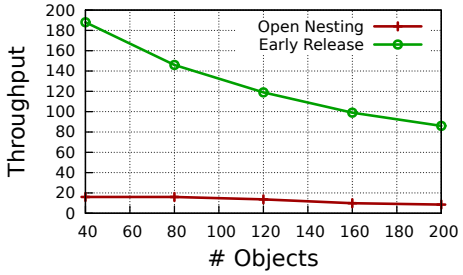
To understand the impact of message size on throughput, we compared QR-ER against QR-ON for 100% read-only workload (i.e., no abort) on all the benchmarks (we omit this plot due to space constraints). Here, the reduction in the message size is the sole factor that can contribute to any potential throughput improvement. We observed an average message size reduction of 8%, contributing to an average throughput improvement of 40%.

We also compared QR-ER against QR-ON for the TPC-C benchmark. We conducted this experiment on our private cluster (Figure 8(a)), as well as on the Future Grid public infrastructure[1] (Figure 8(b)). We report the throughput varying with the number of nodes. We observe that QR-ER outperforms QR-ON by 44% on average.

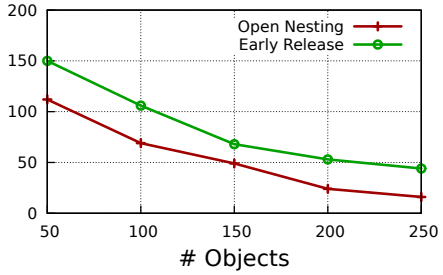
8. RELATED WORK

Replication has been studied in DTM for improving con-

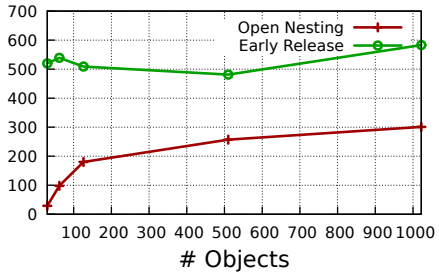
[1]http://www.futuregrid.org/



(a) Linked-List



(b) Hashmap



(c) BST

Figure 6: QR-ER vs. QR-ON: throughput with increasing object count (Calls=3).

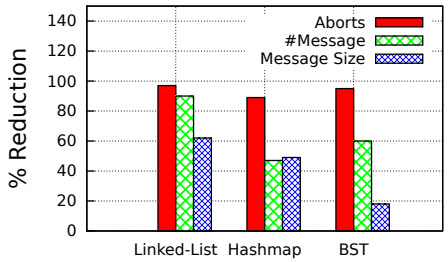
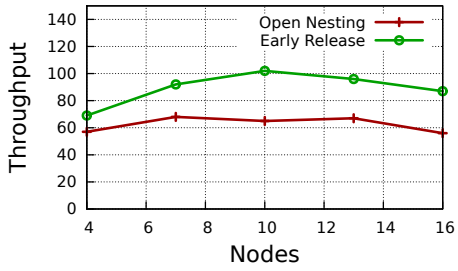
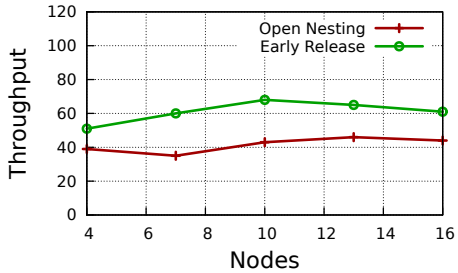


Figure 7: Abort rate and number/size of messages in QR-ER vs. QR-ON.



(a) Private cluster



(b) Public cluster

Figure 8: QR-ER vs. QR-ON: throughput with TPC-C.

currency and for coping with failures, largely in the context of cluster DTM [8, 7]. These works provide fault-tolerance by relying on broadcast primitives. D2STM [8], is a replicated DTM that provides strong consistency through a distributed certification scheme. GenRSTM[7] is a generic framework for replicated DTM, and supports replication via a replication manager, which is notified of updates made by local STMs.

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.* [13] 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 [2], 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 [23]. 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 [6], TCC [20], and LogTM [22] describe HTM implementations of closed and open nesting, with commit and abort handlers for open nesting. Agrawal *et al.* [4] 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.

None of the DTM efforts [8, 18, 17, 7] consider transactional nesting or checkpointing. The nested DTM works that we are aware of include the N-TFA protocol [29], which supports closed nesting, and the TFA-ON protocol [28], which supports open nesting. However, N-TFA and TFA-ON use a single copy DTM model and therefore are not fault-tolerant.

The problem of reducing false conflicts in TM has been addressed also in [10] and [11]. The former [10] allows the execution of independent chunks of DTM transactions at commit time after the lock acquisition. The latter [11] ap-

plies to general memory accesses without considering the semantics of the accessed data.

9. CONCLUSIONS

Transactional workloads, in particular, transactional data structures, can suffer from false conflicts. This phenomenon is exacerbated in DTM where each abort has a significant negative impact on total transaction execution time. Open nesting is the typical solution for solving false conflicts, but we determined that it has significant commit overhead in fault-tolerant DTM. We showed that optimistic open nesting can outperform open nesting in low contention scenarios. Additionally, we showed that early release can provide substantial performance improvement – up to an order of magnitude – over its open nesting counterparts. Along with abort rate reduction, we observed that reducing message size also helps to significantly improve throughput. This observation is very relevant for fault-tolerant DTM, where reduction in message size can reduce network latency to a large extent.

10. ACKNOWLEDGMENTS

This work is supported in part by US National Science Foundation under grants CNS-1217385 and CNS-1116190.

11. REFERENCES

[1] TPC-C benchmark: Transaction processing performance council. www.tpc.org.

[2] 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*, pages 26–37, 2006.

[3] D. Agrawal and A. El Abbadi. The tree quorum protocol: An efficient approach for managing replicated data. In *VLDB '90*.

[4] K. Agrawal, I.-T. A. Lee, and J. Sukha. Safe open-nested transactions through ownership. In *SPAA*, pages 110–112, 2008.

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

[6] 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*, pages 1–13, 2006.

[7] N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *NCA '11*, pages 271–274.

[8] M. Couceiro et al. D2STM: Dependable distributed software transactional memory. In *PRDC*, pages 307–313, 2009.

[9] A. Dhoke, B. Ravindran, and B. Zhang. On closed nesting and checkpointing in fault-tolerant distributed transactional memory. In *IPDPS*, pages 41–52, 2013.

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

[11] N. L. Diegues and P. Romano. Time-warp: lightweight abort minimization in transactional memory. In *ACM SIGPLAN PPoPP '14*, pages 167–178, 2014.

[12] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.

[13] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP*, pages 48–60, 2005.

[14] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPoPP '08*, pages 207–216.

[15] M. Herlihy, V. Luchangco, M. Moir, and W. N. S. III. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, 2003.

[16] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann, 2012.

[17] S. Hirve, R. Palmieri, and B. Ravindran. Archie: A Speculative Replicated Transactional System. In *ACM/IFIP/USENIX Middleware*, 2014.

[18] S. Hirve, R. Palmieri, and B. Ravindran. Hipertm: High performance, fault-tolerant transactional memory. In *ICDCN*, pages 181–196, 2014.

[19] F. Marchioni and M. Surtani. *Infinispan Data Grid Platform*. Packt Pub., 2012.

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

[21] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS*, pages 359–370, 2006.

[22] M. J. Moravan et al. Supporting nested transactional memory in logTM. In *ASPLOS*, pages 359–370, 2006.

[23] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI '06*.

[24] J. E. B. Moss and A. L. Hosking. Nested tm: Model and architecture sketches. *Sci Comp Prog*, 63(2):186–201, 2006.

[25] Y. Ni, V. Menon, A. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *ACM PPOPP*, pages 68–78, 2007.

[26] S. Peluso, P. Romano, and F. Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *ACM/IFIP/USENIX Middleware*, pages 456–475, 2012.

[27] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, 1995.

[28] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *SYSTOR*, page 12, 2012.

[29] A. Turcu, B. Ravindran, and M. Saad. On closed nesting in distributed transactional memory. In *TRANSACT*, 2012.

[30] W. E. Weihl. Commutativity-based concurrency control for abstract data types. *IEEE Trans. Computers*, 37(12):1488–1505, 1988.

[31] B. Zhang and B. Ravindran. A quorum-based replication framework for distributed software transactional memory. *OPODIS*, pages 18–33, 2011.