On Partial Aborts and Reducing Validation Costs in Fault-tolerant Distributed Transactional Memory

Aditya Dhoke

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Science and Application

Binoy Ravindran, Co-Chair
Eli Tilevich, Co-Chair
Wu-chun Feng

Sep 4, 2013
Blacksburg, Virginia

Keywords: Distributed Transactional Memory, Closed Nesting, Open Nesting, Automated Nesting
Copyright 2013, Aditya Dhoke

On Partial Aborts and Reducing Validation Costs in Fault-tolerant Distributed Transactional Memory

Aditya Dhoke

(ABSTRACT)

Distributed Transactional Memory (DTM) is an emerging synchronization abstraction that promises to alleviate the scalability, programmability, and composability challenges of lock-based distributed synchronization. With DTM, programmers organize code that read/write shared memory objects, both local and remote, as *memory transactions*. An underlying DTM framework guarantees atomicity, isolation, and consistency properties for those transactions through speculative concurrency control.

In DTM, restarting an aborted transaction from the beginning can degrade performance as transactional conflicts may have occurred in the later part of the transaction, wasting work. The *partial abort* method alleviates this difficulty by enabling a transaction to be rolled back to the point where objects in the transaction's read-set and write-set are still consistent.

In this thesis, we present protocols for supporting partial aborts in QR-DTM, a fault-tolerant DTM that uses quorum protocols for distributed concurrency control in the presence of failures. We focus on two partial abort models: closed nesting, which allows transactions to be nested and inner transactions to be rolled back without rolling back outer transactions; and checkpointing, which allows the transaction state to be saved in checkpoints throughout execution and transactions to be rolled back to those checkpoints. We present protocols that support these partial abort models in QR-DTM, called QR-CN and QR-CHK. We implemented these protocols and conducted experimental studies using macro-benchmarks (e.g., distributed versions of STAMP benchmark), and micro-benchmarks (e.g., distributed data structures). Our studies reveal that QR-CN improves throughput by as much as 101% over flat nesting in specific cases, with an average improvement of 53%.

We also develop QR-ACN, a framework that automatically decomposes programmer-written transactions into closed nested transactions, at run-time, to improve performance. The composition of a closed nested transaction depends on the contention of objects, which can change at run-time depending upon the workload at hand. Our implementation and experimental studies reveal that QR-ACN consistently outperforms flat nesting by an average of 51% on benchmarks including TPC-C.

False conflicts occur when high-level operations, even though semantically independent, traverse the same set of objects during transaction execution. Such conflicts can lead to repeated aborts, increasing transaction execution time and degrading performance, which can be significant in DTM, since transaction execution also includes network communication.

We consider the approach of *reducing validation cost* for resolving false conflicts. We present three protocols for reducing validation cost in DTM. Our first protocol, QR-ON, incorporates the open nesting model into QR-DTM. Open nesting allows inner-nested transactions to commit independently of their parent transaction, releasing objects in the transaction read-set and write-set early, minimizing aborts due to false conflicts. We then present 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 present an early release methodology, QR-ER, in which objects that do not affect the final state of the shared data are dropped from the 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× on micro- and macro-benchmarks.

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385. Any opinions, findings, and conclusions or recommendations expressed in this site are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

Dedication

I dedicate this thesis to my late brother, Apoorva Dhoke and my parents.

Without their love and support this would not have been possible

Acknowledgments

I would like to thank my advisor, Dr. Binoy Ravindran, for his help and guidance on both technical and personal aspects. I am thankful to him for his faith in me.

I would also like to thank Dr. Eli Tilevich and Dr. Wu-chun Feng, for serving on my committee. In addition, I would like to thank all of my colleagues at the Systems Software Research lab. I would particularly like to thank Alex Turcu, Bharat Reddy, Roberto Palmieri, Sachin Hirve and Sudhanshu Mishra for their support and encouragement. I have learnt a lot while working on Distributed Transactional Memory and from the people I have worked with.

Contents

1 Introduction **1**

1.1 Partial Transactional Abort 3

1.2 Reducing Transactional Validation Costs 6

1.3 Summary of Thesis Contributions 9

1.4 Thesis Organization 9

2 Related Work **11**

3 Quorum-Based Replication **14**

3.1 Overview 14

3.2 System Model 15

3.3 System Design 16

3.3.1 Components. 16

3.3.2 Network Layer. 17

3.3.3 Feature Implementation 19

3.3.4 Programming Framework. 20

4 Partial Transactional Abort **23**

4.1 QR-CN Protocol 24

4.1.1 QR-CN: Overview 24

4.1.2 QR-CN : Read/Write Operation 25

4.1.3 QR-CN : Commit Operation. 27

4.1.4 Analysis 27
4.2 Checkpointing 29
4.2.1 QR-CHK: Overview 29
4.2.2 QR-CHK: Read/Write Operation 30
4.2.3 Analysis 30
4.3 Automated Closed Nesting 31
4.3.1 Overview 31
4.3.2 System Design 33
4.3.3 Algorithm 35
4.3.4 Contention Level 36
4.3.5 Algorithm Module 36

5 Experimental Evaluation for Partial Abort 38

5.1 Evaluation of Closed Nesting and Checkpointing 38
5.2 Comparison with DTM Implementations 42
5.3 Throughput under Node Failures 44
5.4 Evaluation of Automated Closed Nesting 44

6 Validation Cost Reduction 47

6.1 Open Nesting 47
6.2 Optimistic Open Nesting 51
6.3 Early Release 52

7 Experimental Evaluation for Validation Cost Reduction 55

7.1 Evaluation of Open Nesting 56
7.2 Evaluation of Optimistic Open Nesting 57
7.3 Evaluation of Early Release 57

8 Conclusion and Future Work 60

8.1 Conclusion 60

8.2 Future Work 61

Bibliography **62**

List of Figures

1.1 An Example Flat Nested Transaction. 4

1.2 An Example Closed Nested Transaction. 4

3.1 Ternary tree with 13 nodes. 15

3.2 Client Server Model. 16

3.3 Design Diagram. 18

3.4 Example of Bank Transaction in QR-DTM. 20

3.5 Example of closed nested Bank Transaction in QR-DTM. 21

3.6 Example of open nested List Transaction in QR-DTM. 22

4.1 Flat Nested Bank Transaction. 32

4.2 Flat Nested Bank Transaction. 33

4.3 Code Arrangement using UnitBlocks and Blocks. 34

4.4 Block Diagram. 35

5.1 Flat nesting, closed nesting and checkpointing for different read workload. . . 39

5.2 Flat nesting, closed nesting and checkpointing for different transaction length. 40

5.3 Flat nesting, closed nesting and checkpointing for different number of objects. 41

5.4 Throughput of QR-STM, HyFlow, and Decent-STM for the bank application 43

5.5 Throughput under increasing node failures. 44

5.6 100% NewOrder Transactions. 45

5.7 100% Payment Transactions. 45

5.8 50% Payment and New Order Transactions. 46

5.9 Vacation Benchmark. 46

6.1 Concurrent transactions in the open nesting model. 48

7.1 QR-ON vs QR-DTM. 55

7.2 Speed-up of QR-OON and QR-ER over QR-ON with increasing read %. . . . 56

7.3 Abort rate and number/size of messages in QR-ER vs. QR-ON. 57

7.4 QR-ER vs. QR-ON: throughput with increasing nested calls. 58

7.5 QR-ER vs. QR-ON: throughput with increasing object count (Calls=3). . . 59

7.6 QR-ER vs. QR-ON: throughput with TPC-C. 59

List of Tables

5.1 Abort rate and message % for QR-CN and QR-CHK compared to flat nesting 42

Chapter 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 [34], sequential code must be minimized as much as possible. Coarse-grained locking, though simple to program, increases sequential execution time. Fine-grained locking (and lock-free synchronization) reduce sequential execution but suffer from poor programmability. Additionally, lock-based synchronization suffers from scalability and composability challenges [26]. 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) [40] has recently emerged as an alternative concurrency control abstraction that promises to alleviate these difficulties. TM is inspired by database transactions [24], which has been a phenomenally successful abstraction for database systems. With TM, code that access shared memory objects are organized as *memory transactions*, which speculatively execute, while logging changes made to objects in memory logs. Transactions are monitored for read/write and write/write conflicts, usually by keeping track of objects that they read from (in read-sets) and write to (in write-sets). When two transactions conflict (detected using read-sets and write-sets), one of them is aborted, and the other's object updates are made permanent (i.e., committed). Aborted transactions are re-started, after rolling-back their object changes using the saved logs. The approach thus yields (the illusion of) transactional atomicity (all or none), memory consistency (linearizable execution), and transactional isolation (local changes are hidden till commit), with a simple programming interface that is devoid of locks (i.e., high programmability).

TM was originally proposed in hardware (called HTM [30]), later in software (called STM [66]), and subsequently in hardware/software combination (called HybridTM [41]). HTM has the lowest overhead, but hardware must support TM and transactions are limited in space and

time. STM does not have such limitations, but has higher overhead.

TM is increasingly gaining traction: Intel has released a C++ compiler with STM support [35]. Oracle [16, 70], AMD [2, 18], and Intel [36] have released experimental or commodity hardware with HTM support. GCC has released language extensions to support STM [3].

Similar to STM, distributed STM (or DTM) [15, 12] is also emerging as a promising alternative to lock-based distributed concurrency control. DTM can be supported in any distributed execution model, including a) control flow [9], where objects are immobile and transactions invoke object operations through RMIs/RPCs; b) dataflow [68], where transactions are immobile, and objects are migrated to invoking transactions; and c) a hybrid model [13], 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 (or cc DTM) [33, 79, 57] and cluster DTM [19, 54, 55]. While both use message-passing links over a communication network for node-to-node communication, they differ in the underlying communication cost model. 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 typically uses a cache-coherence protocol (e.g., Ballistic [33], Relay [78]) to locate and move objects in the network, satisfying object consistency properties. In this thesis, we focus on cc DTM.

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 [13, 39, 19, 51, 15, 12] – 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) [13, 39, 19, 55, 15, 12]. 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.

The use of STM has broadened the nature of transactional operations, which are no longer seen only as a way for exclusively manipulating application data in a consistent manner, but also as a way to synchronize worker threads within any software layer. Additionally, they have induced a shift in terms of the transactional properties that must be guaranteed and how those guarantees should be conveniently supported. Historically, transactional data manipulation supports were required to enforce the so called ACID (Atomicity, Consistency, Isolation, Durability) properties. On the other hand, durability via the logging of transaction updates on stable storage systems, like in traditional DBMS architectures, may be sub-optimal in STM-based environments; durability could instead be efficiently achieved using replication strategies that exclusively operate in main memory.

Replication strategies can be classified based on the number of copies available for each shared object. One is called *partial replication* [52, 53, 63], in which each object is located at more than one node, but fewer than all nodes in the system. The second is called *full replication* [65], in which each node has all the shared objects. Even though the latter model guarantees greater resilience in the presence of faults, it needs costly atomic broadcast protocols that are inherently non-scalable in terms of messages transmitted for establishing agreement. This limitation can be overcome using quorums that minimize the number of contacted nodes – e.g., [5, 17]'s quorum-based replication (or QR) protocol.

1.1 Partial Transactional Abort

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 approach ensures that the transaction is able to access the latest copy of objects the next time. However, it can be conservative in many cases, and can result in doing redundant work, degrading performance – e.g., the transaction may have suffered a conflict in its later part, while the earlier part was conflict-free, but the entire transaction is nevertheless rolled back and re-executed upon a conflict.

Partial abort is a mechanism that enables a transaction to partially rollback its changes and restart from a point where it has consistent and valid view of the shared data. This enables rolling back (potentially) only the conflicting part of the transaction, improving performance, which can be significant in DTM, where transaction execution involves remote object accesses and thus network communication. Partial aborts therefore help improve performance in DTM particularly by reducing remote accesses.

Closed nesting [49] and checkpointing [37] are two different mechanisms to partially abort or rollback a transaction. In the closed nesting model, a transaction can enclose sub-transactions inside it, which can be aborted and rolled back independently of the outer transaction. In contrast, in the checkpointing model, the transaction execution state is saved throughout the transaction execution using special API calls, enabling the transaction to be rolled back to a previous execution state.

We show the benefit of partial abort by illustrating an example of closed nesting. Figure 1.1 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 before T_{flat} attempts to commit its changes, a conflicting transaction, say T_c, has successfully modified m_3. This will cause T_{flat} to abort, and start again from the

```
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 1.1: An Example Flat Nested Transaction.

```
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 1.2: An Example Closed Nested Transaction.

beginning (label *T_flat* in Figure 1.1). In the next attempt, T_{flat} again reads m_1 and m_2, though they were unchanged, incurring additional remote calls.

Figure 1.2 shows a closed nested transaction, T_{closed}, enclosed inside a parent transaction, T_{parent}. Here, T_{parent} adds m_1 and m_2, while T_{closed} adds the intermediate matrix m_3. Similar to the previous example, assume that before 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 restarts from label *T_closed* in Figure 1.2. In the subsequent attempt, T_{closed} will only read m_3. After T_{closed} commits, T_{parent} will attempt to commit its 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 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 a 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.

In the example in Figure 1.2, 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:

- What application workload will benefit from partial abort, as compared to flat nesting?
- What is the potential performance improvement or degradation due to partial abort?
- Which parameters of a transaction will affect partial abort's performance (improvement or degradation)?
- How should the transaction code be transformed to obtain maximum benefits from partial abort?

In this thesis, we answer these questions by developing DTM protocols in the following partial abort models:

- *Closed Nesting.* In this model, a transaction can consist of multiple inner or closed nested transactions. Each inner transaction attempts to commit locally, but the commit is not visible outside the enclosing transaction. Inner transactions can abort independently of their parent, without altering the state of the outer transaction (i.e., partial rollback).
- *Checkpointing.* In this model, transactions create checkpoints by saving transactional execution state. Checkpoints provide a way to resume from a consistent transaction state. In case of conflict, a transaction can partially rollback to a checkpoint to resolve conflict and resume execution. Checkpointing can be thought of as a generalization of closed nesting, where a transaction can rollback to any of the checkpoints, whereas, in closed nesting, it is limited to one of the enclosing transactions.

To develop partial abort protocols in these models, we use Zhang and Ravindran's quorum-based DTM concurrency control protocol called QR [80] as the baseline DTM. In QR, a *quorum system* [5] is used to manage replication and concurrency control of shared objects. 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 [11].

We develop protocols for supporting closed nesting and checkpointing in QR, called, *QR-CN* and *QR-CHK*, respectively. We also improve QR's commit overhead through an incremental

validation technique called, *Read Quorum Validation* or *Rqv*. We show that Rqv also ensures 1-copy equivalence.

We construct a Java implementation of QR-based DTM, called QR-DTM, and implement our proposed protocols. We conduct experimental studies using macro-benchmarks including distributed versions of applications from the STAMP benchmark suite and micro-benchmarks including distributed data structures. 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.

We also develop QR-ACN, a framework that automatically decomposes programmer-written transactions into closed nested transactions, at run-time, to improve performance. The composition of a closed nested transaction depends on the contention of objects, which can change at run-time depending upon the workload at hand. QR-ACN decomposes transactions into closed-nested transactions depending on the workload and contention of objects. We implement QR-ACN and conduct experimental studies using benchmarks including TPC-C [1] and Vacation [44]. Our results reveal that QR-ACN provides an average improvement of 51%, with highest improvement of 112% for Vacation benchmark.

1.2 Reducing Transactional Validation Costs

In DTM, validation of objects in a transaction's read-set and write-set is performed during its execution and commit operation. Successful validation means that the objects' versions have not changed since they were read by the transaction i.e., the objects are still valid. Unsuccessful validation means that an object version has increased and therefore the transaction must be aborted and retried.

Long running transactions have higher chance of unsuccessful validations because of large size of their read-sets and write-sets that need to be validated. With several long running transactions executing simultaneously, the chances of conflict among them further increases. Abort of such transactions would mean redoing the entire execution from the beginning, which can degrade performance. This can be significant in DTM, where the read or commit operation involves network communication.

To mitigate such performance degradation, the cost of validation must be reduced by reducing the objects that need to be validated. This will minimize the chance of unsuccessful validations. There are two general approaches for reducing validation costs:

- *Approach A.* Transactions can globally commit sub-transactions or inner transactions to expose their partial changes [49]. Once the changes are committed, objects corresponding to inner transactions can be removed from the read-set and write-set. The outer transaction can then resume further execution. Thus, at any point during exe-

cution, transaction has to validate lesser number of objects, thus, reducing the cost of validations performed during execution and commit operation.

- *Approach B.* Transactions can selectively drop objects from their read-set based on the knowledge that changes to these objects do not affect transaction semantics [67, 69]. This approach effectively reduces the objects accessed by a transaction, and those objects no longer participate in the validation operation. This reduces the cost of validation for the transaction.

A conflict occurs among transactions when at least one of them is writing to the same object. This can give rise to read/write or write/write conflict. One of the transactions must be aborted to resolve the conflict. In addition, STM systems also suffer from false conflicts [50, 48], which occur among transactions performing seemingly independent operations.

False conflict occurs among transactions involving semantically independent high-level operations that traverse the same set of objects during transaction execution. We illustrate this with an example of a set implemented using a sorted list. Insertion/deletion of an element in the set can be viewed as a high-level operation, whereas insertion of an object in the sorted list can be viewed as a low-level operation. Consider two transactions T_1 and T_2, each attempting to insert objects O_1 and O_2 with distinct values into the set. Both T_1 and T_2 will traverse from the head of the list, and continue traversal until they find an object with a value greater than their input object. Once the target object is discovered, they will perform a write operation on the two nearby objects for inserting the new object. From a low-level perspective, if T_1 and T_2 are executing simultaneously, they will conflict since there is an intersection between T_1's write-set and T_2's read-set or vice-versa. From a high-level perspective, the operations of T_1 and T_2 are semantically independent as the input elements are different, and the transactions merely traverse through the objects to reach the target objects to perform their operation. Conflicts among such independent transactions are called false conflicts. Note that the conflict depends on the choice of implementation of the set.

The problem of false conflicts has been studied in centralized settings (i.e., multicore/multiprocessors) [50, 69, 27]. However, transaction execution characteristics in distributed settings are significantly different from multiprocessor settings. Unlike multiprocessors, 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 communication and synchronization with remote nodes. False conflicts can therefore degrade performance, which can be significant in fault-tolerant DTM, where repeated aborts can significantly increase transaction execution time and network resource consumption.

We propose to mitigate false conflicts by reducing the validation cost. This raises several interesting questions:

- What is the performance improvement that can be obtained by reducing the validation cost?

- What validation cost reduction approach has the least performance degradation with increasing number of operations or inner transactions within a transaction?
- What applications are most suited for what validation cost reduction approaches?

The thesis answers these questions by developing validation cost reduction protocols for reducing false conflicts in QR-based DTM. We consider both the previously mentioned approaches. In the first approach, we globally commit the semantically independent high level operations, thus, releasing the read-set and write-set objects. Here, we use the independence of the operations to increase concurrency among the transactions. In the second approach, certain objects are dropped from the read-set, thus, from that point onwards, they will not be validated. Here, we use the fact that the high-level operation traverses through objects to reach the target object, and, any invalidations on those dropped objects do not affect the correctness of the transaction.

We present three protocols that are based on these two approaches: QR-ON, QR-OON, and QR-ER.

QR-ON incorporates the open nesting model [49] into QR-DTM. In the open nesting model, only the objects accessed within the (open) nested transactions are validated and (globally) released after successful commit. Once the inner transactions commit, the objects are no longer accessed by the inner or outer transactions. This early memory release increases the potential for improving concurrency: two parent transactions that have read from or written to the same set of objects in their inner transactions will not detect any conflict during validation at commit time. Since each nested transaction directly commits to the shared memory, other transactions can also immediately access the just committed data.

QR-OON improves upon QR-ON. The commit phase is costly in fault-tolerant DTM. Hence, the open nesting model does not scale when there are many open-nested transactions. QR-OON is based on this observation. In QR-OON, the commit phase is 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 and improving throughput.

QR-ER embodies an early release methodology. Even though in QR-OON, the expensive cost of commit is alleviated by an asynchronous implementation, fault-tolerant DTM protocols such as QR-DTM still extensively use the network during transaction execution. To further reduce false conflicts and improve performance, QR-ER releases objects from the read-set that do not need to be validated because, even in case of invalidation, they do not compromise correctness of transaction execution. This approach is suited for transactional data structures [27] and for protocols that require a significant amount of network communication for execution and commit operation. With early release, each transaction locally

decides if it wants to exclude objects from its read-set, according to the semantics of the data structure used, thereby saving additional messages and improving throughput.

We implemented the protocols and conducted experimental studies using micro benchmarks (i.e., data structures such as Linked List, Hashmap, Binary Search Tree) and a version of the TPC-C benchmark [1] implemented using a DHT [52, 53]. 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×.

1.3 Summary of Thesis Contributions

The thesis contributions are summarized as follows:

- We present two protocols for supporting partial aborts in fault-tolerant DTM called QR-CN and QR-CHK. While QR-CN uses the closed nesting partial abort model, QR-CHK uses the checkpointing model for partial aborts. Our implementation and experimental evaluation of these protocols show their improvement over state-of-the-art i.e., flat nesting in fault-tolerant DTM: QR-CN improves throughput by as much as 101%.
- We present a framework called QR-ACN that automatically decomposes programmer-written transactions into closed nested transactions, at run-time, to improve performance. QR-ACN decomposes transactions into closed-nested transactions depending on the workload and object contention. Our implementation and evaluation reveal that QR-ACN improves flat nesting by 51% (on average).
- We present three protocols for avoiding false conflicts in fault-tolerant DTM called QR-ON, QR-OON, and QR-ER. While QR-ON incorporates the open nesting model into QR-DTM, QR-OON commits open-nested transactions asynchronously, and QR-ER releases objects that do not affect the final state of the shared data from the transaction's read-set. All three techniques are shown to be highly effective through implementation and experimental studies: QR-OON outperforms QR-ON by up to 43%, and QR-ER outperforms QR-ON and QR-OON by up to 10×.

1.4 Thesis Organization

The rest of the thesis is organized as follows.

We overview past and related work in Chapter 2. Since our work is based on the QR protocol for DTM concurrency control [80], we summarize this protocol in Chapter 3 for completeness.

We describe the partial abort protocols QR-CN and QR-CHK in Chapter 4. We also introduce the Read Quorum Validation protocol, Rqv, which improves QR by incremental validation. In this chapter, we describe QR-ACN, an automated closed nesting framework. Chapter 5 reports our experimental evaluation of QR-CN and QR-CHK protocols, as well as the QR-ACN framework.

The validation cost reduction protocols QR-ON, QR-OON, and QR-ER are described in Chapter 6. We report our experimental evaluation for these protocols in Chapter 7.

The thesis concludes and outlines potential future research directions in Chapter 8.

Chapter 2

Related Work

Locking is one of the most widely used mechanisms for concurrency control. Locks provide mutual exclusion, thus, guaranteeing that only one thread can enter the critical section. Coarse-grained locking locks all the shared objects, thus, enforcing sequential execution. On the other hand, fine-grained locking provides much higher parallelism as it acquires locks only on selected objects [8, 71]. Non-blocking mechanisms using atomic read-modify-write primitives are much more performant than fine-grained locking [10, 77, 76]. Unfortunately, all these mechanisms are prone to deadlock, priority inversion and starvation. Moreover, methods implemented using locking and non-blocking mechanisms are non-composable. Efforts have been made to automate fine-grained locking [23], however, it has limited applicability.

STM is an alternative to lock-based concurrency control mechanisms. STM systems provide atomicity, isolation and consistency. They support composability i.e. the properties guaranteed by a transaction are also guaranteed by combination of two transactions. STM has been implemented in centralized settings [61, 22, 62, 28].

DTM was first proposed by Herlihy and Sun [32] as an alternative to distributed locking mechanisms. They introduced Ballistic cache coherence protocol in which clusters are arranged hierarchically and the leaders of the clusters communicate to track and update object copies. The dynamic analysis of Relay cache coherence protocol by Zhang and Ravindran [78] shows that Relay is appropriate for DTM and that contention manager has an impact on the performance of DTM. They also introduce a location aware protocol which takes advantage of proximity of the nodes. The Transaction Forwarding Algorithm [58] observes the changes in version numbers of objects while transaction execution, and perform early validation to detect conflict earlier in the transaction. Objects can be migrated to specific nodes based on access pattern of transactions to maximize local execution, thereby increasing performance. This protocol has been implemented in Java(HyFlow [59]), Scala(Hyflow2 [72]) and C++(HyflowCPP [45].

The aforementioned DTMs maintain single copy of the object, hence, cannot handle node

failure. Replication has been used in the context of cluster for coping with failures [13, 39, 19, 51, 15, 12]. These works provide fault-tolerance properties by inheriting fault-tolerance protocols from database replication schemes, which rely on broadcast primitives. D2STM [19], is a replicated cluster STM that provides snapshot isolation through a distributed certification scheme. Read-only transactions can execute locally without incurring any aborts as D2STM provides full-replication. It uses Bloom Filter Scheme (BFC) for transaction involving writes to atomically broadcast the read-set and write-set in order to reduce the message size. This scheme helps to reduce the broadcast latency, thus, improving performance by up to 37 % compared to naive distributed certification scheme. DiSTM [38] implements three cache coherence protocol in a cluster setting with master nodes having multiple slave nodes. They compare the protocols for different benchmarks and infer that applications which have transactions with large number of remote calls should use centralized protocols to avoid high network overhead. On the other hand, transactions which spend most of the time in local execution will benefit maximum from decentralized protocol. Decent STM [12] implements a decentralized snapshot isolation protocol for guaranteeing consistency. A list of committed versions is maintained for the executing transaction to choose the object version on which it is dependent upon. The use of randomized consensus helps to decentralize the concurrency control. They observed that although the scheme scales with increasing application threads, it also suffers from overhead due to maintainence of versioned objects. GenRSTM [15] is a generic framework for replicated DTM, and supports replication via a replication manager, which is notified of updates made by local STMs. This work is aimed to provide a common platform for development and testing for DTM applications.

In contrast to the earlier works, Zhang and Ravindran's QR protocol [80] implement a replicated DTM model, based on metric-space network, that relies on quorums for managing transactional metadata, and ensures consistency using multicast among the replicas. (QR is the foundation of our work.). Granola [20] executes most of the transactions locally based on its data partitioning strategy. Distributed transactions can be executed as co-ordinated transactions using two-phase commit or as independent transactions by breaking original transaction into multiple independent operations. Sinfonia [7] uses the concept of mini-transaction where instead of sending individual requests, the entire transaction is sent to the memory node. Additionally, Sinfonia provides fault-tolerance and high availability.

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.* [25] 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 [4], 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 [47]. 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 [14], TCC [43], and LogTM [46] describe HTM implementations of closed and open nesting, with commit and abort handlers for open nesting. Agrawal *et al.* [6] 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 [37] 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. However, checkpointing needs to save and restore the processor context to resolve conflicts. Therefore, this approach may not be suited for all platforms. Rainbow OS [64] propose checkpointing to include entire view of the DTM system which can be used in case of node failure. Note, this is different from the checkpoints for transaction mentioned in this paper.

None of the DTM efforts [13, 42, 19, 54, 55, 57, 15, 12] consider transactional nesting or checkpointing. The nested DTM works that we are aware of include the N-TFA protocol [74], which supports closed nesting, and the TFA-ON protocol [73], 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 [74] reports 2% average performance benefit for closed nesting compared to flat nesting (and 84% speedup in certain cases). In TFA-ON [73], abstract locks are used to guarantee that no data conflicts occur. The average speedup for open nesting is 30% compared to flat nesting [73]. 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), checkpointing (QR-CHK) and open nesting (QR-ON) in replicated (and thus fault-tolerant) DTM, and is the first work to do so.

Chapter 3

Quorum-Based Replication

3.1 Overview

Zhang and Ravindran's quorum-based replication protocol [80] (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 [11], 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 [5]. 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.

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

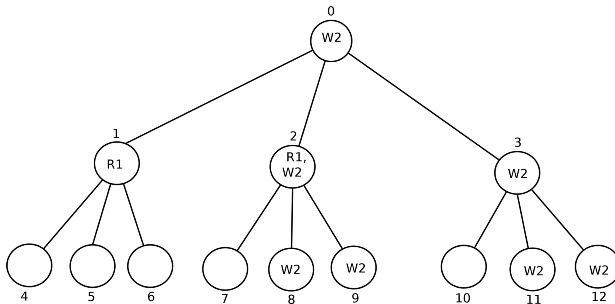


Figure 3.1: Ternary tree with 13 nodes.

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 *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* [5] 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 3.1 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.

3.2 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.

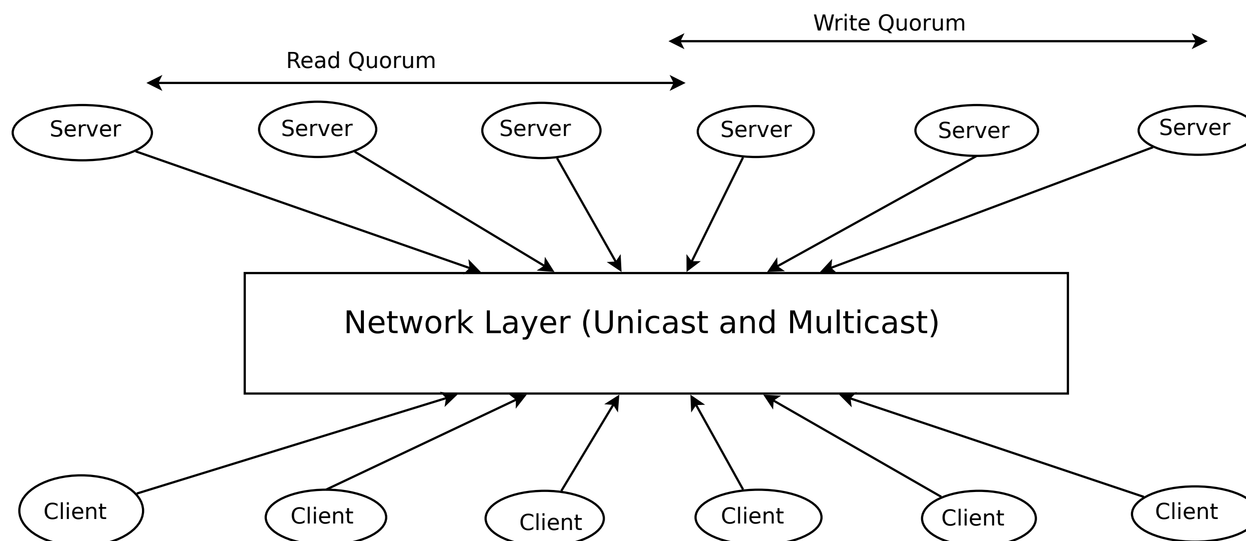


Figure 3.2: Client Server Model.

3.3 System Design

We have implemented QR protocol in QR-DTM as a Java based DTM. QR-DTM uses the QR protocol [80] as the basic algorithm for providing replication and cache coherency. During system initialization, objects are created on the quorum nodes. The nodes are provided with an unique ID which are used to create an overlay logical ternery tree. This tree forms the basis for the creation of read and write quorums for each of the nodes. On tree formation, the quorum system is ready for servicing transactions. QR-DTM follows *client-server* architecture, the quorum nodes form the server while the client nodes execute transactions, as shown in Figure 3.2. The clients send read/write and commit requests for executing the transaction. The requests are processed synchronously i.e. the client sends request, and waits till it recieves response from the quorum nodes.

Figure 3.3 shows the architecture diagram for QR-DTM. Both the clients and quorum nodes have a dedicated process, there are modules within this process each with different responsibility.

3.3.1 Components.

A client node consists of the following components/modules:

- *Context.* There is a *Context* object associated with every transaction (both parent and child). It maintains transactional metadata like *transaction ID*, *read-set*, *write-set*, *create-set*, *transaction priority* and data specific to the nesting models.

- *ClusterManager.* This module keeps track of the read and quorum designated to that client node.
- *ClientProxy.* This module acts as an interface for transactions to service their requests. Every request is first processed by this module. Depending on the type of request, pre-processing is performed and message that is to be sent by *ClientNetworkLayer* is generated. Once the request recieves responses, they are processed again to obtain final result which can be used by the transaction for furthur execution.
- *ClientNetworkLayer.* This module is used for multicast and unicast communication between client node and quorum nodes. *ClientProxy* delegates every request to this module. *ClientNetworkLayer* sends messages to quorum nodes (multicast) and waits for the expected number of replies, which is read from the *ClusterManager*. Once all the replies are collected they are sent back to *ClientProxy*.

A quorum node consists of the following:

- *ServerNetworkLayer.* This layer waits for messages from client, delegates the processing to *ServiceThreadPool* and sends back the response to the sender.
- *ServiceThreadPool.* This consists of pool of threads to process the incoming requests. The request is first recieved by *Monitor* thread which delegates the processing task to one of the service threads in the pool in round-robin manner. The service thread invokes appropriate function based on the type of message. Every message type inherits *Message* class and implements *process* method which is invoked to process the request. A response message is created which is then returned to the *ServerNetworkLayer* for replying back.
- *ResourcePool.* This consists of the objects shared across the system. This module is responsible for changing the state of objects for locking and updating the version number. The operations on PW and PR lists are also performed by this module.

The quorum node maintains the shared objects while client nodes maintain their copies. Every object,*QObject*, has fields like *version number* to maintain the version number of object, *protected* to inform if the object is in locked state and *owner* to note the transaction which has set the protected or locked state on the object. The state of shared objects can change when *process* method is invoked. The client node send their consistent version of object copy to the quorum nodes for validation and applying commit changes.

3.3.2 Network Layer.

Network Layer (*ServerNetworkLayer* and *ClientNetworkLayer* combined) is responsible for reliable unicast and multicast messaging between the client nodes and quorum nodes. Based

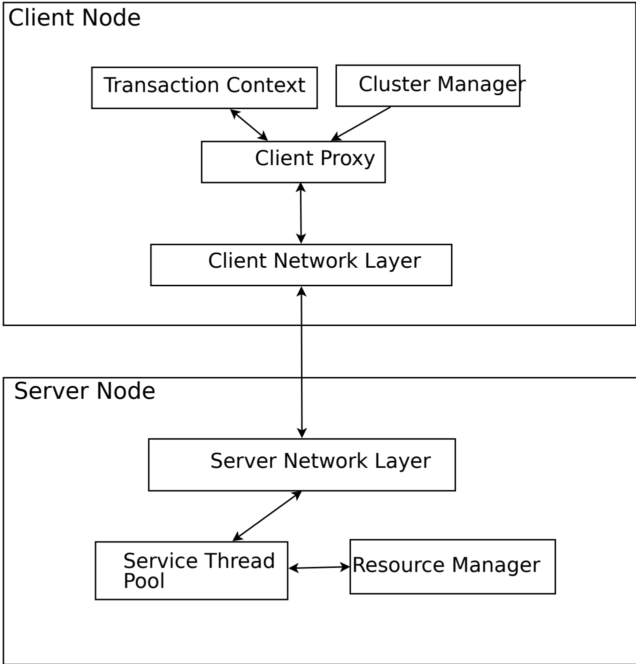


Figure 3.3: Design Diagram.

on whether the infrastructure provides UDP multicast or not, this layer supports two types of clusters that QR-DTM can run on, private and public cluster. In private cluster, messages are send via UDP multicast. In public cluster, UDP multicast is not allowed, hence, multicast is implemented by sending UDP unicast to all the nodes in the quorum.

With UDP there are chances of packets being dropped. We have implemented a simple timeout mechanism on the client side to detect packet drop. On occurance of timeout, message is re-sent. We choose UDP over TCP because we observed minimal packet loss in our testbed setup, and, thus ACK sent for every message in TCP would have been redundent and put extra overhead on the network layer.

We use *Kryo* library for serialization and deserialization of messages. We observed that the network latency reduced from 10ms for Java Serialization Library to 1ms for Kryo Serialization. The reason for this improvement is because Java Serialization being generic generates messages of size 10KB as opposed to only 500B for Kryo.

3.3.3 Feature Implementation

For the closed nesting model, the nested transactions are distinguished from flat nested transactions. Every child transaction keeps note of its parent transaction. This information is noted by remote nodes whenever they process a request from the transaction. The read/write operation of child and root transactions are similar, except that no metadata is maintained on the remote node for child transactions. During the commit of a child transaction, its read-set and write-set are merged with its parent's respective sets.

Each transaction is implemented as a regular function call in Java. A parent transaction must be passed as an argument to the child transaction. A root transaction takes *null* as its argument since it does not have any parent. On successful commit of a child transaction, the execution returns to the parent.

QR-DTM uses Java Exceptions for partial rollback in closed nesting. On abort of a transaction, an exception encapsulating the transaction $abortTxn$ is thrown. Transaction catching this exception compares $abortTxn$ 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 $abortTxn$. 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 [56].

```
1 atomic {

3 for (int i = 0; i < 0xffff; i++) {
          Context context = getTransactionContext(null);
5         Account acc1 = ClientProxy.write(accObj1, context);
          Account acc2 = ClientProxy.write(accObj2, context);
7         acc1.deposit(10);
          acc2.withdraw(10);
9
          if (context.commit()) {
11                        return;
          } else {
13                // restart the transaction
          }
15 }
```

Figure 3.4: Example of Bank Transaction in QR-DTM.

3.3.4 Programming Framework.

Programming framework consist of writing application code in sequential manner and wrapping it inside a transaction. The shared objects require *ClientProxy* API for accessing them. Figure 3.4 is an example of transaction in *Bank* application. Each transaction commit attempt creates a new *Context* object. The object copies are fetched from remote nodes via the *ClientProxy* interface. The read-set and write-set are maintained in within the *Context* object.

Closed nesting requires enclosing the nested transactions inside the parent transaction. In the example code Figure 3.5, the account transfer includes a closed nested transaction as well *updateTotalAmount* for updating the total amount within the bank. It is triggered as a function invocation. Initially, the parent context is recorded by the *getTransactionContext*. The read requests for objects are similar to flat nested transaction, while for the commit *merge* is invoked.

Figure 3.6 shows code for transaction which encloses three open nested transactions of list. Each open nested transaction records the context of parent transaction in *getTransactionContext*.

```
1 atomic {

3 for (int i = 0; i < 0xffff; i++) {
          Context context = getTransactionContext(null);
5         Account acc1 = ClientProxy.write(accObj1, context);
          Account acc2 = ClientProxy.write(accObj2, context);
7         acc1.deposit(10);
          acc2.withdraw(10);
9
          updateTotalAmount(acc1, acc2);
11
          if (context.commit()) {
13                  return;
          } else {
15              // restart the transaction
          }
17 }

19 updateTotalAmount(Account acc1 , Account acc2, Context parentContext) {

21 for (int i = 0; i < 0xffff; i++) {
              Context context = getTransactionContext(parentContext);
23
              Account global = ClientProxy.write(globalAcc, context);
25            global.amount =  acc1.getAmount() + acc2.getAmount();

27            context.merge();
          }
29 }
```

Figure 3.5: Example of closed nested Bank Transaction in QR-DTM.

```
   atomic {
 2
   for (int i = 0; i < 0xffff; i++) {
 4         Context context = getTransactionContext(null);

 6         list.insert(10, context);
           list.insert(23, context);
 8         list.insert(68, context);

10         if (context.commit()) {
                           return;
12         } else {
                   context.compensate();
14         }
   }
16
   insert(int element, Context parentContext) {
18
   for (int i = 0; i < 0xffff; i++) {
20                 Context context = getTransactionContext(parentContext);

22                 listInsert(element);

24                 if (context.commit()) {
                           context.recordAbstractLock(element);
26                 } else {
                           // restart the open nested transaction
28                 }
           }
30 }
```

Figure 3.6: Example of open nested List Transaction in QR-DTM.

Chapter 4

Partial Transactional Abort

Typically, transactions in TM system rollback completely and retry from the beginning of the transaction. Often this results in redundant work. This can degrade the performance of the system, espcially in fault-tolerant DTM, where each operation is network intensive. Partial abort enables a transaction to rollback transactions partially and restart from a consistent transactional state. The rollback point is determined such that all the objects in that state are valid i.e. have not changed since it was read. The execution of transaction can resume from this point without altering the state of the previously read objects.

Following are the definitions of transactions in the system [49] to understand partial aborts.

Closed Nesting Model

- **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. Root transaction is the enclosing transaction of CT. Root transaction can have multiple enclosed CTs.

The rollback occurs at the granularity of closed nested transactions. This means if a conflict occurs during execution of inner transaction then restart will occur either from the inner transaction itself or from the parent transaction, depending on which object has encountered the conflict.

Checkpointing Model

This model requires transactions to support checkpoint creation and rollback.

- **Checkpointing transaction.** A checkpointing transaction (CPT) creates checkpoints based on specific criterion during its execution. Note, this criterion is predefined in the system, as opposed to programmer created manual checkpoints [37]. The checkpoint consists of execution state stack reference and local variable as well as the metadata like read-set and write-set of transaction at that point. Commit of CPT is globally visible.

The rollback occurs at the granularity of checkpointing. This means if a conflict occurs during execution of a CPT then it can resume from a point where the objects are valid. Checkpointing can be seen as generalization of closed nesting.

We have implemented partial aborts through *Closed Nesting (QR-CN)* and *Checkpointing (QR-CHK)* in QR-DTM framework. We describe both the protocols in detail, followed by experimental evaluation.

4.1 QR-CN Protocol

4.1.1 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 [80], 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 4.1.2, 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 3.

4.1.2 QR-CN : 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 3.1, 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 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: 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.

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);
```

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 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 $abortClosed$.

4.1.3 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;
```

4.1.4 Analysis

Theorem 4.1.1. *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 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 [80].

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. □

Theorem 4.1.2. *Rqv provides* 1*-copy serializability.*

Proof. We have already proved that Rqv provides 1-copy equivalence i.e. it provides single copy of an object and that any transaction observing inconsistent state will abort. Assume T_1 and T_2 are conflicting transactions attempting to update an object o, T_1 is propogating changes to object o while T_2 had accessed its read quorum to read o before T_1 had started propogating its changes. Since Rqv provides 1-copy equivalence, T_1 will commit its changes while T_2 will abort as it has the stale version of o. During the next execution of T_2, it

can read the latest copy of object o, apply its changes and commit. Thus, the commit of transactions T_1 and T_2 will appear as sequential execution.

Therefore, any two transactions will appear to commit sequentially. Theorem follows. □

Theorem 4.1.3. *QR-CN preserves* 1*-copy equivalence for all objects.*

Proof. Let transaction T_{parent} be the parent of closed nested transaction T_{closed}. While reading a remote object, if Rqv fails for T_{closed}, then the transactions are rolled back up to the point where read-set and write-set objects are valid, and the execution is restarted from this point. On the contrary, if T_{closed} completes reading all the remote objects, it means that the objects in T_{closed}'s read-set and write-set are valid. Next, T_{closed}'s objects are merged with that of T_{parent} locally, and the execution is returned to T_{parent}. Let T_{root} be a root transaction in QR-CN. The commit or abort of any closed nested transaction enclosed within T_{root} is not globally visible, hence it will not affect any other transaction. T_{root} performs a commit request operation before committing or aborting itself. It commits only when it receives *commit* response from all the nodes in its write quorum, else it aborts.

Therefore, any root transaction in QR-CN observing inconsistent state of object will not commit. Theorem follows. □

4.2 Checkpointing

4.2.1 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.

4.2.2 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*.

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;
```

4.2.3 Analysis

Theorem 4.2.1. *QR-CHK preserves* 1*-copy equivalence for all objects.*

Proof. Let transaction T_{parent} be the parent of closed nested transaction T_{closed}. While reading a remote object, if Rqv ails for T_{closed}, then the transactions are rolled back up to the point where read-set and write-set objects are valid, and the execution is restarted from this point. On the contrary, if T_{closed} completes reading all the remote objects, it means that the objects in T_{closed}'s read-set and write-set are valid. Next, T_{closed}'s objects are merged with that of T_{parent} locally, and the execution is returned to T_{parent}. Let T_{chk} be a transaction in QR-CHK. Any partial rollback occuring within T_{chk} is not globally visible, hence it will not affect any other transaction. T_{chk} performs a commit request operation before committing

or aborting itself. It commits only when it receives *commit* response from all the nodes in its write quorum, else it aborts.

Therefore, any transaction in QR-CHK observing inconsistent state of object will not commit. Theorem follows. □

4.3 Automated Closed Nesting

Closed nesting can significantly improve performance of an application in fault-tolerant DTM. However, in certain cases performance can degrade due to large number of partial aborts. Therefore, it is important to appropriately compose closed nested transactions to utilize the benefits of partial aborts. In order to figure out the correct composition of closed nested transactions, one needs to know the level of contention of the objects involved in transaction execution. Contention of an object depends on factors like write workload, access pattern, number of servers (quorum nodes) and clients in the system. Given the dynamic nature of these factors, contention of objects can only be determined during application runtime. Therefore, it is almost impossible for a developer to correctly compose closed nested transactions.

To this end, we develop an automated closed nesting framework, *QR-ACN*, to dynamically determine the composition and order of closed nested transactions. We follow a hybrid approach where we use static and dynamic analysis to figure out the closed nested transactions that will provide performance improvement. The static analysis involves data dependency analysis of transactional code, while dynamic analysis involves measuring the contention of objects dynamically. The information collected from these analyses is used to appropriately compose closed nested transactions.

4.3.1 Overview

The idea behind automated closed nesting rests on the observation that closed nesting is best applicable to transactions which access high contention objects only when their execution is nearing commit operation. Thus, the transactions which access low contention objects initially while access high contention objects later in their execution can obtain maximum benefit from closed nesting.

Let us consider an example to understand this concept. We consider Bank transaction, where funds are transferred from bank account *Account1* belonging to branch *Branch1* to another account *Account2* belonging to branch *Branch2*. The psuedo code for flat nested transaction is shown in Figure 4.1. Line 1-5 corresponds to reading remote object copies of the branch objects *Branch1* and *Branch2*, and withdrawing and depositing amounts from them, respectively. Line 7-11 corresponds to reading remote object copies and performing

```
 1 T_flat
   branch1 = getRemote(branchId1);
 3 branch2 = getRemote(branchId2);
   branch1.withdraw(amt);
 5 branch2.deposit(amt);

 7 account1 = getRemote(accountId1);
   account2 = getRemote(accountId2);
 9 account1.withdraw(amt1);
   account2.deposit(amt2);
11
   if !commit()
13        retry T_flat;
```

Figure 4.1: Flat Nested Bank Transaction.

withdrawal and deposit on *Account1* and *Account2*, respectively.

During the run-time, the objects *Branch1* and *Branch2* will be highly contended as they are global objects for their respective branches, hence, other transactions will also access them. On the other hand, the objects *Account1* and *Account2* will have low contention as they would have lesser accesses. Therefore, the contention levels of branch and account objects will differ significantly. For this transaction, wrapping branch operation (line 1-5) and account operation (line 7-11) inside closed nested transaction would not yield any performance improvement over flat nesting. This is because probability of abort for branch operation being higher the behavior would be same as that of flat nesting.

Recall the observation stated regarding the applicability of closed nesting. If we look closely at Figure 4.1, there is no data dependency between the branch operation and withdrawal operation. We can potentially change the order of these operations as shown in Figure 4.2. With this change the branch operations are closer to commit operation. We observe that transaction is more likely to encounter conflict due to branch objects, hence, we wrap the branch operation inside closed nested transaction shown in line 6-12 Figure 4.2. The conflict encountered in closed nested transaction can be resolved by partial abort by restarting from line 6.

Note, changing the order of operations will not affect the correctness of transaction. The functional behavior of transaction still remains the same. This is because in both Figure 4.1 and Figure 4.2 the transaction read-set and write-set are exactly in the same state before the commit operation begins. As a result, the state after transaction commit will also be same.

In order to devise such a framework to automatically determine the composition of closed nested transactions, we need mechanism to find out the data dependency and contention

```
1 T_parent
  account1 = getRemote(accountId1);
3 account2 = getRemote(accountId2);
  account1.withdraw(amt1);
5 account2.withdraw(amt2);

7 T_closed
          branch1 = getRemote(branchId1);
9         branch2 = getRemote(branchId2);
          branch1.withdraw(amt);
11        branch2.deposit(amt);
          if (!commit())
13                retry T_closed

15 if !commit()
        retry T_flat;
```

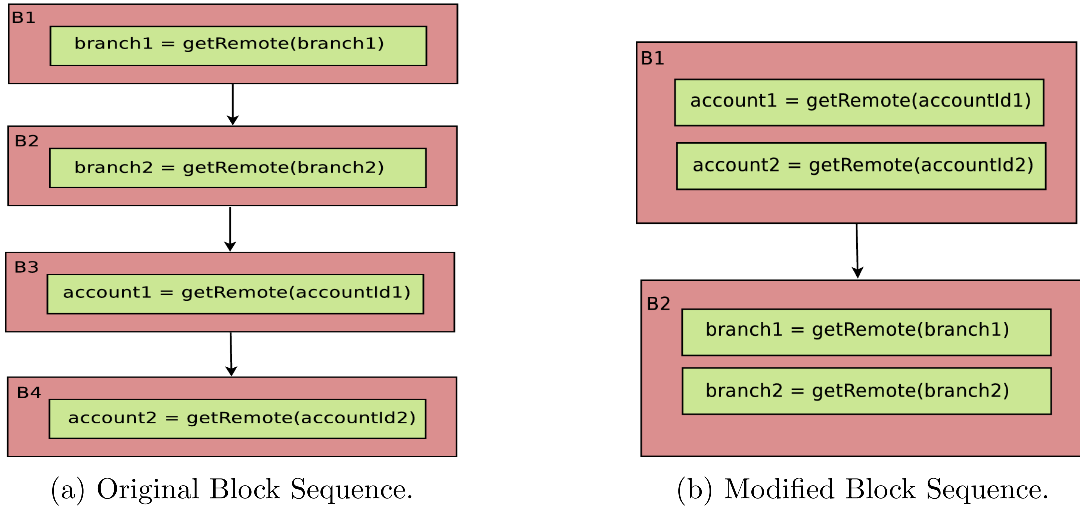
Figure 4.2: Flat Nested Bank Transaction.

level of objects. Also, in order to change the object access order for transaction we need to change the transaction execution dynamically. This requires the code to be arranged in a specific manner such that composition of closed nested transactions can be changed dynamically.

4.3.2 System Design

QR-ACN can change the composition of closed nested transaction dynamically depending on the contention of the objects accessed. To achieve this, the transaction code has to be transformed during initialization. We will introduce certain definitions to understand how the execution for transaction occurs in QR-ACN:

- *UnitBlock* : We divide the given transactional code in distinct sections called as *unitBlocks*. It is the smallest logical unit of code in QR-ACN. Each unitBlock comprises of exactly one remote object invocation and local operations performed on the already opened object copies. The contention level of an unitBlock is the contention level of the remote object it opens.
- *Blocks* : Multiple UnitBlocks can be combined to form a *Block*. The contention level of block is the average of all the unitBlocks enclosed inside it. Thus, block enclosing multiple unitBlocks represent a piece of code to be executed.
- *Executor Engine* : This engine is responsible for maintaining the sequence of blocks that comprises a transaction and execute these blocks in that order. In order to execute



(a) Original Block Sequence. (b) Modified Block Sequence.

Figure 4.3: Code Arrangement using UnitBlocks and Blocks.

a block, it executes all the unitBlocks enclosed inside it. The sequence of the blocks maintained by Executor can be changed, in which case, the order of execution changes.

Figure 4.3 shows the bank transaction arranged in unitBlocks and blocks. In the orginal block sequence (Figure 4.3a), each block contains exactly one unitBlock which in turn consists of one remote object invocation. After transformation (Figure 4.3b), the unitBlocks corresponding to branch objects are merged inside a single block *B2*. Similarly, account objects are merged into block *B1*. As the order of execution has changed, block *B1* precedes *B2* in the block sequence. Note, we have not shown local operations inside unitBlocks for the sake of clarity.

We now describe each of the modules of QR-ACN :

- *Static Module* : The static module maintains static information of the transaction code. It is triggered at the beginning of the application where it creates graph model of the transactional code, called *UnitGraph*. This information is queried throughout the lifetime of application. During the run-time, it is queried by the Algorithm Module for data dependency.

- *Dynamic Module* : The dynamic module is triggered as soon as the application starts. It collects run-time parameters like the contention level of objects. It is used as input to the Algorithm Module to know contention levels of objects. The Algorithm Module requests the information and feeds the response to the Algorithm Module.

- *ACN Module* : This Automated Closed Nesting (ACN) Module is invoked periodically. It has the input from Static Module and Dynamic Module which is processed based

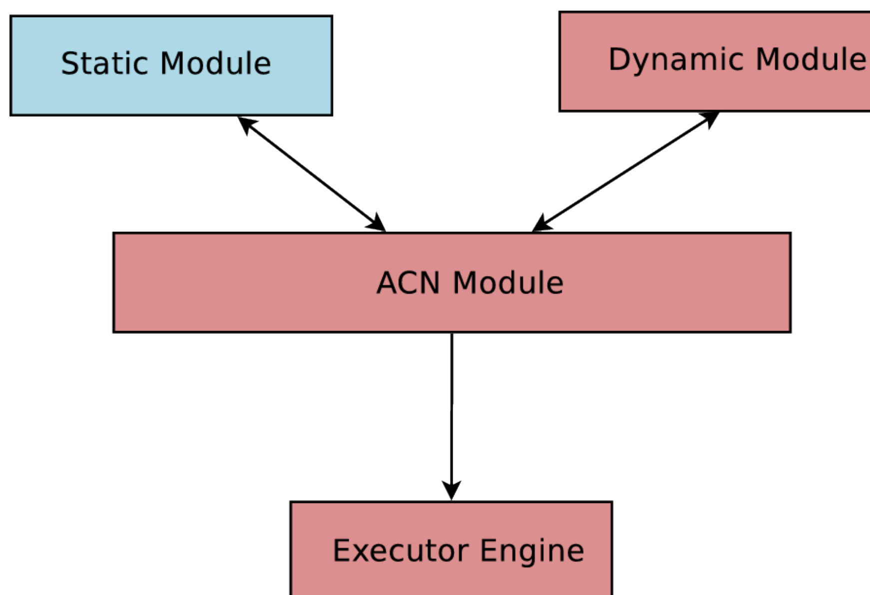


Figure 4.4: Block Diagram.

on the algorithm to produce a sequence of blocks. This Executor Engine picks up the changed sequence of blocks and executes them in that order.

Figure 4.4 shows the interaction of these modules with the Executor Engine. The Static Module is initialized only at the start of application (colored blue) while the other modules are active during the application run-time. The ACN Module queries the Static Module and Dynamic Module to create the next block sequence.

4.3.3 Algorithm

Control-flow Graph

Soot [75] is a language manipulation framework for Java. It converts Java classes into intermediate languages which can be used through its API for program analysis. We use Soot to create a control-flow graph, UnitGraph, to analyze the transactional code. In this graph, each statement is a node and one node is connected to another node if control can flow from the source node to the target node. This graphical representation of the transaction is used to detect dependency among *Blocks* to decide whether it can be shifted.

Next, we identify remote object calls from the control-flow graph. Code blocks are created such that there is remote invocation at the start of the blocks while it ends before the subsequent invocation (exclusive). For every such code block, a new unitBlock is created and registered with the ACN Module. Every unitBlock is associated with the remote object's

class. Once every code block of the transaction is encompassed inside a unitBlock, a new block is created for each of the unitBlocks. The new blocks are also registered with the ACN Module.

The Executor Engine looks for the registered blocks of a transaction and arranges them in a sequence of blocks that adheres to the one in UnitGraph. This forms the initial block sequence.

Note, in the benchmarks we have used every object of a given class has similar behavior, hence, any object can be used as a representative of its class. From now on, we would use the term object for referring to its class.

4.3.4 Contention Level

Contention Level for each object is maintained by the quorum nodes. Quorum nodes track the total number of object accesses (*TransactionAccess*) and total number of aborts due to that objects (*TransactionAbort*). *TransactionAccess* is incremented by on read request for that object. *TransactionAbort* is incremented on encountering abort due to that object while reading the object or during the commit operation for a transaction. At any point of time, the contention level of an object is calculated as the ratio of *TransactionAbort* and *TransactionAccess*.

For obtaining the blocks sequence for providing better performance, the clients request the contention level of objects. They create list of objects that are contained in the block sequence. This list is sent as part of request to the quorum nodes. On receiving the request, the quorum nodes calculate the contention level of that object as the ratio of *TransactionAbort* and *TransactionAccess*, and the list of contention levels is sent back to the requesting client. The pseudo code for request and response messages is shown :

4.3.5 Algorithm Module

The Algorithm Module is triggered periodically on the client nodes. The input for this module is the current block sequence, contention level for each of the blocks and dependency model between the blocks, while the output is the new block sequence to improve the performance of the system dynamically. The dependency model that is created during the initialization of the system, while the contention level of objects at that point of time is requested from the quorum nodes. Contention level of block is defined as the average of contention level of the unitBlocks it is enclosing.

The Algorithm module works on the basis that high contention objects should be as close to commit as possible. In order to achieve this, it attempts to shift higher contention blocks as low as possible while maintaining the data dependency tracked in the UnitGraph.

Algorithm 5: Contention Level Request and Response.

```
   procedure RequestContention (T)
1  Local:
2  blockList = getBlocks(T);
3   foreach block in blockList do
4        foreach unit in block.units do
5            objectList.add(unit.obj);
6  response = sendRequest(objectList, ReadQuorum);
7  updateContentionLevel(response)
   procedure ResponseContention (objectList)
8  Remote:
9  response = null;
10  foreach o in objectList do
11      contentionLevel = getTransactionAccess(o) / getTransactionAbort(o);
12      response.put(o, contentionLevel);
13 return response;
```

The current block sequence is sorted based on the contention values of the blocks. For every block starting from the lowest contention level, we try to shift it such that all the blocks above it have lower contention level while preserving the data dependency. The pseudo code for this is shown in Algorithm 6. The *blockSeq*, the current block sequence, is updated with new contention levels by *updateContentionLevel*. Next, *getCurrentBlocks* provides the list of all the blocks. The list is then sorted based on contention level. For every block in the sorted list, we check whether it can be shifted below its adjacent block by checking with the UnitGraph (*canShift*(), line 6) for any edge among the blocks and for the contention level. If *block* does not have data dependency and its contention level is lower than *sBlock* then it is shifted below the given block(*shift*, line 7), else the inner loop breaks.

Algorithm 6: Automated Closed Nesting Algorithm.

```
   procedure getNewBlockSeq (blockSeq)
1  Local:
2  updateContentionLevel(blockSeq);
3  sortedBlocks = getCurrentBlocks(T).sort();
4   foreach sBlock in sortedBlocks do
5        foreach blockwithindexgreaterthansBlock in blockSeq do
6            if canShift(sBlock, block) then
7                 shift(sBlock, block);
             else
8                 break;
```

Chapter 5

Experimental Evaluation for Partial Abort

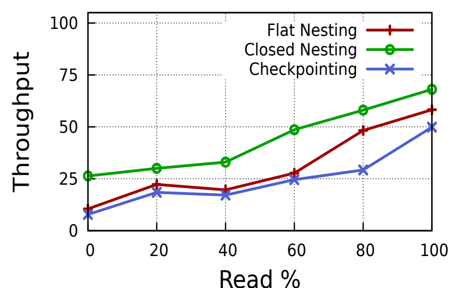
We conducted the experiments using a 40-node testbed. Each node is an AMD Opteron processor clocked at 1.9GHz and running the Ubuntu Linux 10.04 server operating system. Each node was assigned the same read and write quorums.

In section 5.1, 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 [57]) and STAMP benchmark [44] Vacation. In Section 5.2, we compare existing DTM implementations with QR-DTM for Bank benchmark. In Section 5.3, we show graceful degardation in failure scenario for three benchmarks.

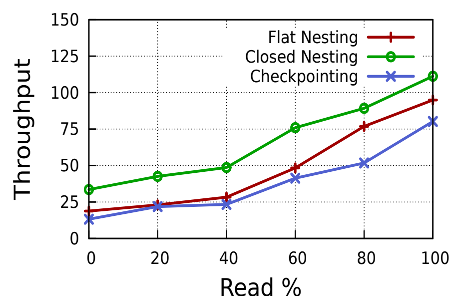
5.1 Evaluation of Closed Nesting and Checkpointing

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.

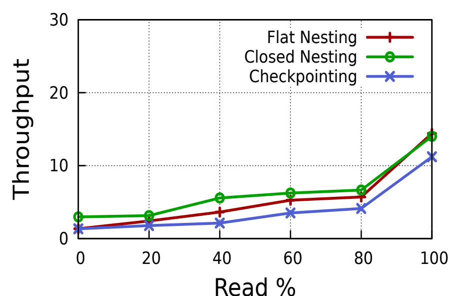
In these experiments, we measured the throughput (transactions committed per second) by varying the following parameters: 1) read workload i.e. the percentage of read operations 2) the number of nested calls, which affects the number of objects read within a transaction, thereby controlling its length,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



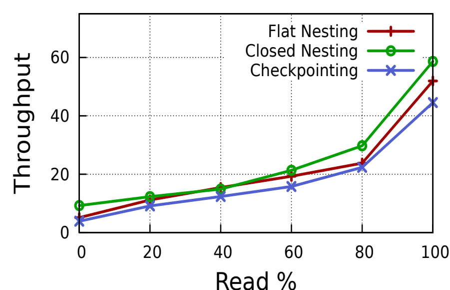
(a) Bank.



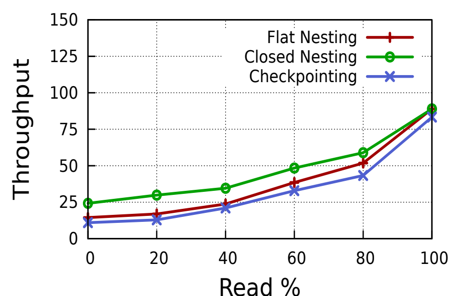
(b) Hashmap.



(c) Skiplist.

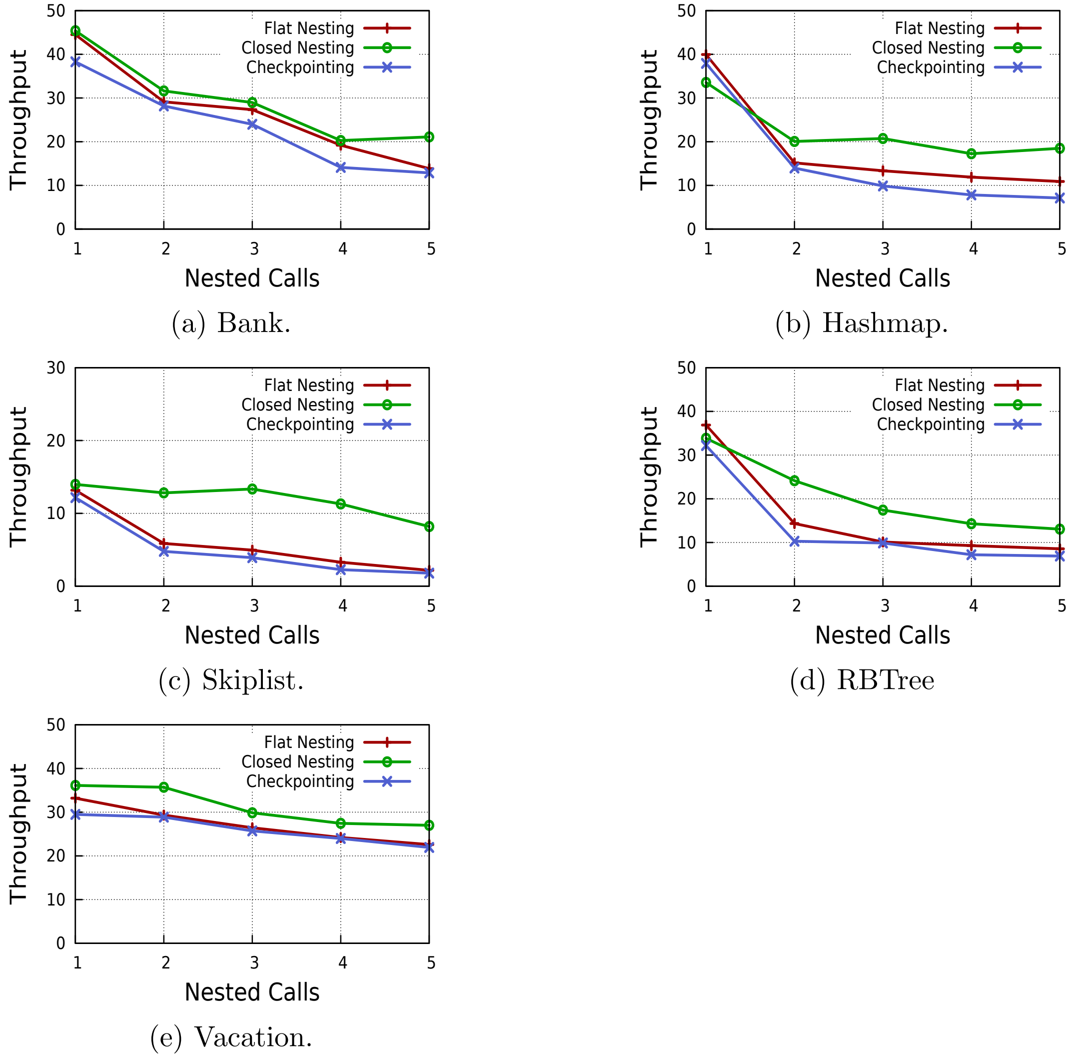


(d) RBTree.



(e) Vacation.

Figure 5.1: Flat nesting, closed nesting and checkpointing for different read workload.



(a) Bank.

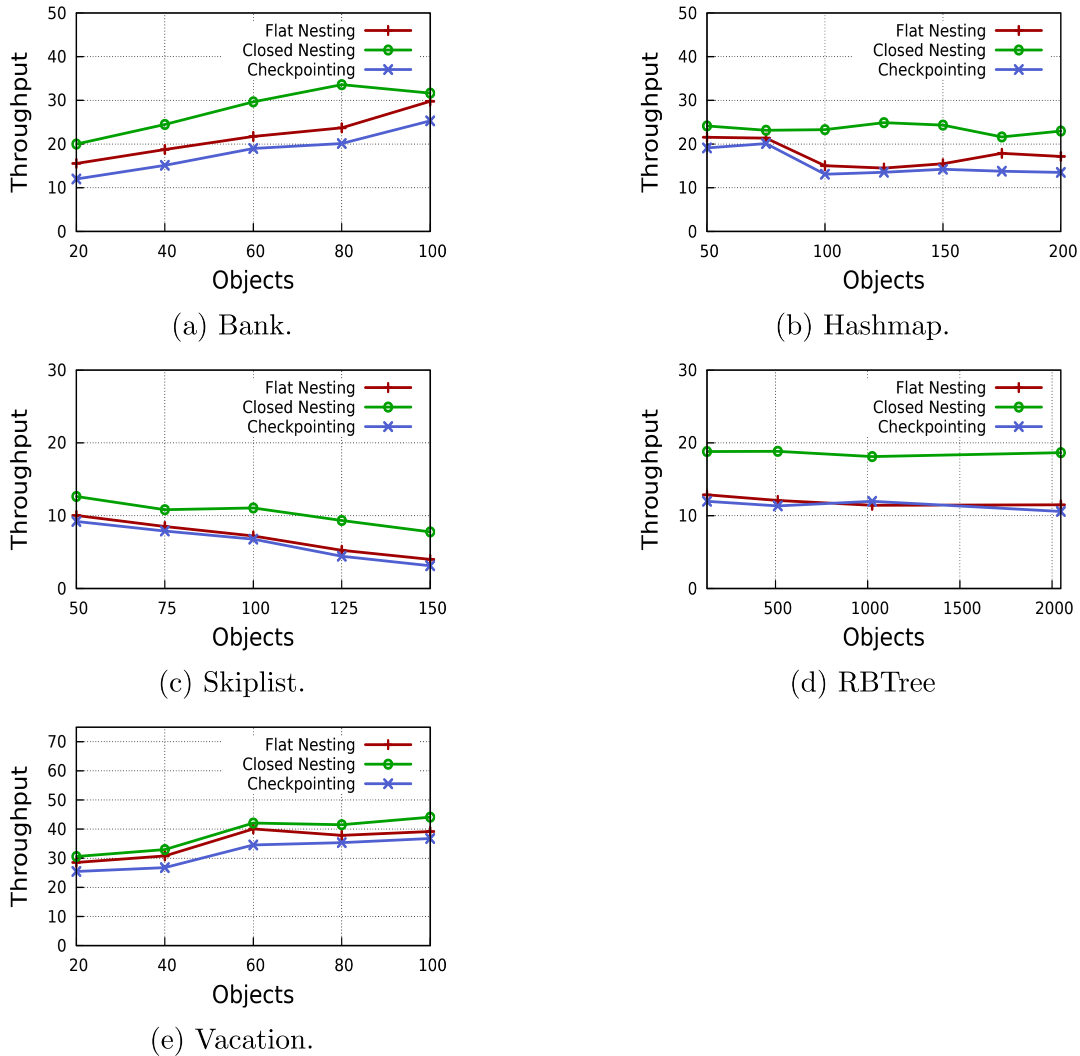
(b) Hashmap.

(c) Skiplist.

(d) RBTree

(e) Vacation.

Figure 5.2: Flat nesting, closed nesting and checkpointing for different transaction length.



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

Figure 5.3: Flat nesting, closed nesting and checkpointing for different number of objects.

Figures 5.1a, 5.1b, 5.1c, 5.1d and 5.1e show the variation with read workload for five benchmarks, varied from 0 to 100 %. Figures 5.2a, 5.2b, 5.2c, 5.2d and 5.2e show the variation with number of calls, varied from 1 to 5. Figures 5.3a, 5.3b, 5.3c, 5.3d and 5.3e 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.

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

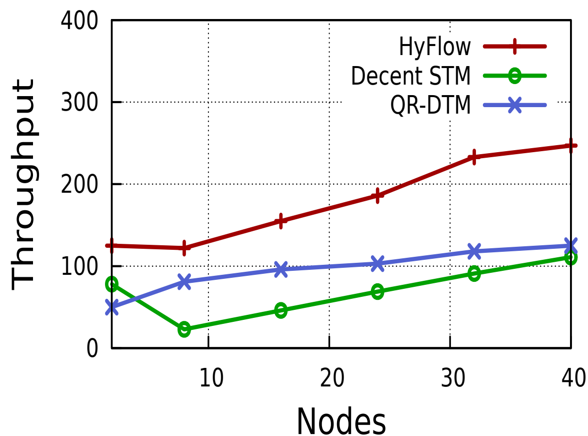
Table 5.1: Abort rate and message % for QR-CN and QR-CHK compared to flat nesting

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 5.1 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 5.1. 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.

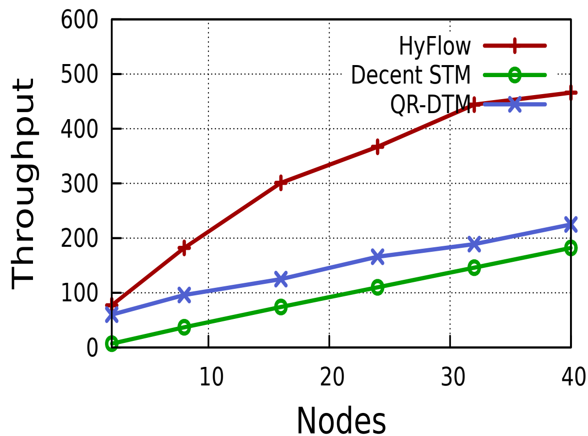
From the results so far, we also observe that the improvement obtained by closed nesting increases with increase in contention. We also note that the length of transactions has a significant effect in determining the throughput gain for closed nesting. The SList benchmark, which have large lengths, have high throughput gain (as high as 122 %). In contrast, the average gain for rest of the benchmarks with shorter lengths is 40%.

5.2 Comparison with DTM Implementations

We compared QR-DTM with other DTM implementations including HyFlow [57] and Decent STM [12]. 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



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



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

Figure 5.4: Throughput of QR-DTM, HyFlow, and Decent-STM for the bank application

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 [60], and therefore serves as a good baseline for us.

Figure 5.4 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 triggers a multicast message to read and write quorum as opposed to unicast message to the object owner in HyFlow. However, HyFlow cannot cope

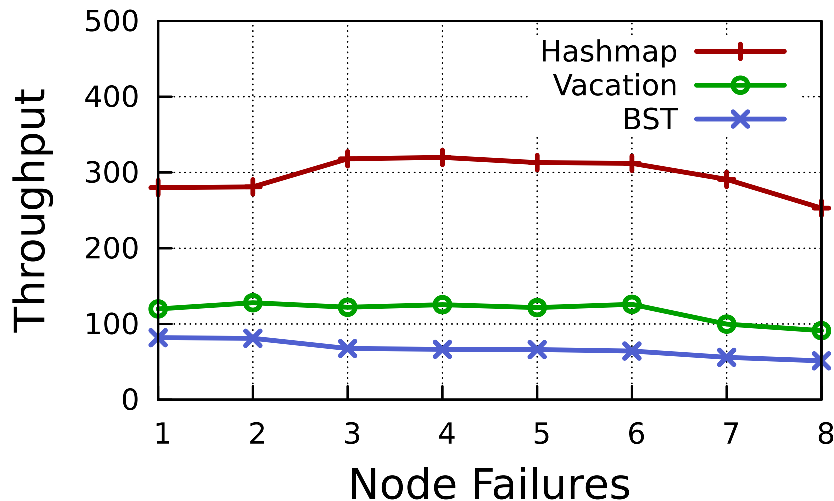


Figure 5.5: Throughput under increasing node failures.

with failures.

5.3 Throughput under Node Failures

We now show fault-tolerance of QR-DTM by measuring throughput under node failures. Figure 5.5 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.

5.4 Evaluation of Automated Closed Nesting

We test the QR-ACN framework for two real-world benchmarks, TPC-C and Vacation. We perform the experiments for configuration of 10 quorum nodes and upto 30 clients. We compare throughput of QR-DTM with system using QR-ACN. For simplicity, when QR-DTM uses QR-ACN framework we will refer to such a system as QR-ACN.

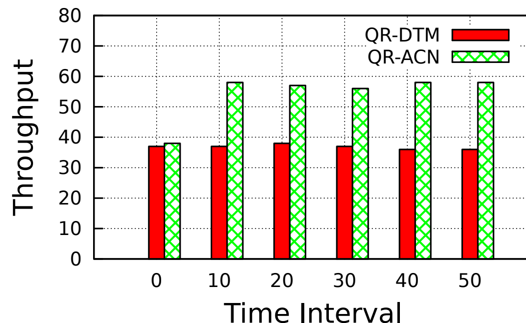


Figure 5.6: 100% NewOrder Transactions.

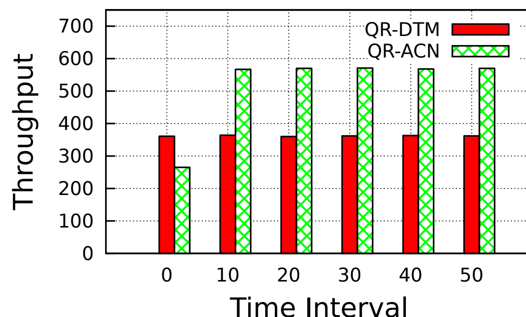


Figure 5.7: 100% Payment Transactions.

TPC-C Benchmark

We compare QR-ACN and QR-DTM for write intensive NewOrder transaction in TPC-C benchmark. We configure the clients to generate 100% NewOrder transactions. Figure 5.6 shows the throughput of QR-DTM and QR-ACN for different time intervals. We have measured the throughput of the system for every 10 second time interval upto 50 seconds. We observe that both the systems have the same throughput up to $t = 10$ sec. In the next time interval, QR-ACN determines that the high contention objects belong to District. The transactional code as per specification performs the remote operations initially in the execution. QR-ACN moves the remote operation on District as close to commit operation as possible by looking at the data dependencies. This shift results in throughput improvement of 30% over QR-DTM.

We perform similar comparison for Payment transaction of TPC-C. The throughput variation with time is shown in Figure 5.7. We observe that initially the throughput for QR-ACN is lower than that of flat nesting, however, after $t = 10$ sec, composition is changed. QR-ACN determines the high contention objects to be District and Warehouse, and shifts them close to commit operation. This shift results in throughput improvement of 54% over QR-DTM.

Next, we compare QR-DTM and QR-ACN in TPC-C profile with 50% of Payment and

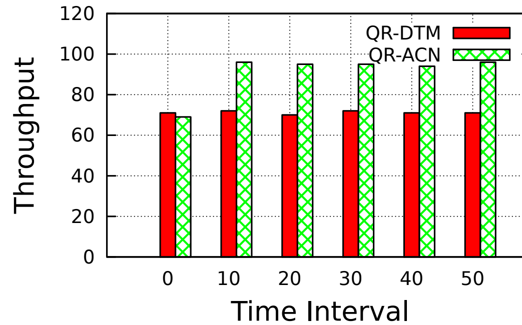


Figure 5.8: 50% Payment and New Order Transactions.

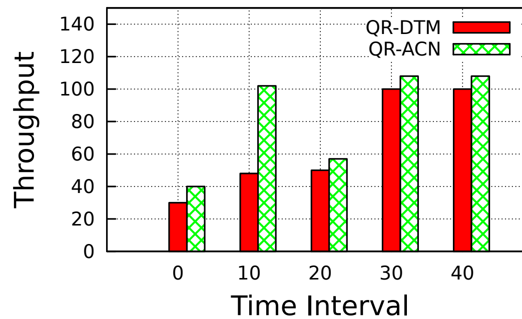


Figure 5.9: Vacation Benchmark.

NewOrder transactions as shown in Figure 5.8. We observe that after QR-ACN kicks in the throughput improvement over QR-DTM is 42%.

Vacation Benchmark

We evaluate QR-ACN for Vacation benchmark. In our setup, we changed the contention of objects during different time intervals. Figure 5.9 shows the variation of throughput for QR-DTM and QR-ACN for the time intervals. The highly contended objects change in the second and fourth time interval. In the second time interval, we observe that QR-ACN adjusts dynamically to the changing workload to provide throughput improvement of 112% over QR-DTM, whose throughput remains the same. In the fourth time interval, QR-DTM's throughput increases as its existing composition favors the changed workload. QR-ACN still outperforms QR-DTM by 8%.

Chapter 6

Validation Cost Reduction

6.1 Open Nesting

Open nesting was introduced by Moss [48] 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 their 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 6.1 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 6.1: 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, physical serializability is violated as partial changes of a transaction are exposed. However, at the semantic level, ordering among the open-nested transactions does not matter. For example, adding elements 10 and 15 to a set are commutative operations, and therefore, the set's final state is not affected. Thus, open-nesting preserves *abstract serializability* [50]. In our early release protocol, QR-ER, physical serializability is violated as transactions ignore validation of certain read-set objects during their commit. However, updates on other sections of the shared data (i.e., data structures) do not affect the current transaction. This protocol is strictly applicable to data structures, and we argue about its correctness in Section 6.3.

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.

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 3.

The commit 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 transac-

tion 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 7 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 invokes *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).

Algorithm 7: 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();
```

6.2 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.

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 8 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.

Algorithm 8: 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);
```

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.

6.3 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 [69, 29], but they address the problem in centrailized 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, [69, 29] release objects from the read-set.

We build QR-ER on top of the QR-CN protocol [21], which extends the QR protocol with the closed nesting model [49]. QR-ER inherits all properties of the closed nesting model: 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. The nested transaction's commits are local, while the parent transaction commits globally. During the parent transaction's commit, only objects with true *validate* flags are validated.

Algorithm 9: 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 9 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 6.3.1. *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 [31]. 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 which 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.

Chapter 7

Experimental Evaluation for Validation Cost Reduction

We experimentally evaluated the protocols using the micro-benchmarks Linked-List, Hashmap, and Binary Search Tree (BST) as well as real-world application like TPC-C benchmark [1] and Vacation [44]. For the benchmarks, each transaction consisted of multiple (open) nested transactions, each of which enclosed an operation on the data structure (TPC-C is based on Hashmap). We also included the original QR-DTM in our comparison. Open nesting has also been evaluated in [73] presenting TFA-ON. However, this work focuses on distributed (non-replicated) systems. Our testbed consisted of 13 nodes, 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.

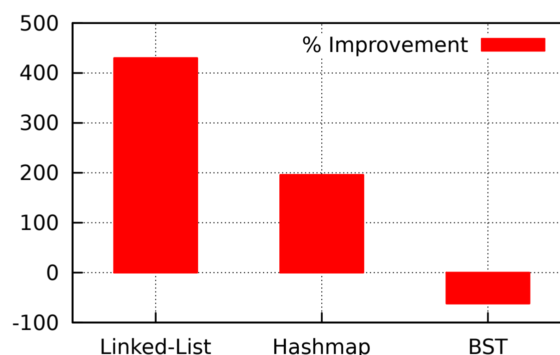
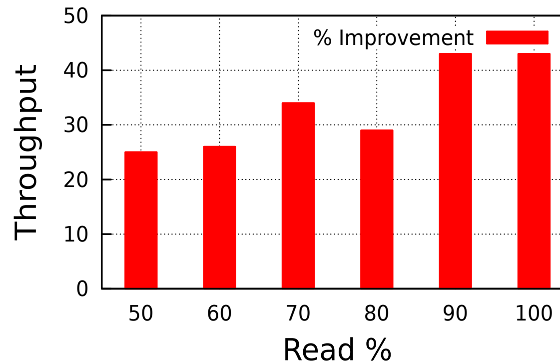


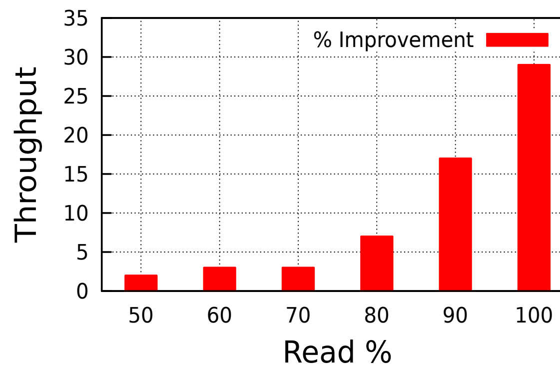
Figure 7.1: QR-ON vs QR-DTM.

7.1 Evaluation of Open Nesting

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.



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



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

Figure 7.2: Speed-up of QR-OON and QR-ER over QR-ON with increasing read %.

Figure 7.1 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, and 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 does not suffer from false conflicts. As a consequence, QR-ON pays the overhead of committing (open) nested transactions without reaping open nesting's benefits.

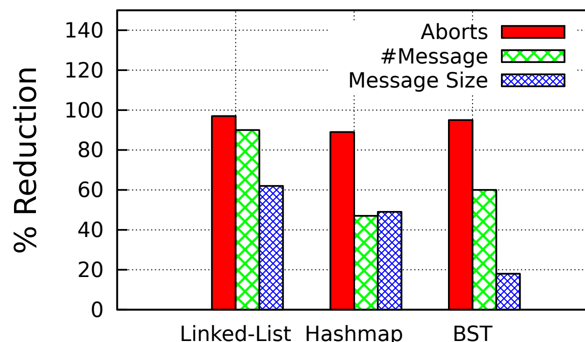


Figure 7.3: Abort rate and number/size of messages in QR-ER vs. QR-ON.

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 designed to overcome this.

7.2 Evaluation of Optimistic Open Nesting

Figure 7.2 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.

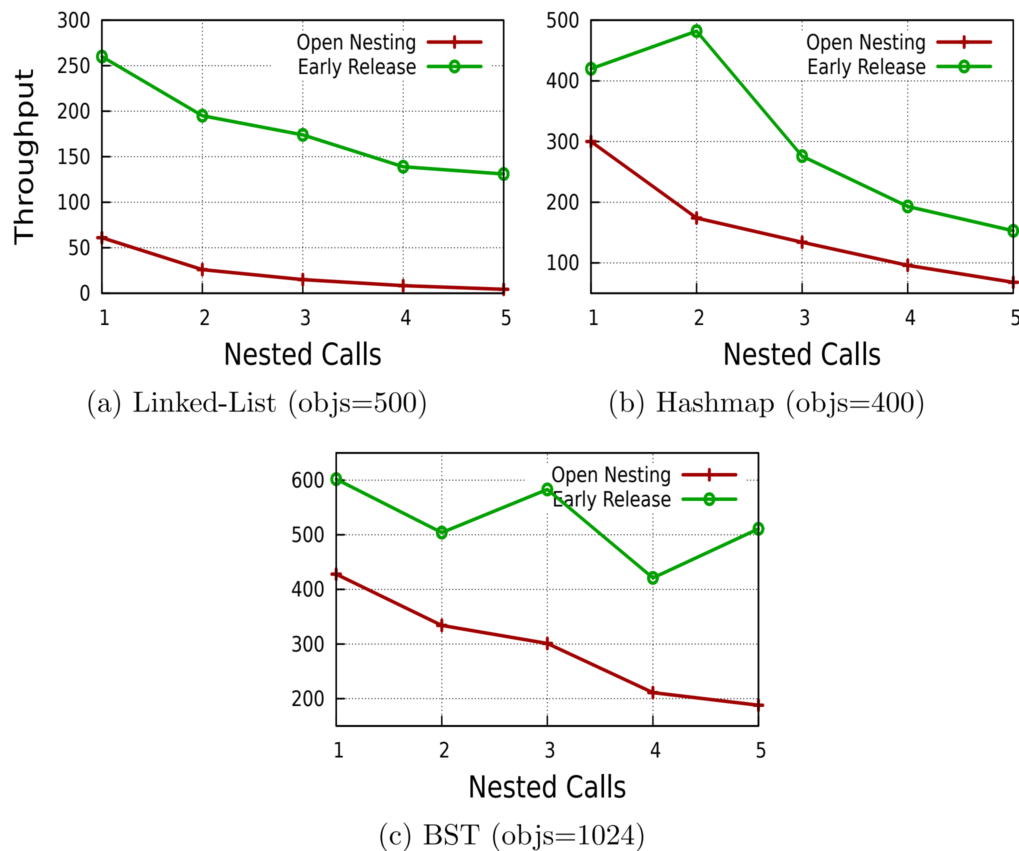
7.3 Evaluation of Early Release

QR-OON is susceptible to workload changes and its performance 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 7.4) and the number of nested transactions (Figure 7.5). The workload is configured with 50% of write transactions.

We observe that QR-ER outperforms QR-ON by as much 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 7.3 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. 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%.



(a) Linked-List (objs=500) (b) Hashmap (objs=400)

(c) BST (objs=1024)

Figure 7.4: QR-ER vs. QR-ON: throughput with increasing nested calls.

We also compared QR-ER against QR-ON for the TPC-C benchmark. We conducted this experiment on our private cluster, as well as on the Future Grid public infrastructure[1]. Figure 7.6 shows how throughput varies with the number of nodes. We observe that QR-ER outperforms QR-ON by 44% on average.

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

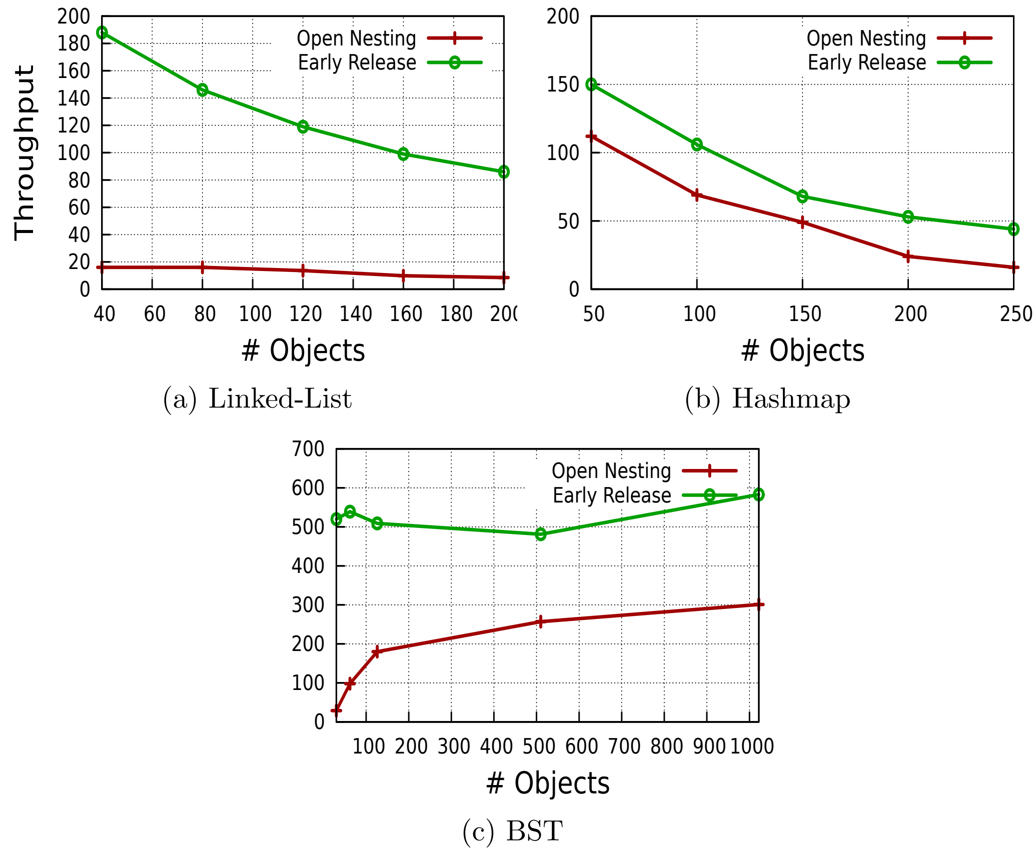


Figure 7.5: QR-ER vs. QR-ON: throughput with increasing object count (Calls=3).

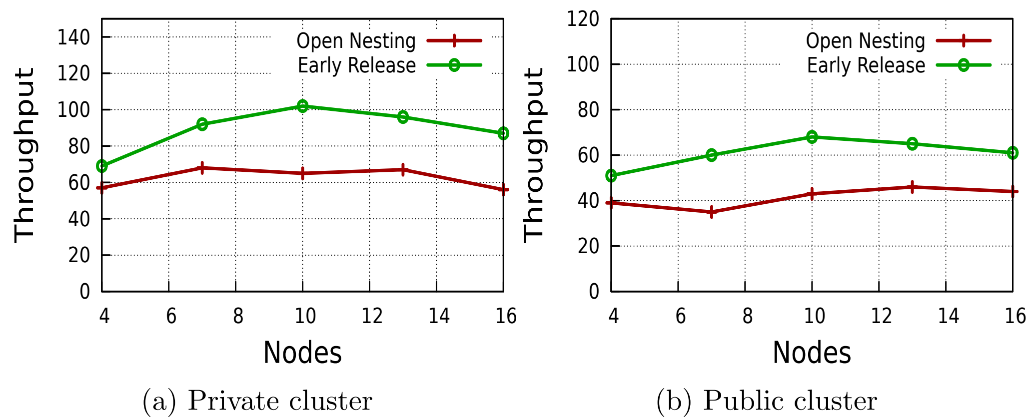


Figure 7.6: QR-ER vs. QR-ON: throughput with TPC-C.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

We presented the QR-CN and QR-CHK protocols that support closed nesting and checkpointing in quorum-based replicated DTM. We showed that Rqv ensures 1-copy equivalence and serializability. 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.

We developed ,QR-ACN, a framework for automated closed nesting. QR-ACN can dynamically adjust the composition and order of closed nested transactions. We found that the average throughput improvement for QR-ACN over flat nesting is 51% across all benchmarks while the maximum throughput improvement is 112% for Vacation benchmark. We determined that QR-ACN outperforms QR-DTM in most of the scenarios and is best suited for applications where workload changes during run-time.

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.

8.2 Future Work

Checkpointing allows fine grained rollback, which is not possible in closed nesting. We can consider checkpointing at specific points in transaction to maximize performance. The automated framework can be extended for automatically creating checkpoints during transaction execution.

Open nesting and early release impose additional complexity on the developer. An interactive framework to guide developer for creating inner transactions and dropping objects from transaction read-set will be extremely useful.

We have performed our evaluation in QR-DTM which uses data flow model. It would be interesting to perform similar evaluation in control flow and hybrid models.

There are distributed systems that use partitioning of data so as to execute transactions locally within a partition. They provide partial replication by replicating the partitions. While most of the transactions access local partitions, few transactions can require access to remote partitions. In these scenarios, partial aborts and validation cost reduction approaches can be explored to improve performance of distributed transactions.

Bibliography

[1] Tpc-c benchmark : Transaction processing performance council.

[2] Advanced Micro Devices, Inc. Advanced Synchronization Facility – Proposed Architectural Specification, 2.1 edition. `http://developer.amd.com/assets/45432-ASF_Spec_2.1.pdf`, 2009.

[3] Transactional Memory in GCC. `http://gcc.gnu.org/wiki/TransactionalMemory`, 2012.

[4] 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.

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

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

[7] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: a new paradigm for building scalable distributed systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174. ACM, 2007.

[8] G. R. Andrews. *Concurrent programming: principles and practice.* Benjamin/Cummings Publishing Company, 1991.

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

[10] G. Barnes. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 261–270. ACM, 1993.

[11] P. Bernstein and N. Goodman. Multiversion concurrency controltheory and algorithms. *TODS*, pages 465–483, 1983.

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

[13] R. L. Bocchino et al. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, 2008.

[14] 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.

[15] N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *IEEE NCA11*, pages 271–274, 2011.

[16] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A high-performance SPARC CMT processor. *Micro, IEEE*, 29(2):6–16, 2009.

[17] W. Chen, S. Lin, Q. Lian, and Z. Zhang. Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. In *PRDC '05*, pages 7–14. IEEE Computer Society, 2005.

[18] D. Christie, J. Chung, et al. Evaluation of AMD's advanced synchronization facility within a complete transactional memory stack. In *EuroSys*, pages 27–40, 2010.

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

[20] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 21–21. USENIX Association, 2012.

[21] A. Dhoke, B. Ravindran, and B. Zhang. On closed nesting in replicated distributed transactional memory. *IPDPS 2013*, 2012.

[22] P. Felber, C. Fetzer, and T. Riegel. Dynamic performance tuning of word-based software transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 237–246. ACM, 2008.

[23] G. Golan-Gueta, N. Bronson, A. Aiken, G. Ramalingam, M. Sagiv, and E. Yahav. Automatic fine-grain locking using shape properties. In *ACM SIGPLAN Notices*, volume 46, pages 225–242. ACM, 2011.

[24] J. Gray. The transaction concept, virtues and limitations. In *Proceedings of 7th VLDB*, pages 144–154, 1981.

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

[26] M. Herlihy. The art of multiprocessor programming. In *PODC*, pages 1–2, 2006.

[27] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[28] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. In *ACM SIGPLAN Notices*, volume 41, pages 253–262. ACM, 2006.

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

[30] M. Herlihy and J. Moss. Transactional memory: Architectural support for lock-free data structures. *ACM SIGARCH computer architecture news*, 21(2):289–300, 1993.

[31] M. Herlihy and N. Shavit. *Art of Multiprocessor Programming.* Morgan Kaufman, 2011.

[32] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *DISC*, pages 324–338, 2005.

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

[34] M. D. Hill and M. R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, 2008.

[35] Intel Corporation. Intel C++ STM Compiler 4.0, Prototype Edition. `http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/`, 2009.

[36] James Reinders. Transactional synchronization in Haswell. `http://software.intel.com/en-us/blogs/2012/02/07/transactional-synchronization-in-haswell/`, 2012.

[37] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA*, pages 160–168, 2008.

[38] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. Distm: A software transactional memory framework for clusters. In *Parallel Processing, 2008. ICPP'08. 37th International Conference on*, pages 51–58. IEEE, 2008.

[39] C. Kotselidis et al. DiSTM: A software transactional memory framework for clusters. In *ICPP*, pages 51–58, 2008.

[40] J. R. Larus and R. Rajwar. *Transactional Memory.* Morgan and Claypool, 2006.

[41] S. Lie. Hardware support for unbounded transactional memory. Master's thesis, MIT, 2004.

[42] K. Manassiev et al. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP*, pages 198–208. 2006.

[43] 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.

[44] C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun. Stamp: Stanford transactional applications for multi-processing. In *IISWC*, pages 35–46, 2008.

[45] S. Mishra. *HyflowCPP: A Distributed Transactional Memory framework for C+*. PhD thesis, Virginia Polytechnic Institute and State University, 2013.

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

[47] J. E. B. Moss. Open nested transactions: Semantics and support. In *WMPI*, 2005.

[48] J. E. B. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, volume 28, 2006.

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

[50] Y. Ni, V. S. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 68–78. ACM, 2007.

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

[52] S. Peluso, P. Romano, and F. Quaglia. Score: a scalable one-copy serializable partial replication protocol. In *Proceedings of the 13th International Middleware Conference*, pages 456–475. Springer-Verlag New York, Inc., 2012.

[53] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465. IEEE, 2012.

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

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

[56] J. Rose. Multi-language virtual machine.

[57] M. M. Saad and B. Ravindran. Distributed Hybrid-Flow STM : Technical Report. Technical report, ECE Dept., Virginia Tech, 2010.

[58] M. M. Saad and B. Ravindran. HyFlow: A high performance distributed software transactional memory framework. In *HPDC*, 2011.

[59] M. M. Saad and B. Ravindran. Supporting STM in distributed systems: Mechanisms and a Java framework. In *TRANSACT*. ACM, 2011.

[60] M. M. Saad and B. Ravindran. Transactional forwarding algorithm. Technical report, ECE Dept., Virginia Tech, January 2011.

[61] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. C. Minh, and B. Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 187–197. ACM, 2006.

[62] W. N. Scherer III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248. ACM, 2005.

[63] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 214–224. IEEE, 2010.

[64] T. Schmitt, N. Kammer, P. Schmidt, A. Weggerle, S. Gerhold, and P. Schulthess. Rainbow os: A distributed stm for in-memory data clusters. In *MIPRO*, pages 200–208, 2011.

[65] F. B. Schneider. Replication management using the state-machine approach, distributed systems. 1993.

[66] N. Shavit and D. Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[67] M. F. Spear, V. J. Marathe, W. N. Scherer III, and M. L. Scott. Conflict detection and validation strategies for software transactional memory. In *Distributed Computing*, pages 179–193. Springer, 2006.

[68] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *ECOOP*, pages 1–3, 2002.

[69] R. Titos-Gil, M. E. Acacio, J. M. Garcia, T. Harris, A. Cristal, O. Unsal, I. Hur, and M. Valero. Hardware transactional memory with software-defined conflicts. *ACM Transactions on Architecture and Code Optimization (TACO)*, 8(4):31, 2012.

[70] M. Tremblay. Transactional memory for a modern microprocessor. In *PODC*, pages 1–1, 2007.

[71] D. M. Tullsen, J. L. Lo, S. J. Eggers, and H. M. Levy. Supporting fine-grained synchronization on a simultaneous multithreading processor. In *High-Performance Computer Architecture, 1999. Proceedings. Fifth International Symposium On*, pages 54–58. IEEE, 1999.

[72] A. Turcu and B. Ravindran. Hyflow2: A high performance distributed transactional memory framework in scala. Technical report, Technical report, Technical report, Virginia Tech, April 2012. URL http://hyflow. org/hyflow/chrome/site/pub/hyflow2-tech. pdf.

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

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

[75] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot: A java bytecode optimization framework. In *CASCON First Decade High Impact Papers*, pages 214–224. IBM Corp., 2010.

[76] J. D. Valois. Implementing lock-free queues. In *Proceedings of the seventh international conference on Parallel and Distributed Computing Systems*, pages 64–69, 1994.

[77] J. D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM, 1995.

[78] B. Zhang et al. Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS*, pages 48–53, 2009.

[79] B. Zhang and B. Ravindran. Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *SRDS '09: Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, pages 268–277, Washington, DC, USA, 2009. IEEE Computer Society.

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