Scheduling Memory Transactions in Distributed Systems

Junwhan Kim

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

Doctor of Philosophy
in
Computer Engineering

September 3, 2013
Blacksburg, Virginia

Keywords: Software Transactional Memory, Distributed Systems, Transactional Scheduling,
Partial Replication, (Parallel) Nested Transactions

Scheduling Memory Transactions in Distributed Systems

Junwhan Kim

(ABSTRACT)

Distributed transactional memory (DTM) is an emerging, alternative concurrency control model that promises to alleviate the difficulties of lock-based distributed synchronization. In DTM, transactional conflicts are traditionally resolved by a contention manager. A complementary approach for handling conflicts is through a transactional scheduler, which orders transactional requests to avoid or minimize conflicts. We present a suite of transactional schedulers: *Bi-interval*, *Commutative Requests First* (CRF), *Reactive Transactional Scheduler* (RTS), *Dependency-Aware Transactional Scheduler* (DATS), *Scheduling-based Parallel Nesting* (SPN), *Cluster-based Transactional Scheduler* (CTS), and *Locality-aware Transactional Scheduler* (LTS). The schedulers consider Herlihy and Sun's dataflow execution model, where transactions are immobile and objects are migrated to invoking transactions, relying on directory-based cache-coherence protocols to locate and move objects. Within this execution model, the proposed schedulers target different DTM models.

Bi-interval considers the single object copy DTM model, and categorizes concurrent requests into read and write intervals to maximize the concurrency of read transactions. This allows an object to be simultaneously sent to read transactions, improving transactional makespan. We show that Bi-interval improves the makespan competitive ratio of DTM without such a scheduler to $O(log(N))$ for the worst-case and $\theta(log(N-k)$ for the average-case, for N nodes and k read transactions. Our implementation reveals that Bi-interval enhances transactional throughput over the no-scheduler case by as much as $1.71\times$, on average.

CRF considers multi-versioned DTM. Traditional multi-versioned TM models use multiple object versions to guarantee commits of read transactions, but limit concurrency of write transactions. CRF relies on the notion of *commutative* transactions, i.e., those that ensure consistency of the shared data-set even when they are validated and committed concurrently. CRF detects conflicts between commutative and non-commutative write transactions and then schedules them according to the execution state, enhancing the concurrency of write transactions. Our implementation shows that transactional throughput is improved by up to $5\times$ over a state-of-the-art competitor (DecentSTM).

RTS and DATS consider transactional nesting in DTM, and focus on the *closed* and *open* nesting models, respectively. RTS determines whether a conflicting outer transaction must be aborted or enqueued according to the level of contention. If a transaction is enqueued, its closed-nested transactions do not have to retrieve objects again, resulting in reduced communication delays. DATS's goal is to boost the throughput of open-nested transactions by reducing the overhead of running expensive compensating actions and acquiring/releasing abstract locks when the outer transaction aborts. The contribution of DATS is twofold. First, it allows *commutable* outer transactions to be validated concurrently and allows non-commutable outer transactions – depending on their inner transactions – to be committed before others without dependencies. Implementations reveal effectiveness: RTS and DATS improve throughput (over the no-scheduler case), by as much as $1.88\times$ and $2.2\times$, respectively.

SPN considers parallel nested transactions in DTM. The idea of parallel nesting is to execute the inner transactions that access different objects concurrently, and execute the inner transactions that access the same objects serially, increasing performance. However, the parallel nesting model may be ineffective if all inner transactions access the same object due to the additional overheads needed to identify both types of inner transactions. SPN avoids this overhead and allows inner transactions to request objects and to execute them in parallel. Implementations reveal that SPN outperforms non-parallel nesting (i.e., closed nesting) by up to 3.5× and 4.5× on a micro-benchmark (bank) and the TPC-C transactional benchmark, respectively.

CTS considers the replicated DTM model: object replicas are distributed across clusters of nodes, where clusters are determined based on inter-node distance, to maximize locality and fault-tolerance, and to minimize memory usage and communication overhead. CTS enqueues transactions that are aborted due to early validation over clusters and assigns their backoff times, reducing communication overhead. Implementation reveals that CTS improves throughput over competitor replicated DTM solutions including GenRSTM and DecentSTM by as much as 1.64×, on average.

LTS considers the genuine partial replicated DTM model. In this model, LTS exploits locality by: 1) employing a transaction scheduler, which enables/disables object ownership changes depending on workload fluctuations, and 2) splitting hot-spot objects into multiple replicas for reducing contention. Our implementation reveals that LTS outperforms state-of-the-art competitors (Score and CTS) by up to 2.6× on micro-benchmarks (Linked List and Skip List) and by up to 2.2× on TPC-C.

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.

To my parents, wife, and son

Acknowledgments

I would like to gratefully acknowledge the supervision of my advisor, Dr. Binoy Ravindran during my Ph.D. study, for his enthusiasm, inspiration, and his great efforts to guide my research from the start. Particularly, when I experienced the hardest time during my study, I would not overcome the frustration and stress without his encouragement and support. This dissertation could not have been completed without his constant support.

Many thanks to the rest of my committee: Dr. Robert P. Broadwater, Dr. Paul E. Plassmann, Dr. Anil Vullikanti and Dr. Maurice Herlihy, for their invaluable advice and comments during my preliminary and defense exams. It is a great honor to have them serving in my committee. In addition, I would like to thank all my colleagues in Systems Software Research Group, who provided my great environment for collaboration and discussion. I especially thank Dr. Roberto Palmieri and all the TM members: Bo Zhang, Alexandru Turcu, Mohamed Mohamedin, Mohamed Saad, Ahmed Elasyed, Sachin Hirve, and Aditya Dhoke for helpful discussion and comments. During the two years at NDSSL (Network Dynamic and Simulation Science Laboratory), I was extremely fortunate to be able to work with Dr. Anil Vullikanti, Dr. Achla Marathe, Dr. Madhav V. Marathe and my NDSSL colleagues: Fei Huang, Balaaji Sunapanasubbiah, Guanhong Pei, and Sudip Saha. Their warm suggestions and help made me never feel alone in this long journey.

Last but not least, thank all my family members for their love and support. I am grateful to my parents, who always did their best in supporting my education from the childhood, and suggested me the correct direction to make my dream come true. Finally, my wife, Eunseo Kwon, devoted her love and support to me through the ups and downs over the past years. It is difficult to overstate my gratitude to her for being such a wonderful wife.

This dissertation is dedicated to all the people who helped me and are helping me all the way.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Transactional Memory

Lock-based synchronization is inherently error-prone. For example, coarse-grained locking, in which a large data structure is protected using a single lock is simple and easy to use, but permits little concurrency. In contrast, with fine-grained locking, in which each component of a data structure (e.g., a hash table bucket) is protected by a lock, programmers must acquire only necessary and sufficient locks to obtain maximum concurrency without compromising safety, and must avoid deadlocks when acquiring multiple locks. Both these situations are highly prone to programmer errors. The most serious problem with locks is that they are not easily *composable*—i.e., combining existing pieces of software to produce different functionality is not easy. This is because, lock-based concurrency control is highly dependent on the order in which locks are acquired and released. Thus, it would be necessary to expose the internal implementation of existing methods, while combining them, in order to prevent possible deadlocks. This breaks encapsulation, and makes it difficult to reuse software.

Transactional memory (TM) is an alternative synchronization model for shared memory data objects that promises to alleviate the difficulties of lock-based synchronization (i.e., scalability, programmability, and composability issues). As TM code is composed of read/write operations on shared objects, it is organized as *memory transactions*, which optimistically execute, while logging any changes made to accessed objects. Two transactions *conflict* if they access the same object and one access is a write. When that happens, a contention manager (CM) resolves the conflict by aborting one and allowing the other to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately, after rolling-back the changes. Sometimes, a *transactional scheduler* is also used, which determines an ordering of concurrent transactions so that conflicts are either avoided altogether or minimized.

In addition to a simple programming model, TM provides performance comparable to fine-grained locking [1] and is composable. TM for multiprocessors has been proposed in hardware (HTM) [2, 3, 4, 5, 6], in software (STM) [7, 8, 9, 10, 11], and in hardware/software combination [12, 13, 14, 15].

With a single copy for each object, i.e., *single-version* STM (SV-STM), when a read/write conflict occurs between two transactions, the contention manager resolves the conflict by aborting one and allowing the other to commit, thereby maintaining the consistency of the (single) object version. SV-STM is simple, but suffers from large number of aborts [16]. In contrast, with multiple versions for each object, i.e., *multi-versioning* STM (MV-STM), unnecessary (or *spare*) aborts of transactions that could have been committed without violating consistency are avoided [17]. Unless a conflict occurs between operations accessing a shared object, MV-STM allows the corresponding transactions to read the object's old versions, enhancing concurrency. MV-STM has been extensively studied for multiprocessors [18, 16, 19].

Many libraries or third-party software contain atomic code, and application developers often desire to group such code, with user, other library, or third-party (atomic) code into larger atomic code blocks. This can be accomplished by nesting all atomic code within their enclosing code, as permitted by the inherent composability of TM. But doing so — i.e., *flat nesting* — results in large monolithic transactions, which limits concurrency: when a large monolithic transaction is aborted, all nested transactions are also aborted and rolled back, even if they don't conflict with the outer transaction.

Further, in many nested settings, programmers desire to respond to the failure of each nested action with an action-specific response. This is particularly the case in distributed systems—e.g., if a remote device is unreachable or unavailable, one would want to try an alternate remote device, all as part of a top-level atomic action. Furthermore, inadequate performance of a nested third-party or library code must often be circumvented (e.g., by trying another nested code block) to boost overall application performance. In these cases, one would want to abort a nested action and try an alternative, without aborting the work accomplished so far (i.e., aborting the top-level action).

Three types of nesting have been studied in TM: *flat*, *closed*, and *open* [20]. If an inner transaction I is *flat-nested* inside its outer transaction A, A executes as if the code for I is inlined inside A. Thus, if I aborts, it causes A to abort. If I is *closed-nested* inside A, the operations of I only become part of A when I commits. Thus, an abort of I does not abort A, but I aborts when A aborts. Finally, if I is *open-nested* inside A, then the operations of I are not considered as part of A. Thus, an abort of I does not abort A, and vice versa.

1.2 Distributed Transactional Memory

The challenges of lock-based concurrency control are exacerbated in distributed systems, due to the additional complexity of multicomputer concurrency. Distributed TM (or DTM) has been similarly motivated as an alternative to distributed lock-based concurrency control. DTM can be classified based on the system architecture: cache-coherent DTM (or cc DTM) [21, 22, 23] and cluster DTM [24, 25, 26]. 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.

Most cc DTM works consider Herlihy and Sun's dataflow execution model [21], in which transactions are immobile and objects move from node to node to invoking transactions. cc DTM uses a cache-coherence protocol, often directory-based [27, 21, 22], to locate and move objects in the network, satisfying object consistency properties.

Similar to multiprocessor TM, DTM provides a simple distributed programming model (e.g., locks are entirely precluded in the interface), and performance comparable or superior to distributed lock-based concurrency control [24, 25, 26, 28, 23, 29].

1.3 Transactional Scheduling

As mentioned before, a complementary approach for dealing with transactional conflicts is transactional scheduling. Broadly, a transactional scheduler determines the ordering of concurrent transactions so that conflicts are either avoided altogether or minimized. Two kinds of transactional schedulers have been studied in the past: reactive [7, 30] and proactive [31, 32]. When a conflict occurs between two transactions, the contention manager determines which transaction wins or loses, and then the loosing transaction aborts. Since aborted transactions might abort again in the future, *reactive schedulers* enqueue aborted transactions, serializing their future execution [7, 30]. *Proactive schedulers* take a different strategy. Since it is desirable for aborted transactions to be not aborted again when re-issued, proactive schedulers abort the loosing transaction with a backoff time, which determines how long the transaction is stalled before it is re-started [31, 32]. Both reactive and proactive transactional schedulers have been studied for multiprocessor TM. However, they have not been studied for DTM, which is the focus of this dissertation.

We now motivate and overview the seven different transactional schedulers developed in the dissertation. The schedulers target data-flow cc DTM and are called *Bi-interval* [28], *commutative requests first* (CRF) [33], *reactive transactional scheduler* (RTS) [34], *dependency-aware transactional scheduler* (DATS) [35], *scheduling-based parallel nesting* (SPN), *cluster-based transactional scheduler* (CTS) [36], and *locality-aware transactional scheduler* (LTS) [37].

Scheduling in single-version DTM. We first consider the single object copy DTM model (i.e., SV-STM). A distributed transaction typically has a longer execution time than a multiprocessor transaction, due to communication delays that are incurred in requesting and acquiring objects, which increases the likelihood for conflicts and thus degraded performance [32]. We present a novel transactional scheduler called Bi-interval [28] that optimizes the execution order of transactional operations to minimize conflicts. Bi-interval focuses on read-only and read-dominated workloads (i.e., those with only early-write operations), which are common transactional workloads [38]. Read transactions do not modify the object; thus transactions do not need exclusive object access. Bi-interval categorizes concurrent requests for a shared object into read and write intervals to maximize the concurrency of read transactions. This reduces conflicts between read transactions, reducing transactional execution times. Further, it allows an object to be simultaneously sent to nodes of read transactions, thereby reducing the total object traveling time.

Scheduling in multi-versioned DTM. We then consider multi-versioned STM (i.e., MV-STM). Unless a conflict occurs between operations that access a shared object, MV-STM allows the corresponding transactions to read the object's old versions, potentially enhancing concurrency. MV-STM has been extensively studied for multiprocessors [16, 19] and also for distributed systems [39]. MV-STM uses *snapshot isolation* (SI), which is weaker than serializability [40]. A transaction executing under SI operates on a snapshot taken at the start of the transaction. The transaction successfully commits if the objects updated by the transaction have not been changed externally since the snapshot was taken, guaranteeing that all read transactions will see a consistent snapshot. Many past efforts [40, 41, 42] have used SI for improving performance in centralized and distributed TM environments. Even though SI allows greater concurrency among transactions than serializability, a write-write conflict under SI causes the transaction to abort. In write-intensive workloads, this conflict cannot be avoided because the concurrency of write transactions may violate SI.

To increase performance in multi-version DTM, the dissertation presents a scheduler called CRF that allows multiple conflicting transactions to commit concurrently without violating SI, in order to enhance the concurrency of write transactions. CRF increases the concurrency of write transactions by exploiting the notion of commutative operations. Two operations are called *commutative* if executing them sequentially in either order transitions the objects accessed to the same state and returns the same values [43]. A very intuitive example is when two operations, say, $call1(X)$ and $call2(X)$, access the same object X, but different fields of X. CRF checks whether write operations are commutative and allows them to validate and commit simultaneously. Unlike past STM works that exploit high concurrency based on the commutativity property, CRF maintains a scheduling queue to identify commutative and non-commutative transactions, and allows all commutative transactions to commit first than the others, maximizing their concurrency. To support a broad range of applications, CRF allows programmers to explicitly specify non-commutative operations.

Scheduling nested transactions. We now turn our attention to scheduling nested transactions. In the flat and closed nesting models, if an outer transaction with multiple nested transactions aborts due to a conflict, the outer and inner transactions will restart and request all objects regardless of which object caused the conflict. Even though the aborted transactions are enqueued to avoid conflicts, the scheduler serializes the aborted transactions to reduce the contention on only the object that caused the conflict. With nested transactions, this may lead to heavy contention because all objects have to be retrieved again.

We first consider scheduling closed-nested transactions, which are more efficient than flat nesting and guarantees serialization [44]. RTS considers both aborting or enqueuing a parent transaction including closed-nested transactions. RTS decides which transaction is aborted or enqueued to protect its nested transactions according to a contention level, and assigns a backoff time to the enqueued transaction to boost throughput.

We then consider scheduling open-nested transactions. The open-nesting model [45] ensures abstract serializability for open-nested transactions (which requires abstract locking), and serializability for outer transactions (using a compensation strategy). An outer transaction commits multiple objects in a single step if there is no conflict, but its open-nested transactions do multiple per-object commits. In DTM, abstract locking incurs communication delays to remotely acquire and release the locks. If multiple inner transactions commit, locks must be acquired and released for each open-nested transaction, degrading performance.

Moreover, if an outer transaction (with open-nested inner transactions) aborts, all of its (now committed) open-nested transactions must be aborted and their actions must be undone to ensure transaction serializability. Thus, with the open nesting model, programmers must describe a *compensating* action for each open-nested transaction [44]. When outer transactions increasingly encounter conflicts after greater number of their open-nested transactions have committed, it will increase executions of compensating actions, degrading overall performance. With closed nesting, since closed-nested transactions are not committed until the outer transaction commits (nested transactions' changes are visible only to the outer), no undo is required. Thus, open nesting may perform worse than closed nesting in high contention.

To boost the performance of open-nested transactions in DTM, DATS reduces conflicts of outer transactions and the number of abstract locks of inner transactions, which in turn, minimizes communication overheads and compensating actions. In order to do that, DATS relies on the notions of *commutable transactions*[1] and *transaction dependencies*. Two transactions are defined as commutable if they ensure consistency of the shared data-set even if validated and committed concurrently upon a conflict. An outer transaction is said to be dependent on its inner transactions if *(i)* the inner transactions access the outer transaction's write-set for performing local computations or *(ii)* the results of the outer transaction are used to decide whether or not to invoke an inner transaction.

DATS detects commutable transactions, and validates and commits them, avoiding useless aborts. For non-commutable transactions, DATS determines the degree to which each outer transaction depends on its inner transactions and schedules the outer transaction with the highest dependency to commit before other outer transactions with lower or no dependencies. Committing this outer transaction prevents its dependent inner transactions from aborting, and reduces the number of compensating actions. Moreover, even though the other outer transactions abort, this abort does not affect their independent inner transactions, which reduces the number of compensating actions and abstract locks that must be released and re-acquired without violating the correctness of the object.

[1]Note that, the definition of "commutable" is the same as that of "commutative", but we use the phrase "commutable" for a transaction to identify commutative operations and transactions.

Scheduling parallel nested transactions. In order to enhance the throughput of nested transactions, the parallel nesting model has been proposed [46, 47, 48]. Nested parallelism within an outer transaction exacerbates the overhead for initializing, synchronizing, and balancing inner transactions (due to the additional conflict detection and rollback needed for the inner transactions) [46]. Existing efforts have focused on this challenge and present cost-effective algorithms, but are limited to centralized systems.

We consider closed-nested transactions for exploiting nested parallelism in DTM. To initiate parallel nesting, extra overheads to identify transactions to be parallelized or serialized are inevitable. Due to the nature of closed-nesting, an outer transaction has to wait until all parallel inner transactions commit. If all nested inner transactions are serialized in the worst case, parallel nesting may not be effective. Motivated by this, we consider scheduling inner transactions to enhance parallel nesting in DTM, and develop a scheduler called SPN. SPN speculatively executes all closed-nested transactions in parallel. When conflicts are detected, the scheduler assigns a back off time to each nested transaction to reduce the likelihood for future conflicts.

Scheduling in replicated 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 – 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) [49, 50, 24, 51, 52]. 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 [23]. Thus, directly applying cluster DTM replication solutions to data-flow cc DTM may not yield similar performance.

We therefore consider a cluster-based object replication model for data-flow cc DTM. In this model, nodes are grouped into clusters based on node-to-node distances: nodes which are closer to each other are grouped into the same cluster; nodes which are farther apart are grouped into different clusters. Objects are replicated such that each cluster contains at least one replica of each object, and the memory of multiple nodes is used to reduce the possibility of object loss, thereby avoiding expensive brute-force replication of all objects on all nodes. The CTS scheduler, schedules transactions in this model for high performance. Each cluster has an object owner (node) for scheduling transactions. On each object owner, CTS enqueues live transactions and identifies some of the transactions that must be aborted to avoid future conflicts, thereby increasing the concurrency of the other transactions.

We also enhance locality for high performance in data-flow DTM. Majority of the DTM protocols proposed in the literature use the control-flow execution model [42, 53]. This is because, the problem of locating objects in a distributed system is inherently a barrier for scalability as it involves a distributed protocol that is typically costlier than the replication protocol itself. In fact, when each object is bound to a specific owner, control-flow protocols can rely on a deterministic, consistent hash function [54], which allows transactions to locally compute the node responsible for maintaining the object, without involving any inter-node communication. However, it is clear from the deterministic nature of this function that it does not allow changing the object ownership or biasing the initial object location. Without those features, the replication protocol cannot optimize the object location for the workload at hand. This problem is inherently solved in the data-flow approach, where objects can be moved to nodes that more frequently request and update such objects.

Motivated by these observations, we propose LOCAL-FLOW, a partial replication DTM protocol for the data-flow model that is optimized for applications with inherent time locality on their object accesses. LOCAL-FLOW is *genuine* [55, 56]: only replicas involved during transactional execution participate in the commit phase. The protocol ensures *1-copy-serializability* [57] by acquiring locks on updated objects at commit time (using two-phase commit) and validating the accessed objects after lock acquisition.

To limit the overhead of locating and moving the accessed objects through nodes, LOCAL-FLOW defines two logical execution phases for each object: phase *A*, where object ownership is changed at commit time; and phase *B*, where the object ownership is not modified after commit.[2] During phase *A*, transactions locate objects by simply querying a local structure without invoking any distributed protocol. Phase *B* is needed for addressing workload fluctuations. When a non-optimal placement for an object o_i maintained by node n_i is detected, the node performing the majority of requests for o_i, say n_j, is determined, and o_i's ownership is transferred to n_j (transitioning o_i's phase from *A* to *B*).

The LTS scheduler, embedded in LOCAL-FLOW, is responsible for managing concurrent object requests from processing nodes. LTS monitors the key performance parameters needed for detecting the effectiveness of the current object placement and triggers the transition of misplaced objects from phase *A* to phase *B*.

LOCAL-FLOW's key aspect is the management of system "hot spots". The protocol defines a configuration parameter, called *field replication degree* (FRD), which is used for splitting hot spot objects to alleviate their contention (splitting reduces the invalidation rate on them). In general, an application defines a set of transactional classes, which represent the profiles of transactions injected in the system. Each class is typically interested in only a subset of the fields of an object. If two different classes access the same object but different fields, it is detected as a conflict, and is resolved by aborting one of the two transactions. Moreover, if this object is heavily accessed, the overall concurrency is degraded. To mitigate this, LTS detects the presence of each hot-spot object and splits its fields, only when FRD > 1, onto a subset of the replicas that are already responsible for storing the object (i.e., the object owners).

[2]Note that, the phases are only logical and are not time-dependent. LOCAL-FLOW's underlying model is asynchronous.

1.4 Summary of Research Contributions

We now summarize our contributions.

- The Bi-interval scheduler groups concurrent requests into read and write intervals to exploit concurrency of read transactions. Write transactions are serialized according to shortest object moving times. Bi-interval improves the makespan competitive ratio of DTM (without such a scheduler) from $O(N)$ to $O(log(N))$, for N nodes. Also, Bi-interval yields an average-case makespan competitive ratio of $\Theta(log(N-k))$ for k read transactions. Bi-interval improves throughput over the no-scheduler case by as much as 1.77$\sim$ and 1.65$\times$ speedup under low and high contention, respectively.

- The CRF scheduler minimizes the abort rate and increases the concurrency of write transactions in multi-version DTM. CRF determines the commutativity of write operations, and executes all commutative operations concurrently in a commutative epoch and non-commutative operations in the next epoch. CRF improves throughput by up to 5$\times$ over DecentSTM [52], a state-of-the-art multi-version DTM.

- The RTS scheduler reduces the aborts of closed-nested transactions to improve performance. In order to do that, RTS heuristically determines transactional contention levels to decide whether a live parent transaction aborts or enqueues, and a backoff time that determines how long a live parent transaction waits. RTS enhances throughput (over the no-scheduler case) at high and low contention, by as much as 1.53 (53%) $\sim$ 1.88 (88%) $\times$ speedup over closed-nesting (without scheduling), respectively.

- The DATS scheduler detects commutable open-nested transactions and commits them simultaneously, avoiding useless aborts. Moreover, DATS identifies the degree to which an outer transaction depends on its inner transactions and allows the outer transaction with the highest dependency to commit, reducing unnecessary compensating actions and abstract locks. DATS improves throughput by up to 1.7$\times$ on micro-benchmarks and by up to 2.2$\times$ on TPC-C over open-nested DTM without DATS.

- The SPN scheduler improves performance of nested transactions by speculative parallel nesting: the scheduler identifies inner transactions that access different objects and executes them in parallel. Even if all inner transactions access the same object and are serialized, SPN minimizes the initial overhead to identify them. SPN improves throughput by up to 3.5$\times$ on micro-benchmarks and by up to 4.5$\times$ on TPC-C over closed-nested DTM without SPN.

- The CTS scheduler improves the performance of replicated DTM by avoiding brute force replication of all objects over all nodes to minimize communication overhead. In addition, CTS identifies the transactions that must be aborted in advance to enhance the concurrency of other transactions, reducing significant number of potential future conflicts. CTS enhances throughput by up to 1.73$\times$ (on average) over state-of-the-art replicated DTMs including GenRSTM [51] and DecentSTM [52].

- The LTS scheduler improves the performance of data flow, partial replication-based DTM by exploiting locality: object ownership is dynamically changed depending on workload fluctuations, and hot spot objects are split into multiple replicas to reduce contention. LTS outperforms CTS by up to 2.6× on micro-benchmarks and by up to 2.2× on TPC-C. Moreover, LTS outperforms the Score partial replication DTM protocol [42] by up to 1.55× on TPC-C.

1.5 Organization

The rest of the dissertation is organized as follows. We overview past and related work in Chapter 2. We outline the basic preliminaries and system models in Chapter 3. Chapters 4, 5, 6, 7, 8, 9, and 10 describe the Bi-inteval, CRF, RTS, DATS, SPN, CTS, and LTS schedulers, respectively. Each chapter describes the motivation, scheduler design, an illustrative example, algorithm descriptions, scheduler properties, and experimental evaluations. We conclude and describe future work in Chapter 11.

Chapter 2

Past and Related Work

2.1 Distributed Transactional Memory

Herlihy and Sun proposed distributed STM [21]. They present a dataflow model, where transactions are immobile, and objects are dynamically migrated to invoking transactions. Object conflicts and object consistency are managed and ensured, respectively, by contention management and cache coherence protocols. In [21], they present a cache-coherence protocol, called Ballistic. Ballistic models the cache-coherence problem as a distributed queuing problem, due to the fundamental similarities between the two, and uses the Arrow queuing protocol [27] for managing transactional contention. Ballistic's hierarchical structure degrades its scalability—e.g., whenever a node joins or departs the network, the structure has to be rebuilt. This drawback is overcome in Zhang and Ravindran's Relay cache-coherence protocol [22], which improves scalability by using a peer-to-peer structure. They also present a class of location-aware cache-coherence (or LAC) protocols [58], which improve the makespan competitive ratio, with the optimal Greedy contention manager [59].

While these efforts focused on distributed STM's theoretical properties, other efforts developed implementations. In [49], Bocchino *et. al.* decompose a set of existing cache-coherent TM designs into a set of design choices, and select a combination of such choices to support TM for commodity clusters. They show how remote communication can be aggregated with data communication to obtain high scalability. In [60], Manassiev *et. al.* present a page-level distributed concurrency control algorithm, which automatically detects and resolves conflicts caused by data races for distributed transactions accessing shared data structures. Kotselidis *et. al.* present the DiSTM distributed TM framework for easy prototyping of TM cache-coherence protocols.

Couceiro *et. al.* present the D^2STM for distributed systems [24]. Here, an STM is replicated on distributed system nodes, and strong transactional consistency is enforced at transaction commit time by a non-blocking distributed certification scheme. Romano *et. al.* extend distributed TM for Web services [25], and cloud platforms [26].

In [25], they present a distributed TM architecture for Web services, where application's state is replicated across distributed system nodes. Distributed TM ensures atomicity and isolation of application state updates, and consistency of the replicated state. In [26], they show how distributed TM can increase the programming ease and scalability of large-scale parallel applications on cloud platforms.

2.2 Multi-Version STM

MV-STM has been extensively studied for multiprocessors. MV increases concurrency by allowing transactions to read older versions of shared data, thereby minimizing conflicts and aborts.

Ramadan *et. al.* present dependency-aware transactional memory (DATM) [61], where transaction execution is interleaved, and show substantially more concurrency than two-phase locking. DATM manages dependency of transactions between live transactions, resulting in concurrency increases of up to 39% and reducing transaction restarts by up to 94%.

Moore *et. al.* present Log-based transactional memory (LogTM) that makes commits fast by storing old versions to a log. LogTM provides fast conflict detection and commit, and is evaluated on 32 multiprocessors, resulting in only 1-2% transaction aborts.

A single-version model supporting permissiveness was first introduced by Guerraoui *et. al.* [62]. An STM satisfies π-permissiveness for a correctness criterion π unless every history accepted by that STM violates π. The notion of online-π-permissiveness, presented in [17], does not allow transactions to abort unless live transactions violate π. In [16], Perelman *et. al.* propose the concept of MV permissiveness, which ensures that read-only transactions never abort in MV-STM. Maintaining all possible multiple versions that might be needed wastes memory. Thus, they define a GC property called *useless prefix* that only keeps multiple versions that some existing read-only transactions may need.

In [17], Keidar and Perelman identify what kinds of *spare aborts* can or cannot be eliminated in MV, and present a Γ-AbortAvoider algorithm that maintains a *precedence graph* (PG) for avoiding spare aborts. They show that an STM with Γ-AbortAvoider satisfies Γ-opacity and online Γ-opacity permissiveness.

Transactional schedulers have been studied for SV-STM. Their purpose is fundamentally similar to that of MV-STM, but the approach is functionally different. Since versions might be needed by live transactions in the future, MV-STM keeps multiple versions of shared objects. Since aborted transactions might be aborted again in the future, scheduling keeps aborted transactions.

2.3 Nested Transactions

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [63]. This work focused on the popular two-phase locking protocol and extended it to support nesting. Also, it proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection.

Open nesting also originates in the database community [64], and was extensively analyzed in the context of undo-log transactions [65]. In these works, open nesting is used to decompose transactions into multiple levels of abstraction, and maintain serializability on a level-by-level basis. One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [66]. They describe the semantics of transactional operations in terms of system states, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written. They also provide sketches for several possible HTM implementations, which work by extending existing cache coherence protocols. Moss further focuses on open nested transactions in [20], explaining how using multiple levels of abstractions can help in differentiating between fundamental and false conflicts and therefore improve concurrency.

Moravan et al. [67] implement closed and open nesting in their previously proposed LogTM HTM. They implement the nesting models by maintaining a stack of log frames, similar to the run-time activation stack, with one frame for each nesting level. Hardware support is limited to four nesting levels, with any excess nested transactions flattened into the inner-most sub-transaction. In this work, open nesting was only applicable to a few benchmarks, but it enabled speedups of up to 100

Agrawal et al. [68] combine closed and open nesting by introducing the concept of transaction ownership. They propose the separation of TM systems into transactional modules (or Xmodules), which own data. Thus, a sub-transaction would commit data owned by its own Xmodule directly to memory using an open-nested model. However, for data owned by foreign Xmodules, it would employ the closed nesting model and would not directly write to the memory.

Even though TM promises to make concurrent programming easy to a wide programmer community, nested transactions are not allowed to run in parallel. This is an important obstacle to the central goal of TM. Due to this issue, parallel nesting has been studied [46, 47, 48]

Agrawal et. al. [47] propose XCilk, a runtime-system design supporting transactions that themselves can contain nested parallelism and nested transactions. XClik shows the first theoretical performance bound on a TM system that supports transactions with nested parallelism. Baek et. al. [46] present NestTM supporting closed nested parallel transactions. NestTM uses eager version management and word-granularity conflict detections targeting the state and runtime overheads of nested parallel transactions. Its performance has been evaluated using STAMP [69].

2.4 Replication for DTM

Zhang and Ravindran [70] propose a quorum-based replication (QR) framework for DTM to enhance availability of objects without incurring high communication overhead. All nodes based on QR have to hold all objects, and one-copy serializability is ensured using a flooding algorithm.

D^{2}STM [24] relies on a commit-time atomic broadcast-based distributed validation to ensure global consistency. Motivated by database replication schemes, distributed certification based on atomic broadcast [71] avoids the costs of replica coordination during the execution phase and runs transactions locally in an optimistic fashion.

Carvalho *et. al.* present asynchronous lease certification (ALC) DTM replication scheme in [72], which overcomes some drawbacks of atomic broadcast-based replication [24]. ALC reduces the replica coordination overhead and avoids unnecessary aborts due to conflicts at remote nodes using asynchronous leases. ALC relies on uniform reliable broadcast [71] to exclusively disseminate the *writeset*s, which reduces inter-replica synchronization overhead.

Manassiev *et al.* present a page-level distributed multiversioning algorithm for cluster DTM [60]. In this algorithm, page differences are broadcast to all other replicas, and a transaction commits successfully upon receiving acknowledgments from all nodes. A central timestamp is employed, which allows only a single update transaction to commit at a time.

Kotselidis *et al.* present the DiSTM cluster DTM framework in [50]. Under the TCC protocol [2], DiSTM induces large traffic overhead at commit time, as a transaction broadcasts its read/write sets to all other transactions, which compare their read/write sets with those of the committing transaction. Using lease protocols [73], this overheard is eliminated. However, they also show that an extra validation step is added to the master node as well as bottlenecks are created under high contention because of acquiring and releasing the leases.

In [42], the authors present Score, a partial replication protocol based on control-flow. Score ensures one-copy serializability on a multi-version model, allowing read-only transactions to commit without performing any validation. The protocol yields the best performance when objects are manually placed in the system, and does not provide any mechanism for changing those locations when workload changes.

Megastore [74] is a NoSQL datastore-based storage system that is locality-aware. The system partitions data for enhancing locality: data is partitioned into a set of entity groups, and each group is assigned to the region or continent from which it is most accessed. Objects are placed manually, based on geographical assumptions.

P-Store [56] is a genuine partially replicated key-value store, where transactions execute on one or more sites and are then certified to guarantee serializability. The protocol offers better scalability than previous solutions, but does not consider locality or workload dynamics.

None of the replication models for cc and cluster DTM consider transactional scheduling. Also, as mentioned before, broadcasting transactional read/write sets or memory differences as done for cluster DTM is inherently non-scalable for cc DTM, as messages transmitted grow quadratically with the number of nodes. CTS and LTS consider a partial replicated model, increasing scalability and performance.

2.5 Transactional Scheduling

Transactional scheduling has been explored in a number of multiprocessor STM efforts [75, 76, 77, 31, 7, 30]. However, none of transactional schedulers considers DTM.

In [76], Dragojević *et. al.* describe an approach that dynamically schedules transactions based on their predicted read/write access sets. In [77], Ansari *et. al.* discuss the Steal-On-Abort transaction scheduler, which queues an aborted transaction behind the non-aborted transaction, and thereby prevents the two transactions from conflicting again.

Yoo and Lee present the adaptive transaction scheduler (ATS) [31] that adaptively controls the number of concurrent transactions based on the contention intensity: when the intensity is below a threshold, the transaction begins normally; otherwise, the transaction stalls and does not begin until dispatched by the scheduler.

Dolev *et. al.* present the CAR-STM scheduling approach [7], which uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts.

Blake, Dreslinski, and Mudge propose the proactive transactional scheduler (PTS) in [32]. PTS detects hot spots of contention that can degrade performance, and proactively schedules affected transactions around the hot spots. Evaluation on the STAMP benchmark suite [69] shows their scheduler outperforming a backoff-based policy by an average of 85%.

Attiya and Milani present the BIMODAL scheduler [30], which targets read-dominated and bimodal (i.e., those with only early-write and read-only) workloads. BIMODAL alternates between "writing epochs" and "reading epochs" during which writ and read transactions are given priority, respectively, ensuring greater concurrency for read transactions.

Chapter 3

Preliminaries and System Model

We consider a distributed system which consists of a set of nodes $N = \{n_1, n_2, \cdots\}$ that communicate with each other by message-passing links over a communication network. Similar to [21], we assume that the nodes are scattered in a metric space. The metric $d(n_i, n_j)$ is the distance between nodes n_i and n_j, which determines the communication cost of sending a message from n_i to n_j.

3.1 Distributed Transactions

A *distributed transaction* performs operations on a set of *shared objects* in a distributed system, where nodes communicate by message-passing links. Let $O = \{o_1, o_2, \ldots\}$ denote the set of shared objects. A transaction T_i is in one of three possible statuses: *live*, *aborted*, or *committed*. If an aborted transaction retries, it preserves the original starting timestamp as its starting time.

We consider Herlihy and Sun's dataflow distributed STM model [21], where transactions are immobile, and objects move from node to node. In this model, each node has a TM proxy that provides interfaces to its application and to proxies at other nodes. When a transaction T_i at node n_i requests object o_j, the TM proxy of n_i first checks whether o_j is in its local cache. If the object is not present, the proxy invokes a distributed cache coherence protocol (cc) to fetch o_j in the network. Node n_k holding object o_j checks whether the object is in use by a local transaction T_k when it receives the request for o_j from n_i. If so, the proxy invokes a contention manager to mediate the conflict between T_i and T_k for o_j.

When a transaction T_i invokes an operation on object o_j, the cc protocol is invoked by the local TM proxy to locate the current cached copy of o_j. We consider two properties of the DCC. First, when the TM proxy of T_i requests o_j, the cc is invoked to send T_i's read/write request to a node holding a valid copy of o_j in a finite time period. A read (write) request indicates the request for T_i to conduct a read (write) operation on o_j. A valid object copy is defined as a valid version. Thus, a node holding versions of o_j replies with the version corresponding to T_i's request. Second, at any given time, the cc must locate only one copy of o_j in the network and only one transaction is allowed to eventually write to o_j.

3.1.1 Definitions

For the purpose of analysis, we consider a symmetric network of N nodes scattered in a metric space. The metric $d(n_i, n_j)$ is the distance between nodes n_i and n_j, which determines the communication cost of sending a message from n_i to n_j. We consider three different *models*: no replication (NR), partial replication (PR), and full replication (FR) in data-flow DTM to show the effectiveness of Bi-interval, and, RTS, SPN in NR, and CTS in PR.

Definition 1. *Given a scheduler A and N transactions in DTM, $makespan_A^N(Model)$ is the time that A needs to complete N transactions on $Model$.*

Definition 2. *The competitive ratio (CR) of a scheduler A for N transactions in $Model$ is $\frac{makespan_A^N(Modle)}{makespan_{OPT}^N(Model)}$, where OPT is the optimal scheduler.*

Definition 3. *The relative competitive ratio (RCR) of schedulers A and B for N transactions on $Model$ in DTM is $\frac{makespan_A^N(Model)}{makespan_B^N(Model)}$.*

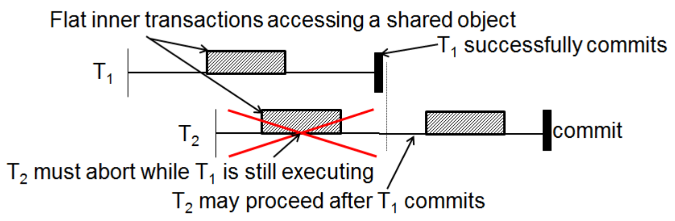
Also, the RCR of model 1 and 2 for N transactions on scheduler A in DTM is $\frac{makespan_A^N(Model1)}{makespan_A^N(Model2)}$. Given schedulers A and B for N transactions, if RCR (i.e., $\frac{makespan_A^N(Model)}{makespan_B^N(Model)}$) < 1, A outperforms B. Thus, RCR of A and B indicates a relative improvement between schedulers A and B if $makespan_A^N(Model) < makespan_B^N(Model)$.

The execution time of a transaction is defined as the interval from its beginning to the commit. In distributed systems, the execution time consists of both communication delays to request and acquire a shared object and the time duration to conduct an operation on a processor, so the local execution time of T_i is defined as γ_i, $\sum_{i=1}^{N} \gamma_i = \Gamma_N$ for N transactions.

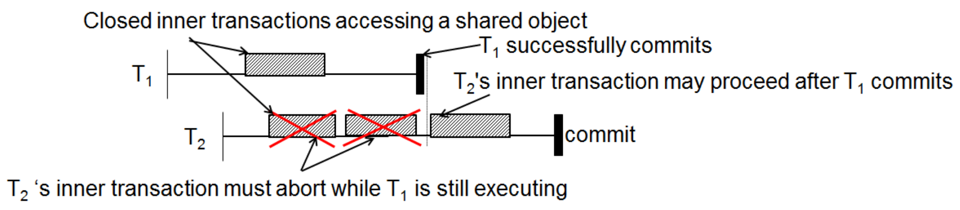
If only a transaction T_i invoking in n_i exists and T_i requests an object from n_j on NR, it will commit without any contention. Thus, $makespan_A^1(NR)$ is $2 \times d(n_i, n_j) + \gamma_i$ under any scheduler A.

3.2 Nested Transactions

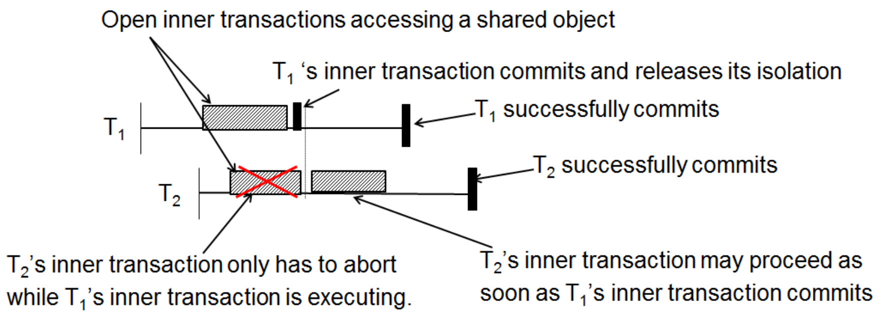
The differences between the nesting models are shown in Figure 3.1, in which there are two transactions containing a nested-transaction. With flat nesting illustrated in Figure 3.1(a), transaction T_2 cannot execute until transaction T_1 commits. T_2 incurs full aborts, and thus has to restart from


Flat inner transactions accessing a shared object
T1 successfully commits
T1
T2
commit
T2 must abort while T1 is still executing
T2 may proceed after T1 commits

(a) Flat Nesting


Closed inner transactions accessing a shared object
T1 successfully commits
T1
T2's inner transaction may proceed after T1 commits
T2
commit
T2 's inner transaction must abort while T1 is still executing

(b) Closed Nesting


Open inner transactions accessing a shared object
T1 's inner transaction commits and releases its isolation
T1
T1 successfully commits
T2
T2 successfully commits
T2's inner transaction only has to abort while T1's inner transaction is executing.
T2's inner transaction may proceed as soon as T1's inner transaction commits

(c) Open Nesting

Figure 3.1: Two Transactions under Flat, Closed and Open Nesting

the beginning. Under closed nesting presented in Figure 3.1(b), only T_2's inner-transaction needs to abort and be restarted while T_1 is still executing. The portion of work T_2 executes before the data-structure access does not need to be retried, and T_2 can thus finish earlier. Under open nesting in Figure 3.1(c), T_1's inner-transaction commits independently of its outer, releasing memory isolation over the shared data-structure. T_2's inner-transaction can therefore proceed immediately, thus enabling T_2 to commit earlier than in both closed and flat nesting.

The flat and closed nested models have a clear negative impact on large monolithic transactions in terms of concurrency. In fact, when a large transaction is aborted all its flat/closed-nested transactions are also aborted and rolled-back, even if they do not conflict with any other transaction. Closed nesting potentially offers better performance than flat nesting because the aborts of closed-nested inner transactions do not affect their outer transactions. However the open-nesting approach outperforms both in terms of concurrency allowed. When an open-nested transaction commits, its modifications on objects become immediately visible to other transactions, allowing those transactions to start using those objects without a conflict, increasing concurrency [66]. In contrast, if the inner transactions are closed- or flat-nested, then those object changes are not made visible until the outer transaction commits, potentially causing conflicts with other transactions that may want to use those objects.

To achieve high concurrency in open nesting, inner transactions have to implement *abstract serializability* [45]. If concurrent executions of transactions result in the consistency of shared objects at an "abstract level", then the executions are said to be abstractly serializable. If an inner transaction I commits, I's modifications are immediately committed in memory and I's read and write sets are discarded. At this time, I's outer transaction A does not have any conflict with I due to memory accessed by I. Thus, programmers consider the internal memory operations of I to be at a "lower level" than A. A does not consider the memory accessed by I when it checks for conflicts, but I must acquire an *abstract lock* and propagates this lock for A. When two operations try to acquire the same abstract lock, the open nesting concurrency control is responsible for managing this conflict (so this is defined "abstract level").

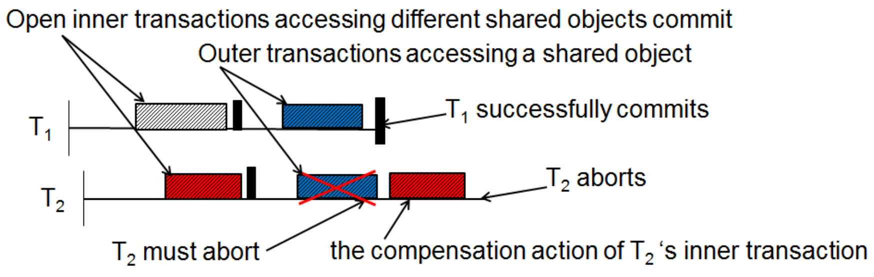

Open inner transactions accessing different shared objects commit
Outer transactions accessing a shared object
T_1
T_1 successfully commits
T_2
T_2 aborts
T_2 must abort
the compensation action of T_2's inner transaction

Figure 3.2: Aborting a Transaction Under Open Nesting

Figure 3.2 shows that transaction T_2 aborts due to a conflict and the compensation action of its inner-transaction is executed. Let us assume that T_1 and T_2's inner transactions access the different object and commit successfully. However, their outer-transactions access the same object and T_2 aborts. The compensating action of T_2's inner-transaction must be executed because the inner-transaction's modification has became visible to other transactions. Even if open-nested transactions provide high concurrency of inner transactions, the open nested model does not always perform better than flat and closed nested models due to a large number of abstract locks and compensation actions [78].

3.3 Atomicity, Consistency, and Isolation

We use the *Transactional Forwarding Algorithm* (TFA) [23] to provide *early validation* of remote objects, guarantee a consistent view of shared objects between distributed transactions, and ensure atomicity for object operations in the presence of asynchronous clocks. TFA is responsible for caching local copies of remote objects and changing object ownership. Without loss of generality, objects export only read and write methods (or operations).

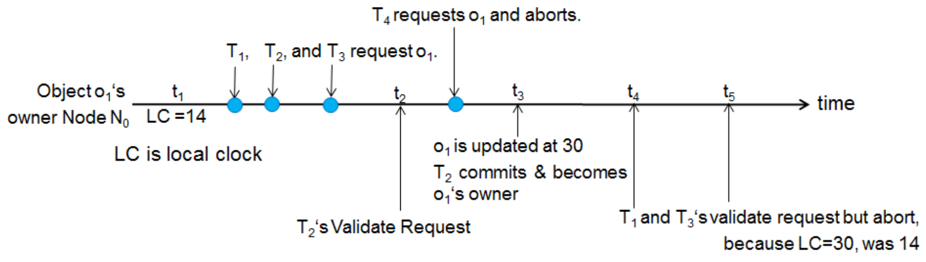

T4 requests o1 and aborts.
T1, T2, and T3 request o1.
Object o1's owner Node N0
t1
LC =14
LC is local clock
t2
t3
t4
t5
time
o1 is updated at 30
T2 commits & becomes o1's owner
T2's Validate Request
T1 and T3's validate request but abort, because LC=30, was 14

Figure 3.3: An Example of TFA

For completeness, we illustrate TFA with an example. In Figure 3.3, a transaction updates object o_1 at time t_1 (i.e., local clock (LC) is 14) and four transactions (i.e., T_1, T_2, T_3, and T_4) request o_1 from the object holder. Assume that T_2 validates o_1 at time t_2 and updates o_1 with LC=30 at time t_3. A validation in distributed systems includes global registration of object ownership. Any read or write transaction (e.g., T_4), which has requested o_1 between t_2 and t_3 aborts. When write transactions T_1 and T_3 validate at times t_1 and t_2, respectively, transactions T_1 and T_3 that have acquired o_1 with LC=14 before t_2 will abort, because LC is updated to 30.

Bi-interval is associated with TFA. RTS and SPN are associated with nested TFA (N-TFA) [79], that is an extension of TFA to implement closed nesting in DTM. DATS is associated with TFA with Open Nesting (TFA-ON) [78], which extends the TFA algorithm [23], to manage open-nested transactions. N-TFA and TFA-ON change the scope of object validations.

The behavior of open-nested transactions under TFA-ON is similar to the behavior of regular transactions under TFA. TFA-ON manages the abstract locks and the execution of commit and compensating actions [78]. To provide conflict detection at the abstract level, an abstract locking mechanism is integrated into TFA-ON. Abstract locks are acquired only at commit time, once the inner transaction is verified to be conflict free at the low level. The commit protocol requests the abstract lock of an object from the object owner and the lock is released when its outer transaction commits. To abort an outer transaction properly, a programmer provides an abstract compensating action for each of its inner transaction to revert the data-structure to its original semantic state. TFA-ON is the first ever implementation of a DTM system with support for open-nested transactions [78].

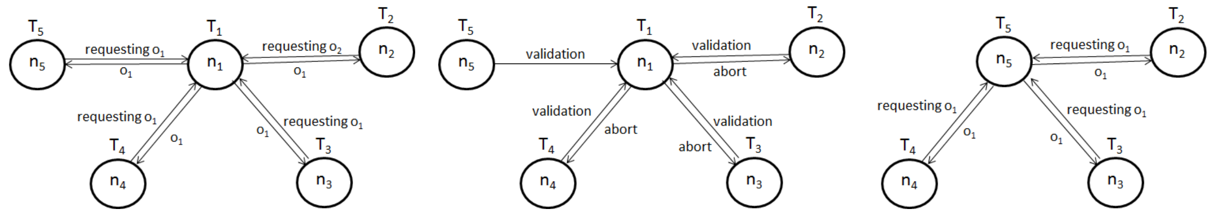
To ensure the consistency of objects, CRF is implemented in MV-TFA, CTS is based on partially replicated TFA, and LTS is associated with LOCAL-FLOW (i.e., a partially replicated data-flow model). We discuss MV-TFA, replicated TFA and LOCAL-FLOW in chapters 5, 9 and 10, respectively.

Chapter 4

The Bi-interval Scheduler

4.1 Motivation

Unlike multiprocessor transactions, data flow-based DTM incurs communication delays in requesting and acquiring objects. Figure 4.1 illustrates a scenario on data flow DTM consisting of five nodes and an object. Figure 4.1(a) shows that nodes n_2, n_3, n_4, and n_5 invoke T_2, T_3, T_4, T_5, respectively and request o_1 from n_1. In Figure 4.1(b), T_5 validates o_1 first and becomes the object owner of o_1. T_2, T_3, and T_4 abort when they validate. Figure 4.1(c) indicates that T_2, T_3, and T_4 request o_1 from n_5 again.



(a) Requesting o_1 (b) Validating o_1 (c) Re-requesting o_1

Figure 4.1: A Scenario consisting Four Transactions on TFA

Contention managers deal with only conflicts, determining which transaction wins or not. Past transactional schedulers (e.g., proactive and reactive schedulers) serialize aborted transactions but do not consider moving objects in data flow DTM. In DTM, the aborted transactions request an object again, increasing communication delays. Motivated by this observation, the transactions requesting o_1 are enqueued and the transactions immediately abort when one of these validate o_1.

As soon as o_1 is updated, o_1 is sent to the aborted transactions. The aborted transaction will receive the updated o_1 without any request, reducing communication delays. Meanwhile, we focus on which order of the aborted transactions lead to improved performance. Read transaction defined as read-dominated workloads will simultaneously receive o_1 to maximize the parallelism of read transactions. Write transactions including write operations will receive o_1 according to the shortest delay to minimize object moving time.

4.2 Scheduler Design

Bi-interval is similar to the BIMODAL scheduler [30] in that it categorizes requests into read and write intervals. If a transaction aborts due to a conflict, it is moved to a scheduling queue and assigned a backoff time. Bi-interval assigns two different backoff times defined as read and write intervals to read and write transactions, respectively. Unless the aborted transaction receives the requested object within an assigned backoff time, it will request the object again.

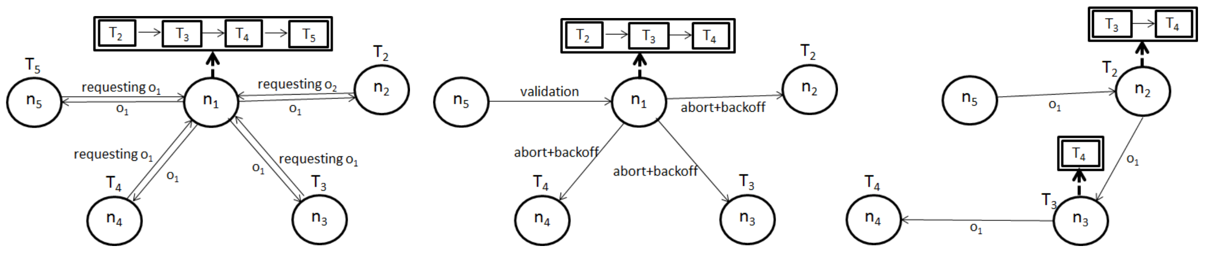
Bi-interval maintains a scheduling queue for read and write transactions for each object. If an enqueued transaction is a read transaction, it is moved to the head of the scheduling queue. If it is a write transaction, it is inserted into the scheduling queue according to the shortest path visiting each node invoking enqueued transactions. When a write transaction commits, the new version of an object is released. If read and write transactions have been aborted and enqueued for the version, the version will be simultaneously sent to all the read transactions and then visit the write transactions in the order of the scheduling queue. The basic idea of Bi-interval is to send a newly updated object to the enqueued-aborted transactions as soon as validating the object completes.

There are two purposes for enqueuing aborted transactions. First, in order to restart an aborted transaction, the CC protocol will be invoked to find the location of an object, incurring communication delays. An object owner holds a queue indicating the aborted transactions and sends the object to the node invoking the aborted transactions. The aborted transactions may receive the object without the help of the CC protocol, reducing communication delays. Second, Bi-interval schedules the enqueued aborted transactions to minimize execution times and communication delays. For reduced execution time, the object will be simultaneously sent to the enqueued read transactions. In order to minimize communication delays, the object will be sent to each node invoking the enqueued write transactions in order of the shortest path, so the total traveling time for the object in the network decreases.

Bi-interval determines read and write intervals indicating when aborted read and write transactions restart, respectively. This intends that an object will visit each node invoking aborted read and write transactions within read and write intervals, respectively. As a backoff time, a read interval is assigned to aborted read transactions and a write interval is assigned to aborted write transactions. A read interval is defined as the local execution time τ_i of transaction T_i. All enqueued-aborted read transactions will wait for τ_i and receive the object that T_i has updated. A write interval is defined as the sum of the local execution times of enqueued write transactions and a read interval.

The aborted write transaction may be serialized according to the order of the scheduling queue. If any of these transactions do not receive the object, they will restart after a write interval.

4.3 Illustrative Example



(a) Requesting o_1 (b) Validating o_1 (c) Scheduling Transactions

Figure 4.2: A Scenario consisting of Four Transaction on Bi-interval

Figure 4.2 shows a scenario consisting of four transactions based on Bi-interval. Node n_1 holds o_1 and write transactions T_2, T_3, T_4, and T_5 request object o_1 from n_1. n_1 has a scheduling queue holding requested transactions T_2, T_3, T_4, and T_5. If T_5 validates o_1 first as being illustrated by Figure 4.2(b), T_2, T_3, and T_4 abort. If n_2 is closest to n_5, o_1 updated by T_5 is sent to n_2, and two backoff times are sent to T_3 and T_4, respectively. Figure 4.2(c) shows one write interval.

While T_5 validates o_1, let us assume that other read transactions request o_1. The read transactions will be enqueued and simultaneously receive o_1 after T_5 completes its validation. Thus, once the scheduling queue holds read and write transactions, a read interval will start first. The write transactions will be serialized according to the shortest object traveling time.

4.4 Algorithms

The data structures depicted in Algorithm 1 are used in Algorithms 2. The data structure of $Requester$ consists of the address and the transaction identifier of a requester. $Requester\_List$ maintains a linked list for $Requester$ and $BackoffTime$. $removeDuplicate()$ checks and removes a duplicated transaction in $Requester\_List$. $scheduling\_List$ is a hash table that holds a $Requester\_List$ including requesters for an object with $Object\_ID$.

Algorithm 2 describes $Retrieve\_Request$, which is invoked when an object owner receives a request. If the corresponding object is being used, $Retrieve\_Request$ has to decide whether the requester is aborted or enqueued on $elapsed\_time$. Unless $BackoffTime$ corresponding to the object exceeds $elapsed\_time$, the requester is added to $scheduling\_List$. $local\_exectuion\_time$ of the the requester is an expected total running time. Thus, $local\_exectuion\ \_time$ - $elapsed\_time$

Algorithm 1: Structure of Scheduling Queue

```
Class Requester {
    Address $address$;
    Transaction_ID $txid$;
}
Class Requester_List {
    List<Requester> Requesters = new LinkedList<Requester>();
    Integer BackoffTime;
    void addRequester(backoff_time, Requester);
    void removeDuplicate(Address);
}
Map<Object_ID, Requester_List> scheduling_List = new ConcurrentHashMap<Object_ID, Requester_List>();
```

is the remained time that the requesting transaction will run in advance. As soon as validating the object completes, it is sent to the first element of $scheduling\_List$.

Algorithm 2: Algorithm of Retrieve_Request

```
Procedure Retrieve_Request
Input: $oid.\ txid$, $local\_exectuion\_time$, $elapsed\_time$
$object$ = get_Object($oid$);
$address$ = get_Requester_Address();
Integer $backoff$ = 0;
if $object$ is not null and in use then
    Requester_List $reqlist$ = scheduling_List.get($oid$);
    if $reqlist$ is null then
        $reqlist$ = new Requester_List();
    else
        $reqlist$.removeDuplicate($address$);
    if $reqlist.BackoffTime < elapsed\_time$ then
        $backoff = reqlist.BackoffTime$;
        $reqlist.addRequest$($local\_exectuion\_time$-$elapsed\_time$, new Requester($address$, $txid$));
        scheduling_List.put($oid$, $reqlist$);
Send $object$ and $backoff$ to $address$;
```

Algorithm 3 shows the $bi\_interval$ scheduler that is invoked after the object indicated by oid was validated. The owner of oid is transferred to the node that has validated the object last. The node has a responsibility to send the object to the first element of $scheduling\_List$ after invoking $bi\_interval$. The $Distance$ function returns a communication delay between the object owner and the requesting node indicated by $Requester.Address$. $readRequesters$ as an instance of $Requester\_List$ holds all requesters for read transactions. After execution Algorithm 3, the object is simultaneously sent to all addresses of $readRequesters$ if $readRequesters$ is not empty. $NextNode$ indicates the nearest nodes's address.

Algorithm 3: Algorithm of the $bi\_interval$ function

```
Procedure $bi\_interval$
Input: $scheduling\_List$, $oid$
Output: $Address$
$reqlist$ = $scheduling\_List$(oid);
NextNode = null;dist=$\infty$
if $reqlist$ is not null then
    foreach $Requester \in reqlist$ do
        if $Requester$ is for write transaction then
            if $dist >$ Distance(Requester.Address) then
                dist = Distance(Requester.Address);
                NextNode = Requester.Address;
        else
            readRequesters.addRequester(Requester);
return NextNode;
```

4.5 Analysis

The object moving cost is defined as $\eta_A(u, V_{T_{N-1}})$, which is the total communication delay for visiting each node from node u holding an object to $N-1$ nodes in $V_{T_{N-1}}$, under scheduler A. $V_{T_{N-1}}$ represents a set of $N-1$ nodes invoking transactions.

Theorem 4.5.1. *Bi-interval's execution makespan competitive ratio is 1+$\frac{I_r}{N-k+1}$ for N transactions including k read transactions, where I_r is the number of read intervals.*

Proof. The optimal off-line algorithm concurrently executes all read transactions. So, Bi-interval's optimal execution for N transactions including k read transactions is $\sum_{m=1}^{N-k+1} \gamma_m$.

$$CR_{Biinterval} \leq \frac{\gamma_\omega \cdot I_r + \sum_{m=1}^{N-k+1} \gamma_m}{\sum_{m=1}^{N-k+1} \gamma_m} \approx \frac{I_r + N - k + 1}{N - k + 1}$$

, where γ_ω is γ of a read transaction. The theorem follows. □

Theorem 4.5.2. *Bi-interval's traveling makespan competitive ratio for k reads of N transactions is* $\log(N + I_r - k - 1)$.

Proof. Bi-interval follows the nearest neighbor path to visit each node in the scheduling list. We define the *stretch* of a transactional scheduler as the maximum ratio of the moving time to the communication delay—i.e., $Stretch_\eta(u, V_{T_{N-1}}) = \max \frac{\eta_{Biinterval}(u, V_{T_{N-1}})}{d(u, V_{T_{N-1}})} \leq \frac{1}{2}\log(N-1) + \frac{1}{2}$ from [80]. Hence, $CR_{Biinterval} \leq \log(N + I_r - k - 1)$. The theorem follows. □

Theorem 4.5.3. *The total worst-case competitive ratio $CR^{Worst}_{Biinterval}$ of Bi-interval for N transactions is $O(\log(N))$.*

Proof. In the worst-case, $I_r = k$. This means that there are no consecutive read intervals. Thus, $\mathrm{makespan}^{\mathrm{N}}_{\mathrm{OPT}}$ and $\mathrm{makespan}^{\mathrm{N}}_{\mathrm{Biinterval}}$ satisfy the following, respectively:

$$makespan^N_{OPT} = \Gamma_{N-k+1} + \min d(u, V_{T_{N-k+1}}) \tag{4.1}$$

$$makespan^N_{Biinterval} = \Gamma_{N-1} + \log(N-1) \max d(u, V_{T_{N-1}}) \tag{4.2}$$

Hence, $CR^{Worst}_{Biinterval} \leq \log(N-1)$. The theorem follows. $\square$

We now focus on the case $I_r < k$.

Theorem 4.5.4. *When $I_r < k$, Bi-interval improves the traveling makespan $(i.e., makesp\,an^N_{Biinterval}(NR))$ as much as $O(|\log(1-(\frac{k-I_r}{N-1})|)$ for k reads of N transactions.*

Proof.

$$\begin{aligned}
\max\ & \frac{\eta(u, V_{T_{N+I_r-k-1}})}{d(u, V_{T_{N-1}})} && (4.3)\\
=\ & \max\Big(\frac{\eta(u, V_{T_{N-1}})}{d(u, V_{T_{N-1}})} + \frac{\varepsilon}{d(u, V_{T_{N-1}})}\Big)\\
\leq\ & \frac{1}{2}\log(N-k+I_r-1) + \frac{1}{2}
\end{aligned}$$

When $I_r < k$, a read interval has at least two read transactions. We are interested in the difference between $\eta(u, V_{T_{N-1}})$ and $\eta(u, V_{T_{N+I_r-k-1}})$. Thus, we define ε as the difference between two η values.

$$\max \frac{\varepsilon}{d(u, V_{T_{N-1}})} \leq \frac{1}{2}\log(\frac{N-k+I_r-1}{N-1}) \tag{4.4}$$

In (4.4), due to $I_r < k$, $\frac{N-k+I_r-1}{N-1} < 1$. Bi-interval is invoked after conflicts occur, so $N \neq k$. Hence, ε is a negative value, improving the traveling makespan. The theorem follows. $\square$

The average-case analysis (or, probabilistic analysis) is largely a way to avoid some of the pessimistic predictions of complexity theory. Bi-interval improves the competitive ratio when $I_r < k$. This improvement depends on the size of I_r—i.e., how many read transactions are consecutively arranged. We are interested in the size of I_r when there are k read transactions. We analyze the expected size of I_r using probabilistic analysis. We assume that k read transactions are not consecutively arranged (i.e., $k \geq 2$) when N requests are arranged according to the nearest neighbor algorithm. We define a probability of actions taken for a given distance and execution time.

Theorem 4.5.5. *The expected number of read intervals $E(I_r)$ of Bi-interval is* $\log(k)$.

Proof. The distribution used in the proof of Theorem 4.5.5 is an independent uniform distribution. p denotes the probability for k read transactions to be consecutively arranged.

$$\begin{aligned}
E(I_r) &= \int_{p=0}^{1} \sum_{I_r=1}^{k} \binom{k}{I_r} \cdot p^k (1-p)^{k-I_r} dp \\
&= \sum_{I_r=1}^{k} \Big(\frac{k!}{I_r! \cdot (k-I_r)!} \int_{p=0}^{1} p^k (1-p)^{k-I_r} dp \Big) \\
&\approx \sum_{I_r=1}^{k} \frac{k!}{I_r!} \cdot \frac{k!}{(2k - I_r + 1)!} \approx \log(k) \qquad (4.5)
\end{aligned}$$

We derive Equation 4.5 using the beta integral. The theorem follows. □

Theorem 4.5.6. *Bi-interval's total average-case competitive ratio ($CR^{Average}_{Biinterval}$) is $\Theta(\log(N-k))$ for k reads of N transactions.*

Proof. We define $CR^{m}_{Biinterval}$ as the competitive ratio of node m. $CR^{Average}_{Biinterval}$ is defined as the sum of $CR^{m}_{Biinterval}$ of $N + E(I_r) - k + 1$ nodes.

$$\begin{aligned}
CR^{Average}_{Biinterval} &\leq \sum_{m=1}^{N+E(I_r)-k+1} CR^{m}_{Biinterval} \\
&\leq \log(N + E(I_r) - k + 1) \approx \log(N-k)
\end{aligned}$$

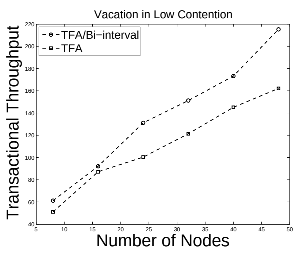
Since $E(I_r)$ is smaller than k, $CR^{Average}_{Biinterval} = \Theta(\log(N-k))$. The theorem follows. □

4.6 Evaluation

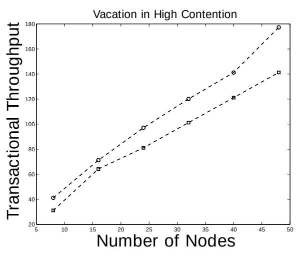
We compared *TFA* with Bi-interval (referred to as TFA/Bi-interval) against competitors only TFA. We measured the transactional throughput—i.e., the number of committed transactions per second under increasing number of requesting nodes, for the different schemes.

We developed a set of four distributed applications as benchmarks. These include distributed versions of the Vacation benchmark of the Stanford STAMP (multiprocessor STM) benchmark suite [69], two monetary applications (Bank and Loan). and Red/Black Tree (RB-Tree) [38] as microbenchmarks. We created 10 objects, distributed them equally over the 48-nodes, and executed hundred transactions at each node. We used low and high contention levels, which are defined as 90% read transactions and 10 objects, and 10% read transactions and 5 objects, respectively.

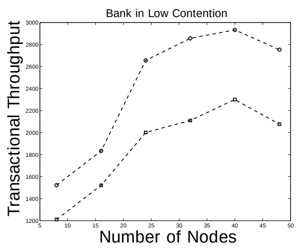
A transaction's execution time consists of inter-node communication delay, serialization time, and execution time. Communication delay between nodes is limited to a number between $1ms$ and $10ms$ to create a static network. Serialization delay is the elapsed time to ensure correctness



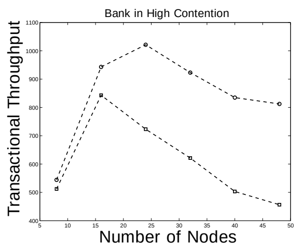
(a) Vacation (Low)



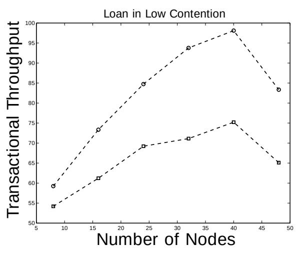
(b) Vacation (High)



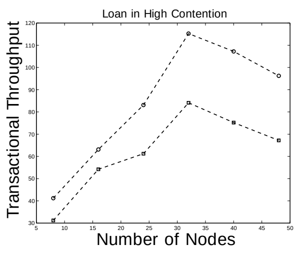
(c) Bank (Low)



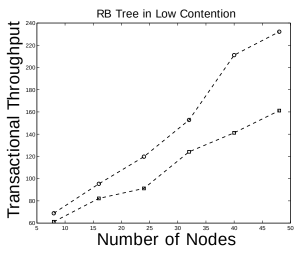
(d) Bank (High)



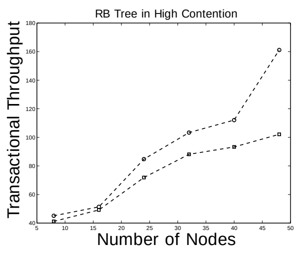
(e) Loan (Low)



(f) Loan (High)



(g) RB Tree (Low)



(h) RB Tree (High)

Figure 4.3: Throughput Under Four Benchmarks in Low and High Contention

of concurrent transactions. This delay also includes waiting time in a scheduling queue and Bi-interval's computational time.

In low contention, Bi-interval produces high concurrency due to the large number of read-only transactions. In high contention, Bi-interval reduces object moving time. In both cases, Bi-interval improve throughput, but concurrency of read-only transactions improves more throughput than reduced object moving time. Our experimental evaluation shows that Bi-interval enhances throughput over TFA as much as 1.77$\sim$ 1.65$\times$ speedup under low and high contention, respectively.

Chapter 5

The Commutative Requests First

5.1 Motivation

With a single copy for each object, i.e., *single-version* STM (SV-STM), when a read/write conflict occurs between two transactions, the contention manager resolves the conflict by aborting one and allowing the other to commit, thereby maintaining the consistency of the (single) object version. SV-STM is simple, but suffers from large number of aborts [16]. In contrast, with multiple versions for each object, i.e., *multi-versioning* STM (MV-STM), unnecessary aborts of transactions, that could have been committed without violating consistency, are avoided [17]. Unless a conflict between operations to access a shared object occurs, MV-STM allows the corresponding transactions to read the object's old versions, enhancing concurrency. MV-STM has been extensively studied for multiprocessors [16, 19] and also for distributed systems [39]. MV-STM uses *snapshot isolation* (SI), which is weaker than serializability [40]. A transaction executing under SI operates on a snapshot taken at the start of the transaction. The transaction successfully commits if the objects updated by the transaction have not been changed externally since the snapshot was taken, guaranteeing that all read transactions will see a consistent snapshot. Many works [40, 41, 42] used SI for improving performance in centralized and distributed TM environments. Even though SI allows more concurrency among transactions respect to with serializability, a write-write transaction's conflict under SI causes the transaction to abort. In write-intensive workloads, this conflict cannot be avoided because the concurrency of write transactions may violate SI.

In this dissertation, we address the problem of permitting multiple conflicting transactions to commit concurrently, in order to enhance concurrency of write transactions without violating SI in multi-version cc DTM for high performance. We propose a transactional scheduler that enables concurrency of write transactions, called Commutative Requests First (CRF). In order to do that, CRF exploits the notion of commutative operations. Two operations are named *commutative* if applying them sequentially in either order, they leave the objects accessed in the same state and both return the same values.

A very intuitive example of commutativity is when two operations, $call1(X)$ and $call2(X)$, accessing both to the same object X but different fields of X. Thus, CRF checks whether write operations are commutative and lets them to validate and commit simultaneously. Unlike past STM works, that exploit high concurrency based on the commutativity property [43], CRF maintains a scheduling queue to identify commutative and non-commutative transactions, and could decide to allow all commutative transactions to commit first than the others, maximizing their concurrency. However, despite the significant performance obtained by adopting the idea of commutativity transactions of CRF, there could be applications that do not admit such kind of commutativity. CRF addresses this issue by permitting the developer to explicitly specify non-commutative operations.

5.2 Commutative Requests First in MV-TFA

5.2.1 Multi-Version TFA

In this section we present multi-version MV-TFA, our extension of TFA supporting SI. The basic idea is to record an event whenever requesting and acquiring an object. Let n_i denote a node invoking a transaction T_i. We define two types of events: (1) ***Request***$(\mathrm{Req}(\mathrm{n_i}, \mathrm{o_j}))$ representing the request of object o_j from node n_i; (2) *Acquisition*$(\mathrm{Acq}(\mathrm{n_i}, \mathrm{o_j}))$ indicating when node n_i acquires object o_j. Figure 5.1 shows an example execution scenario of MV-TFA. We use the same style in the figure as that of [81]. The solid circles indicate write operations and the empty circles represent read operations. Transactions' evolution is represented on horizontal lines with the circles. The horizontal line corresponding to the status of each object describes the time domain. The dotted line indicates which node requests an object from where.

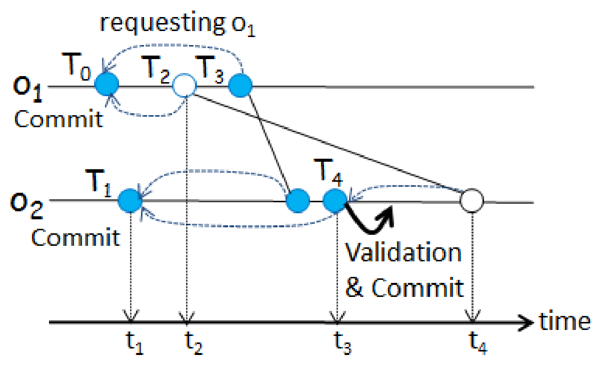
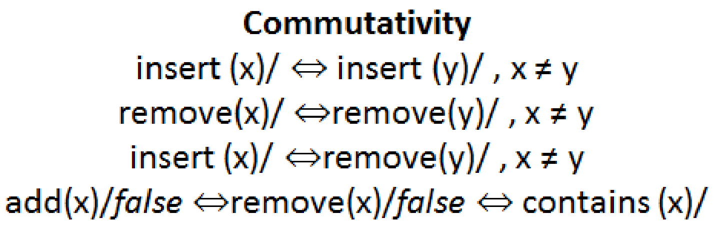


Figure 5.1: Example of MV-TFA

Assume that transactions T_0 and T_1 invoked on nodes n_0 and n_1 commit after writing o_1^0 and o_2^0, respectively. Let transactions T_2, on node n_2, and T_3, on node n_3, request objects o_1 and o_2 from nodes n_0 and n_1, respectively. Node n_1 holds the list of versions of o_2. After that, T_3 requests o_1 from n_0 and subsequently T_4 requests o_2 from n_1. Thus, n_1 records the events $Acq(n_3, o_2^0)$ and $Acq(n_4, o_2^0)$. Then T_4 updates o_2 creating a new version o_2^1. When T_4 validates o_2^1 to commit, $Acq(n_4, o_2^0)$ is removed from the events log of n_3, and T_3 has forced to abort because in the n_3's log there is another request ($Acq(n_3, o_2^0)$) on the same object o_2. The presence of this entry in the log means that T_3 has not yet completed, so T_4 definitively commits before that T_3 validates o_2, invalidating the object o_2^0 accessed by T_3. As a consequence of T_4 commitment, node n_4, which invokes T_4, receives the versions o_2^0 and o_2^1 of object o_2. Now, after the commit of T_4, T_2 requests o_2 with the value $\mid t_4$ - $t_2 \mid$ from n_4. It replies with the version o_2^0 instead of the newly o_2^1 because o_2^0 has been updated at time t_1 to T_2, because $\mid t_4 - t_3 \mid < \mid t_4$ - $t_2 \mid < \mid t_4 - t_1 \mid$. Using this mechanism, T_2 can access to a consistent snapshot that is not affected by a write operation by T_4, instead of being aborted due to T_4 's write. This is how MV-TFA ensures SI.

5.2.2 CRF Scheduler Design

MV-TFA shows how to enhance performance in case of workload characterized by mostly read transactions, exploiting multi-versions. In this subsection we focus on how to schedule write transactions concurrently minimizing the abort rate and increasing the parallelism.


Commutativity

insert (x)/ ⇔ insert (y)/ , x ≠ y

remove(x)/ ⇔remove(y)/ , x ≠ y

insert (x)/ ⇔remove(y)/ , x ≠ y

add(x)/*false* ⇔remove(x)/*false* ⇔ contains (x)/

Figure 5.2: Specification of a Set

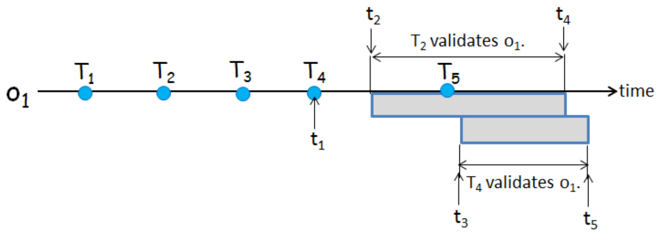
When a transaction T_1 at node n_1 needs object o_1 for an operation, it sends a request to the o_1's object owner. If the operation is read, a version of o_1 is sent to n_1. If the operation is write, we consider two possible cases in terms of o_1. *(A)* The first case happens when other transactions may have requested o_1 but no transaction has validated o_1. In this case, a version of o_1 is sent to n_1 and T_1's request moves into the scheduling queue of the o_1's owner. *(B)* The second case is when another transaction T_2 is validating o_1. In this case, unless T_2 and T_1 commute, T_1 will abort and T_1's request also moves to the scheduling queue. If T_2 and T_1 commutes, o_1 is sent to n_1 and T_1's request moves to the scheduling queue. The o_1's owner maintains the scheduling queue to execute commutative transactions concurrently. Accordingly, the non-commutative transactions will be executed serially. To better assess CRF, we use it to implement the specification of a Set provided by [43]. We recall that a Set is a collection of items without duplications in which the following operations are provided: $add(x)$, $remove(x)$ and $contains(x)$ where x is the item of the

Set accessed. Figure 5.2 summarizes Set operations' commutativity according to [43]'s definition.

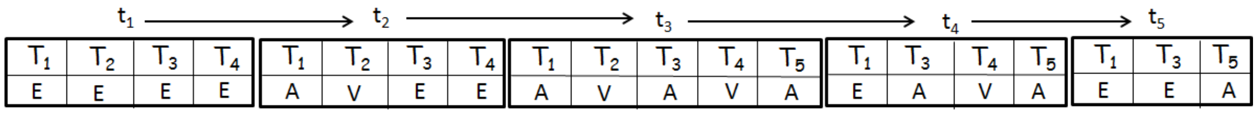
In the specification illustrated in Figure 5.2, operations $insert(x)$, $insert(y)$, and $insert(z)$ commutes if $x \neq y \neq z$. Multiple write transactions may be invoked concurrently on the Set. CRF identifies commutative and non-commutative transactions and gives to the commutative transactions a chance to validate concurrently an object first. However, if we consider the specification of the Set, in which the are no commutative operations declared, and we encapsulate the Set into an object (o_1) and we consider the above operations as transactions, then typical concurrency control does not permit to validate and commit concurrently more than one transaction performing an update on the object (namely updating the Set). Conversely, Figure 5.2 shows that multiple update transactions can be validated concurrently whether they access to different items in the set. The scheduling queue holds requests for those operations. If multiple transactions have requested the same version of o_1, CRF allows the commutative transactions to concurrently validate o_1. Meanwhile, many commutative transactions may validate o_1. This could bring non-commutative transactions to "starve" on o_1. Thus, CRF alternates between periods (called *epochs*), in which it privileges the validation of a group of commutative transactions, with others in which it prefer to validate the non-commutative ones. In this way, CRF handles conflicts between commutative and non-commutative transactions. Although epochs contain commutative transactions, these transactions do not commute with the transactions of the next epoch in the chronological sequence. The terminology "commutative" and "non-commutative" epoch distinguishes between these two epochs. Thus, in commutative epoch, commutative transactions validate o_1 and then in the next (i.e., non-commutative) epoch, non-commutative transactions, excluded in the previous commutative epoch, can validate o_1. If a transaction starts validating o_1, its commutative transactions are also allowed to validate o_1 but its non-commutative transactions abort. The non-commutative transactions will resume after the commutative transactions commit.
CRF checks for whether different operations commute at the level of semantics. Even when commutative operations concurrently update the object, the object preserves a consistent state, ensuring SI. There are two purposes for processing commutative requests first. First, MV-TFA ensures concurrency of read transactions, and CRF is responsible to detect conflicts among commutative and non-commutative write transactions, reducing the number of conflicts. This leads to higher concurrency. Second, CRF alleviates contention when many write transactions are invoked. Even though a conflict between two write transactions occurs, all subsequent commutative transactions are scheduled first. Non-commutative transactions restart simultaneously after the commutative transactions complete, so CRF avoids further conflicts, decreasing contention.

5.3 Illustrative Example

Figure 5.3 shows a scenario of CRF. The write transactions T_1=$insert(x)$, T_2=$remove(x)$, T_3=$insert(y)$ and T_4=$remove(y)$ request concurrently o_1 from its owner. The transactions obtain the version of o_1. The state of the scheduling queue at t_1, illustrated in Figure 5.3(b), shows that the transactions are all executing. At t_2, T_2 starts validating o_1. Consequently, T_1 aborts because T_1 and T_2



(a) Requests of Five Transactions and Validation of Two Transactions for Object o_1.



t1

T1	T2	T3	T4
E	E	E	E

t2

T1	T2	T3	T4
A	V	E	E

t3

T1	T2	T3	T4	T5
A	V	A	V	A

t4

T1	T3	T4	T5
E	A	V	A

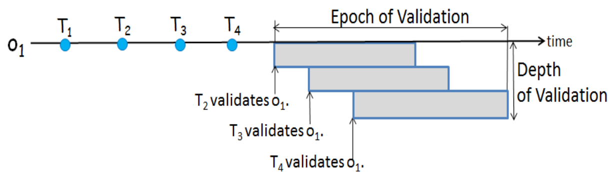
t5

T1	T3	T5
E	E	A

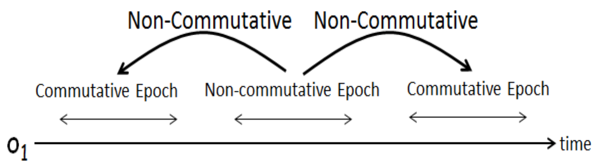
(b) Scheduling Queue Located in o_1 Object Owner. The scheduling queue consists of two rows: Enqueued Transactions and State of the Transactions. V (Validation), A (Abort), and E (Execution)

Figure 5.3: A Scenario of CRF

do not commute. Conversely, T_3 and T_4 can still execute because they are commutative with T_2. Then T_5=$remove(x)$ requests o_1 during the validation of T_2 and immediately aborts because T_5 and T_2 do not commute. At t_3, T_4 starts validating o_1 and T_3 aborts because T_3 and T_4 do not commute. Thus, T_2 and T_4 concurrently validate o_1. When T_2 ends validation (i.e., commits) at t_4, the version updated by T_2 is sent to the non-commutative transaction T_1, and T_1 starts executing. Even though T_5 is a non-commutative transaction of T_2, only T_1 starts to avoid a conflict between non-commutative transactions. Finally, the version updated by T_4 at t_5 is sent to T_3. T_1 and T_3 may validate o_1 concurrently because they commute.



(a) Epoch and Depth of Validation.



(b) Epochs of Validation

Figure 5.4: Epoch-based CRF

Figure 5.4(a) shows that the validation of commutative transactions may not be completely overlapping, so the period of validation may be stretched. This may lead to the deferred execution of non-commutative transactions. To prevent this, we define a new parameter, called *depth of validation*, namely the number of transactions involved in the validation. Figure 5.4(a) indicates 3 for that depth, meaning that the commits of three transactions mark the end of the epoch. Non-commutative transactions will start after the epoch. Figure 5.4(b) illustrates the relationship of epochs. In each epoch, commutative transactions concurrently participate in validation. At the end of the epoch, their non-commutative transactions held in a scheduling queue, restart. Non-commutative transactions will validate in the next epoch.

5.4 Implementation and Experimental Evaluation

5.4.1 Implementation

We implemented CRF on MV-TFA using Scala's actor model for Java Virtual Machine. The actor model prohibits sharing memory by encapsulating mutable state inside light-weight sequential constructs called actors and it become popular with the advent of the Erlang programming language. Since then, many languages (e.g., Google Go) have embraced this model.

Actors communicate through message passing and their operations always execute sequentially, avoiding concurrency problems (e.g., data contention). The actor model is based on communicating sequential processes introduced by Hoare in [82], and become popular with the advent of the Erlang programming language. Since then, many languages (e.g., Google Go) have embraced this model. The Scala API is based on the excellent ScalaSTM API, which is due to be included in Scala's standard library.

5.4.2 Commutativity of Benchmarks

We assess the performance of CRF using LinkedList and SkipList as micro-benchmarks and a TPC-C [83] as real-application benchmark. Regarding the commutativity in micro-benchmarks, the Set (as introduced in Section 5.2) can be implemented with LinkedList and SkipList [43], so we rely on the definition of commutativity in Figure 5.2. Regarding TPC-C, the write transactions consist of update, insert, and/or delete operations accessing a database of nine tables maintained in memory. Each row in the tables has a unique key. It is composed by five different transactions: `New-order`, `Payment` and `Delivery` for write transactions and `Order-status` and `Stock-level` for read transactions.

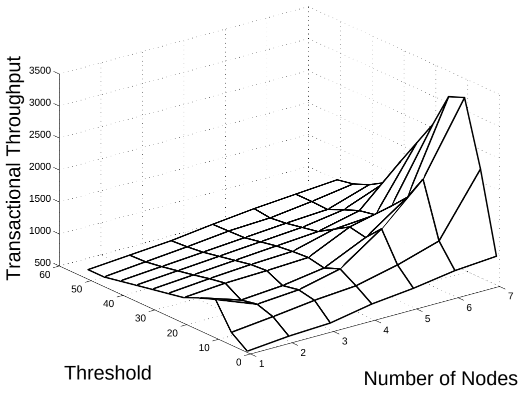
Multiple operations commute if they access to a row (or object) with the same key and modify different columns. We rely on explicit annotations provided by the programmer, indicating the fields accessed by each transaction profile. We configured the benchmark with a limited number of warehouses (#4) in order to generate high conflicts. We recall that, in data flow model, objects are not bound on fixed nodes but move, increasing likelihood of conflicts.

5.4.3 Experimental Setup

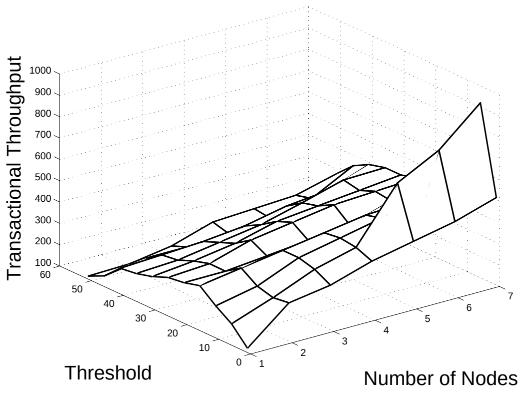
Our test-bed consists of 10 nodes connected via a switched 1 Gigabit network connection. Each node is comprised of 12 Intel Xeon 1.9GHz processor cores. We use the Ubuntu Linux 10.04 server OS. We measured the *transactional throughput* (number of committed transactions per second). To manage garbage collection, versions that are no longer accessible, need to be marked. Unlike multiprocessors, determining old versions for live transactions in distributed systems incurs communication overheads. Thus, we consider a threshold-based garbage collector [84], which checks the number of versions and disposes the oldest if the number of versions exceeds a pre-defined threshold. We consider threshold 4 for measuring the basic event model's throughput, because the observed that the speed-up is relatively less increased after threshold.

5.4.4 Finding a Depth

The large number of concurrent validations may lead to a significant scheduling overhead due to delayed non-commutative transactions. For the balance of commutative and non- requesting transactions, we consider a threshold-based control, switching the next epoch when either a depth or a number of non-commutative transactions enqueued meets a predefined threshold, called $MaxD$.



(a) LinkedList



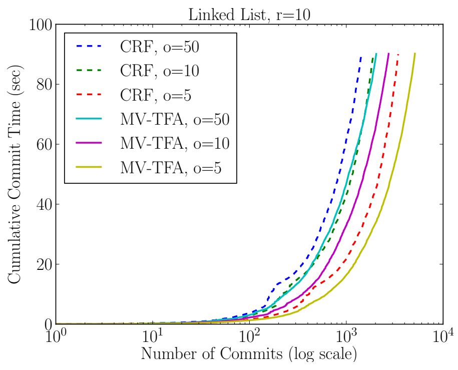
(b) TPC-C

Figure 5.5: Throughput Varying Thresholds

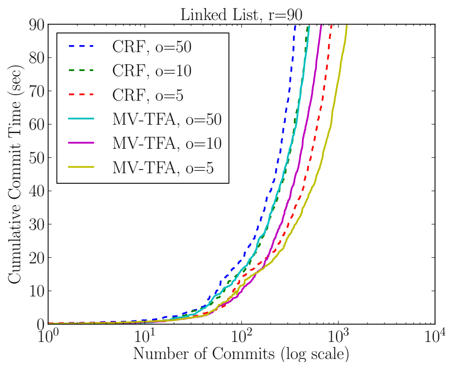
Figure 5.5 shows throughput moving the $MaxD$ from 1 to 50. By the plot is clear that CRF's throughput is not improved after $MaxD$=10 for LinkedList and $MaxD$=5 for TPC-C due to the increasing number of non-commutative transactions aborted. With the previous values of $MaxD$, CRF reaches its maximum throughput, so we used those for the experiments.

5.4.5 Evaluation

We are interested in the execution time for each transaction. The concurrency of transactions leads to reduced execution times, so we measure commit times for each transaction, and compute the cumulative commit time indicating how fast transactions commit. Figure 5.6 shows cumulative commit times of Linkedlist over CRF and MV-TFA. This indicates how many transactions have committed during 90 seconds with a log scale. In Figure 5.6, we focus on the effectiveness of CRF. As long as read-only ratio (r) increases, the probability to schedule commutative and non-commutative transactions increases. Also, lower o values lead to the chance of enhancing concurrency with CRF. As illustrated in Figure 5.6, the gap between the curves of CRF and MV-TFA with r=10 and o=5 is maximized, implying the effectiveness of the CRF scheduler. Meanwhile, commit



(a) Linked List, r=10

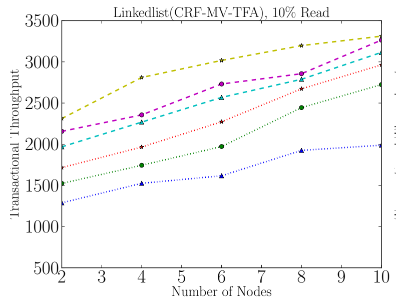


(b) Linked List, r=90

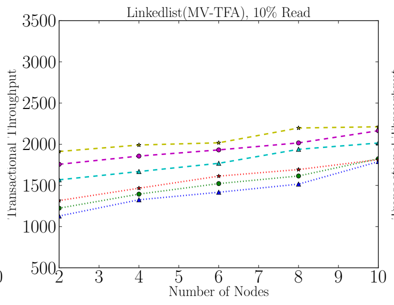
Figure 5.6: Cumulative Commit Times with $r \in \{10, 90\}$, $o \in \{5, 10, 50\}$, and 8 nodes

times on CRF with r=90 and o=50 are not reduced much due to the small number of commutative write transactions.

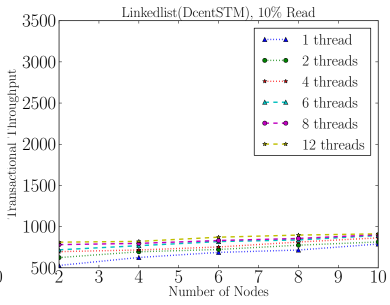
Figures 5.7,5.9 show the throughput of CRF, MV-TFA and DecentSTM using LinkedList(Figure 5.7) and SkipList(Figure 5.9) benchmarks. The legend has to be considered for all the plots and shows the colors differentiate for number of running threads. Each micro-benchmark has been evaluated using two workloads representative of read intensive (10% writes and 90% reads) and write intensive (90% writes and 10% reads) scenarios. The tests have been performed varying the number of nodes and the number of threads per node. Each thread submits requests to the distributed system. Summarizing, we span scenarios from 2 up to 120 concurrent threads in the system. This allows us to exhaustively assess the behavior of CRF. The comparison between CRF and MV-TFA shows how much CRF enhances the concurrency of write transactions. For the LinkedList and SkipList, the new value to add or delete is randomly selected using a uniform distribution. According to the increasing number of threads and nodes, CRF performs better due to the detection of a large number of commutative operations. Even though the throughput of CRF is slightly better than MV-TFA in the scenario characterized by mostly read-only transactions (due to the limited number of commutative write operations), the maximum gain of CRF against competitors is reached in write-intensive workload where CRF exploits the ability to validate and commit concurrently conflicting transactions. In addition, the plot reveals that, in write dominated workload, CRF scales better than MV-TFA and DecentSTM. In fact, in contrast with CRF, their performance stall when increasing the number of concurrent threads in the system. This is also confirmed by the plots in Figure 5.9(a) and 5.9(b) where CRF outperforms MV-TFA by as much as 2×. As a competitor, DecentSTM [52] is based on a snapshot isolation algorithm, which requires searching the history of objects to find a valid snapshot. This algorithm incurs a significant overhead. Thus, we observe that the transactional throughput of DecentSTM is not improved as long as requesting nodes increase. Our evaluations reveal that CRF improves throughput over MV-TFA and DecentSTM by as much as (average) 2× and 3× under 10% read transactions, respectively. Further, our evaluations



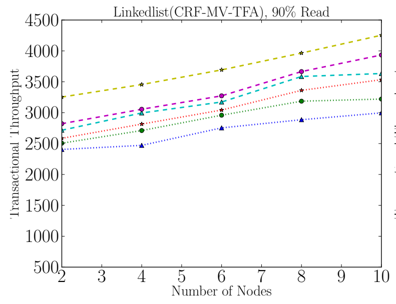
(a) CRF-MV-TFA, 10% Read



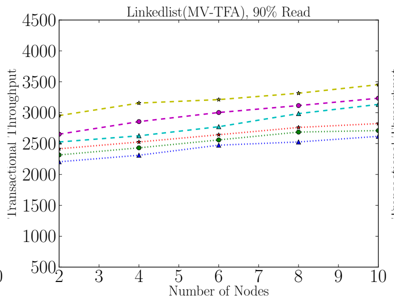
(b) MV-TFA, 10% Read



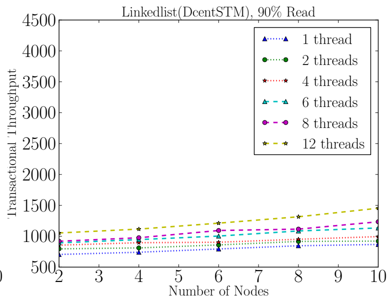
(c) DecentSTM, 10% Read



(d) CRF-MV-TFA, 90% Read



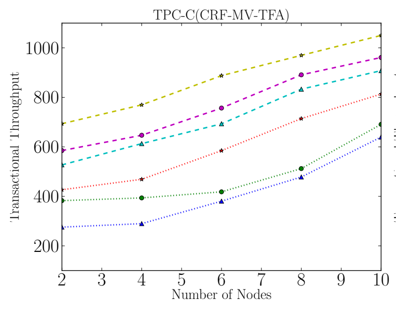
(e) MV-TFA, 90% Read



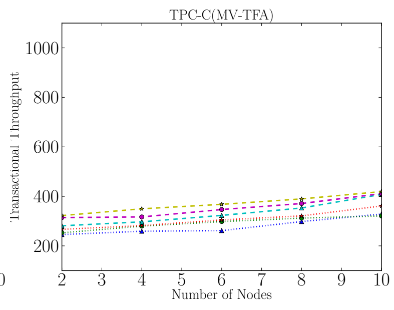
(f) DecentSTM, 90% Read

Figure 5.7: Throughput of CRF, MV-TFA, and DecentSTM Using LinkedList.

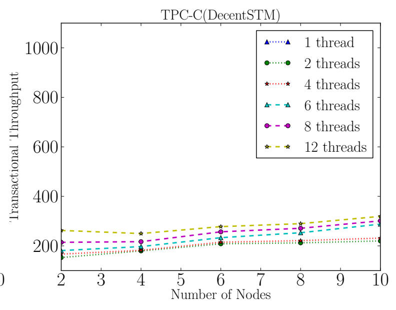
show that MV-TFA outperforms DecentSTM in throughput as much as $2\times$. Figure 5.8 shows the throughput of CRF, MV-TFA, and DecentSTM using TPC-C benchmark. We used the amount of read and write transactions that the specification of TPC-C recommends. TPC-C benchmark accesses large tables to read and write values. Due to the non-negligible transaction execution time, scheduling commutative operations highly impacts the overall performance. In fact, the conflicting transactions generated by the benchmark are well managed by CRF and this results observing that CRF performs better than DecentSTM as much as $5\times$ over 10 nodes.



(a) CRF-MV-TFA

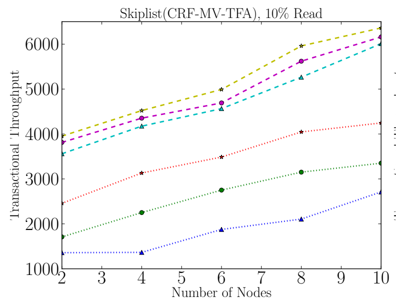


(b) MV-TFA

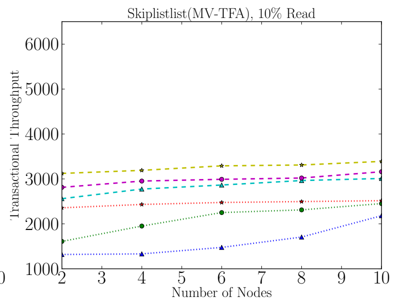


(c) DecentSTM

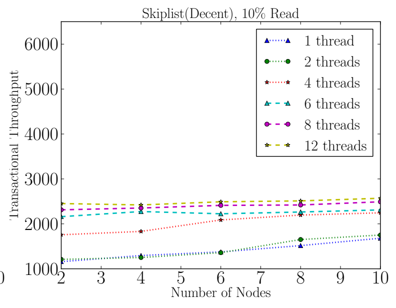
Figure 5.8: Throughput of CRF, MV-TFA, and DecentSTM Using TPC-C



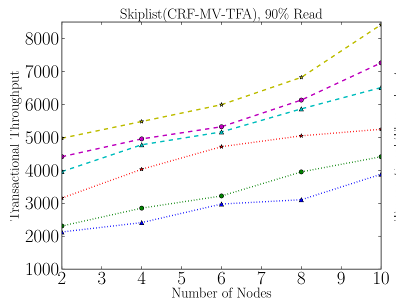
(a) CRF-MV-TFA, 10% Read



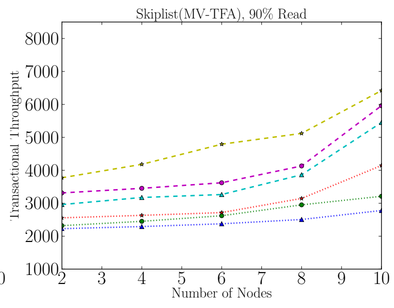
(b) MV-TFA, 10% Read



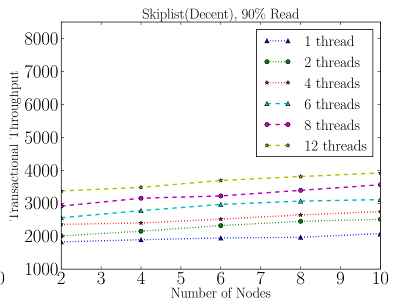
(c) DecentSTM, 10% Read



(d) CRF-MV-TFA, 90% Read



(e) MV-TFA, 90% Read



(f) DecentSTM, 90% Read

Figure 5.9: Throughput of CRF, MV-TFA, and DecentSTM Using SkipList.

Chapter 6

The Reactive Transactional Scheduler

6.1 Motivation

Past transactional scheduler often causes only small number of aborts and reduces the total communication delay in DTM [28]. However, aborts may increase when scheduling nested transactions. In the flat and closed nesting models, if an outer transaction, which has multiple nested transactions, aborts due to a conflict, the outer and inner transactions will restart and request all objects regardless of which object caused the conflict. Even though the aborted transactions are enqueued to avoid conflicts, the scheduler serializes the aborted transactions to reduce the contention on only the object that caused the conflict. With nested transactions, this may lead to heavy contention because all objects have to be retrieved again.

Proactive schedulers abort the losing transaction with a backoff time, which determines how long the transaction is stalled before it is re-started [31, 32]. Determining backoff times for aborted transactions is generally difficult in DTM. For example, the winning transaction may commit before the aborted transaction is restarted due to communication delays. This can cause the aborted transaction to conflict with another transaction. If the aborted transaction is a nested transaction, this will increase the total execution time of its parent transaction. Thus, the backoff strategy may not avoid or reduce aborts in DTM.

Motivated by this, we propose the RTS scheduler for closed-nested DTM. RTS reduces the number of parent transactions' aborts to prevent their committed nested transactions from the aborts. RTS checks the length of the parent transaction's execution time and determines whether losing transaction is aborted or enqueued. If the parent transaction has a short execution time, it aborts. Otherwise, it is enqueued to preserve its nested transactions. A backoff time used for the enqueued parent transaction indicate when the transaction is likely to receive an object.

6.2 Scheduler Design

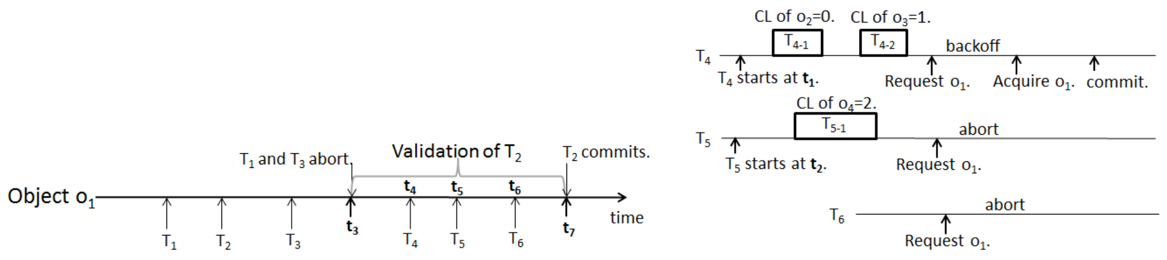
We consider two kinds of aborts that can occur in closed-nested transactions when a conflict occurs: aborts of nested transactions and aborts of parent transactions. Closed nesting allows a nested transaction to abort without aborting its parent transaction. If a parent transaction aborts however, all of its closed-nested transactions are aborted. Thus, RTS performs two actions for a losing parent transaction. First, determining whether losing transaction is aborted or enqueued by the length of its execution time. Second, the losing transaction is aborted if it is a parent transaction with a "high" contention level. A parent transaction with a "low" contention level is enqueued with a backoff time.

The contention level (CL) of an object o_j can be determined in either a local or distributed manner. A simple local detection scheme determines the local CL of o_j by how many transactions have requested o_j during a given time period. A distributed detection scheme determines the remote CL of o_j by how many transactions have requested other objects before o_j is requested. For example, assume that a transaction T_i is validating o_j, and T_k requests o_j from the object owner of o_j. The local CL of o_j is 1 because only T_k has requested o_j. The remote CL of o_j is the local CL of objects that T_k have requested if any. T_i's commit influences the remote CL because those other transactions will wait until T_k completes validation of o_j. If T_k aborts, the objects that T_k is using will be released, and the other transactions will obtain the objects. We define the CL of an object as the sum of its local and remote CLs. Thus, the CL indicates how many transactions want the objects that a transaction is using.

If a parent transaction with a short execution time is enqueued instead of aborted, the queuing delay may exceed its execution time. Thus, RTS aborts a parent transaction with a short execution time. If a parent transaction with a high CL aborts, all closed-nested transactions will abort even if they have committed with their parent and will have to request the objects again. This may waste more time than a queuing delay. As long as their waiting time elapses, their CL may increase. Thus, RTS enqueues a parent transaction with a low CL. We discuss how to determine backoff times and CLs in Section 6.3.

6.3 Illustrative Example

RTS assigns different backoff times for each enqueued transaction. A backoff time is computed as a percentage of estimated execution time. Figure 6.1 shows a example of RTS. Three write transactions T_1, T_2, and T_3 request o_1 from the owner of o_1, and T_2 validates o_1 first at t_3. T_1 and T_3 abort due to the early validation of T_2. We consider two types of conflicts in RTS while T_2 validates o_1. First, a conflict between two write transactions can occur. Let us assume that write transactions T_4, T_5, and T_6 request o_1 at t_4, t_5, and t_6, respectively. T_4 is enqueued because the execution time $\mid t_4 - t_1 \mid$ of T_4 exceeds $\mid t_7 - t_4 \mid$ of T_2 — the expected commit time t_7 of T_2. At this time, the local CL of o_1 is 1 and the CL will be 2 (i.e., the CLs of o_3 + o_2 + o_1), which is a low



(a) Object-based Scenario (b) Transaction-based Scenario

Figure 6.1: A Reactive Transactional Scheduling Scenario

CL. Thus, $\mid t_7 - t_4 \mid$ is assigned to T_4 as a backoff time. When T_5 requests o_1 at t_5, even if $\mid t_5 - t_2 \mid$ exceeds $\mid t_5$ - expected commit time of $T_4 \mid$, T_5 is not enqueued because the CL is 4 (i.e., the local CL of o_1 is 2 and the CL of o_4 is 2), which is a high CL. Due to the short execution time of T_6, T_6 aborts. Second, a conflict between read and write transactions can occur. Let us assume that read transactions T_4, T_5, and T_6 request o_1. As backoff times, $\mid t_7 - t_4 \mid$, $\mid t_7 - t_5 \mid$, and $\mid t_7 - t_6 \mid$ will be assigned to T_4, T_5 and T_6, respectively. o_1 updated by T_2 will simultaneously be sent to T_4, T_5 and T_6, increasing the concurrency of the read transactions.

Given a fixed number of transactions and nodes, object contention will increase if these transactions simultaneously try to access a small number of objects. The threshold of a low or high CL relies on the number of nodes, transactions, and shared objects. Thus, the CL's threshold is adaptively determined. Assume that the CL's threshold in Figure 6.1 is decided as 3. When T_4 requests o_1, the CL for objects o_1, o_2, and o_3 is 2, meaning that two transactions want the objects that T_4 has requested, so T_4 is enqueued. On the other hand, when T_5 requests o_1, the CL of objects o_1 and o_4 is 4, representing that four transactions (i.e., more than the CL's threshold) want o_1 or o_4 that T_5 has requested, so T_5 aborts. As long as the waiting time elapses, their CL may increase. Thus, RTS enqueues a parent transaction with a low CL, which is defined as less than the CL's threshold.

To compute a backoff time, we use a transaction *stats table* that stores the average historical validation time of a transaction. Each table entry holds a *bloom filter* [85] representation of the most current successful commit times of write transactions. Whenever a transaction starts, an expected commit time is picked up from the table. The requesting message for each transaction includes three timestamps: the starting, requesting, and expected commit time of a transaction. In Figure 6.1, if T_5 is enqueued, its backoff time will be $\mid t_7 - t_5 \mid$ + the expected execution time (i.e., the expected commit - requesting time) of T_4.

If the backoff time expires before an object is received, the corresponding transaction will abort. Two possible cases exist in this situation. First, the transaction requests the object and is enqueued again as a new transaction. The duplicated transaction (i.e., the previously enqueued transaction) will be removed from a queue. Second, the object may be received before the transaction restarts. In this case, the object will be sent to the next enqueued transaction.

6.4 Algorithms

We now present the algorithms for RTS. There are three algorithms: Algorithm 4 for $Open\_Object$, Algorithm 5 for $Retrieve\_Request$, and Algorithm 6 for $Retrieve\_Response$. The procedure $Open\_Object$ is invoked whenever a new object needs to be requested. $Open\_Object$ returns the requested object if the object is received. The second procedure, $Retrieve\_Request$, is invoked whenever an object holder receives a new request from $Open\_Object$. Finally, $Retrieve\_Response$ is invoked whenever the requester receives a response from $Retrieve\_Request$. $Open\_Object$ has to wait for a response and $Retrieve\_Request$ notifies $Open\_Object$ of the response.

Algorithm 4 describes the procedure of $Open\_Object$. After finding the owner of the object, a requester sends oid, $txid$, $myCL$, and ETS to the owner. $myCL$ is set when an object is received. $myCL$ indicates the number of transactions needing the objects that the requester is using. The structure of an execution time (ETS) consists of the start time s, the requesting time r, and the expected commit time c of the requester. If the received object is null and the assigned backoff time is not 0, the requester waits for the backoff time. If it expires, $Open\_Object$ returns null and corresponding transaction retries. Otherwise, the requester wakes up and receives the object. The $TransactionQueue$ holding live transactions is used to check the status of the transactions. If a transaction aborts, it is removed from the $TransactionQueue$. In this case, even if an object is received, there is no transaction that needs the object, and therefore it is forwarded to the next transaction.

Algorithm 4: Algorithm of Open_Object

```
1  Procedure Open_Object
2  Input: Transaction_ID txid, Object_ID oid
3  Output: $null$, $object$
4  $owner$ = Find_owner($oid$);
5  Send $oid$, $txid$, $myCL$, and $ETS$ to $owner$;
6  Wait until that Retrieve_Response is invoked;
7  Read $object$, $backoff$, and $remoteCL$ from Retrieve_Response;
8  if $object$ is null then
9      if $backoff$ is not 0 then
10         TransactionQueue.put($txid$);
11         Wait for $backoff$;
12         Read $object$ and $backoff$ from Retrieve_Response;
13         if $object$ is not null then
14             return $object$;
15         else
16             TransactionQueue.remove($txid$);
17     return $null$;
18 else
19     return $object$;
```

The data structures depicted in Algorithm 1 is also used in Algorithms 5 and 6. Algorithm 5 describes $Retrieve\_Request$, which is invoked when an object owner receives a request. If get_Object gives null, it is not the owner of oid. Thus, 0 is assigned as the backoff and the requester must retry to find a new owner. If the corresponding object is locked, the object is being validated, so $Retrieve\_Request$ has to decide whether the requester is aborted or enqueued on ETS and $Contention\_Threshold$. Static variables bks represent backoff times for each object. An object owner holds as many bks as holding objects and updates corresponding bks whenever a transaction is enqueued. Unless the contention level of the requester and the object owner exceeds $Contention\_Threshold$, the requester is added to $scheduling\_List$. As soon as the object is unlocked, it is sent to the first element of $scheduling\_List$.

Algorithm 5: Algorithm of Retrieve_Request

```
Procedure Retrieve_Request
Input: $oid.\ txid$, $Contention\_Level$, $ETS$
$object$ = get_Object($oid$);
$address$ = get_Requester_Address();
Integer $backoff$ = 0;
if $object$ is not null and in use then
    Requester_List $reqlist$ = scheduling_List.get($oid$);
    if $reqlist$ is null then
        $reqlist$ = new Requester_List();
    else
        $reqlist$.removeDuplicate($address$);
    if $bk$ < | $ETS.r$ - $ETS.s$ | then
        Integer contention = $reqlist$.getContention()+$Contention\_Level$;
        if contention < CL_Threshold then
            $bk$ += | $ETS.c$ - $ETS.r$ |; $backoff$ = $bk$;
            $reqlist$.addReqeuster($contention$, new Requester($address$, $txid$));
            scheduling_List.put($oid$, $reqlist$);
Send $object$ and $backoff$ to $address$;
```

In Algorithm 6, $Retrieve\_Response$ sends $Object\_Open$ a signal to wake up if a transaction waits for an object. If any transaction needing the object is not located in $TransactionQueue$, let the object's owner send the object to the next element of $scheduling\_List$. If a transaction completes the validation of objects (i.e., commit), the node invoking the transaction receives $Requster_List$s of each committed object. The newly updated object will be sent to the first element of $scheduling\_List$.

Whenever an object is requested, RTS performs Algorithms 4, 5, and 6. We use a hash table for objects and a linked list for transactions. The transactions will be enqueued as many as CL threshold. The time complexity is $O(1)$ to enqueue a transaction. To check duplicated transactions in all enqueued transactions, the time complexity is $O(CL\ threshold)$. Thus, the total time complexity of RTS is $O(CL\ threshold)$.

Algorithm 6: Algorithm of Retrieve_Response

```
1 Procedure Retreive_Response
2 Input: $object$, $txid$, and $backoff$
3 if $txid$ is found in TransactionQueue then
4     TransactionQueue.remove($txid$);
5     Send a signal to wake up and give $object$ and $backoff$;
6 else
7     Send a message to the object owner;
```

6.5 Analysis

We now show that RTS outperforms another scheduler in speed. Recall that RTS uses TFA to guarantee a consistent view of shared objects between distributed transactions, and ensure atomicity for object operations. In [23], TFA is shown to exhibit opacity (i.e., its correctness property) [86] and strong progressiveness (i.e., its progress property [87]). In the worst case, N transactions are simultaneously invoked to update an object. Whenever a conflict occurs between two transactions, let scheduler B abort one of these and enqueue the aborted transaction (to avoid repeated aborts) in a distributed queue. The aborted transaction is dequeued and restarts after a backoff time. Let the number of aborts of T_i be denoted as λ_i. We have the following lemma.

Lemma 6.5.1. *Given scheduler B and N transactions, $\sum_{i=1}^{N} \lambda_i \leq N-1$.*

Proof. Given a set of transactions $T = \{T_1, T_2, \cdots T_N\}$, let T_i abort. When T_i is enqueued, there are δ_i transactions in the queue. T_i can only commit after δ_i transactions commit if δ_i transactions have been scheduled. Hence, if a transaction is enqueued, it does not abort. Thus, one of N transactions does not abort. The lemma follows. □

Let node n_0 hold an object. We have the following two lemmas.

Lemma 6.5.2. *Given scheduler B and N transactions, $makespan_B^N(NR) \leq 2(N-1)\sum_{i=1}^{N} d(n_0, n_i) + \sum_{i=1}^{N} \gamma_i$.*

Proof. Lemma 6.5.1 gives the total number of aborts on N transactions under scheduler B. If a transaction T_i requests an object, the communication delay will be $2 \times d(n_0, n_i)$. Once T_i aborts, this delay is incurred again. To complete N transactions using scheduler B, the total communication delay will be $2(N-1)\sum_{i=1}^{N} d(n_0, n_i)$ and the total local execution time will be $\sum_{i=1}^{N} \gamma_i$. □

Lemma 6.5.3. *Given scheduler RTS and N transactions, $makespan_{RTS}^N(NR) \leq \sum_{i=1}^{N} d(n_0, n_i) + \sum_{i=1}^{N} d(n_{i-1}, n_i) + \sum_{i=1}^{N} \gamma_i$.*

Proof. Given a set of transactions $T = \{T_1, T_2, \cdots T_N\}$, which is ordered in the queue of node n_0, if $\forall T_i \in T$ requests an object, the communication delay of requesting an object will be

$\sum_{i=1}^{N} d(n_0, n_i)$. The total communication delay to complete N transactions will be $\sum_{i=1}^{N} d(n_0, n_i) + \sum_{i=1}^{N} d(n_{i-1}, n_i)$ and the total local execution time will be $\sum_{i=1}^{N} \gamma_i$. □

We have so far assumed that all N transactions share an object to study the worst-case contention. We now consider contention of N transactions with M objects. We have the following theorem.

Theorem 6.5.4. *Given N transactions and M objects, the RCR of schedulers RTS and B is less than 1, where $N \geq 2$.*

Proof. Consider a transaction that includes multiple nested-transactions and accesses multiple shared objects. In the worst case, the transaction has to update all shared objects. $makespan_{RTS}^{N}(NR) < makespan_{B}^{N}(NR)$ because $\frac{\sum_{i=1}^{N} d(n_{i-1},n_i)}{\sum_{i=1}^{N} d(n_0,n_i)} < 2N - 3$. The best case of scheduler B for aborted transactions is that its communication delays for M objects to visit all nodes invoking N transactions is incurred on shortest paths. Thus, $\frac{\sum_{i=1}^{N} d(n_{i-1},n_i)}{\sum_{i=1}^{N} d(n_0,n_i)} < \log N$ [80]. Hence, $M \times \log N < M \times (2N - 3)$, when $N \geq 2$. The theorem follows. □

6.6 Evaluation

We implemented RTS in the HyFlow DTM framework [23] for experimental studies. We developed a set of six distributed applications as benchmarks. These include distributed versions of the Vacation benchmark of the STAMP benchmark suite [69], Bank as a monetary application [23], and four distributed data structures including Linked-List (LL), Binary-Search Tree (BST), Red/Black Tree (RB-Tree), and Distributed Hash Table (DHT) [38] as microbenchmarks. We used *low* and *high contention*, which are defined as 90% and 10% read transactions of one thousand active concurrent transactions per node, respectively [7]. A read transaction includes only read operations, and a write transaction consists of both read and write operations. Five to ten shared objects are used at each node. Communication delay between nodes is limited to a number between 1 and $50msec$ to create a static network.

Under long execution time and large CL's threshold, Vacation and Bank benchmarks suffer from high contention because their queueing delay is longer than that of the other benchmarks. In the mean time, under long execution time and short CL's threshold, the aborts of parent transactions increase. At a certain point of the CL's threshold, we observe a peak point of throughput. Thus, in this experiment, the CL's threshold corresponding to the peak point is determined.

We measured the throughput (i.e., the number of committed transactions per second) of RTS, TFA, and TFA+Backoff. TFA means TFA without any transactional scheduler supporting closed-nested transactions [79]. The purpose of measuring the throughput of TFA is to understand the overall performance improvement of RTS. TFA+Backoff means TFA utilizing a transactional scheduler. With the scheduler, a transaction aborts with a backoff time if a conflict occurs. The purpose of measuring TFA+Backoff's throughput is to understand the effectiveness of enqueuing live transactions to prevent the abort of nested transactions.

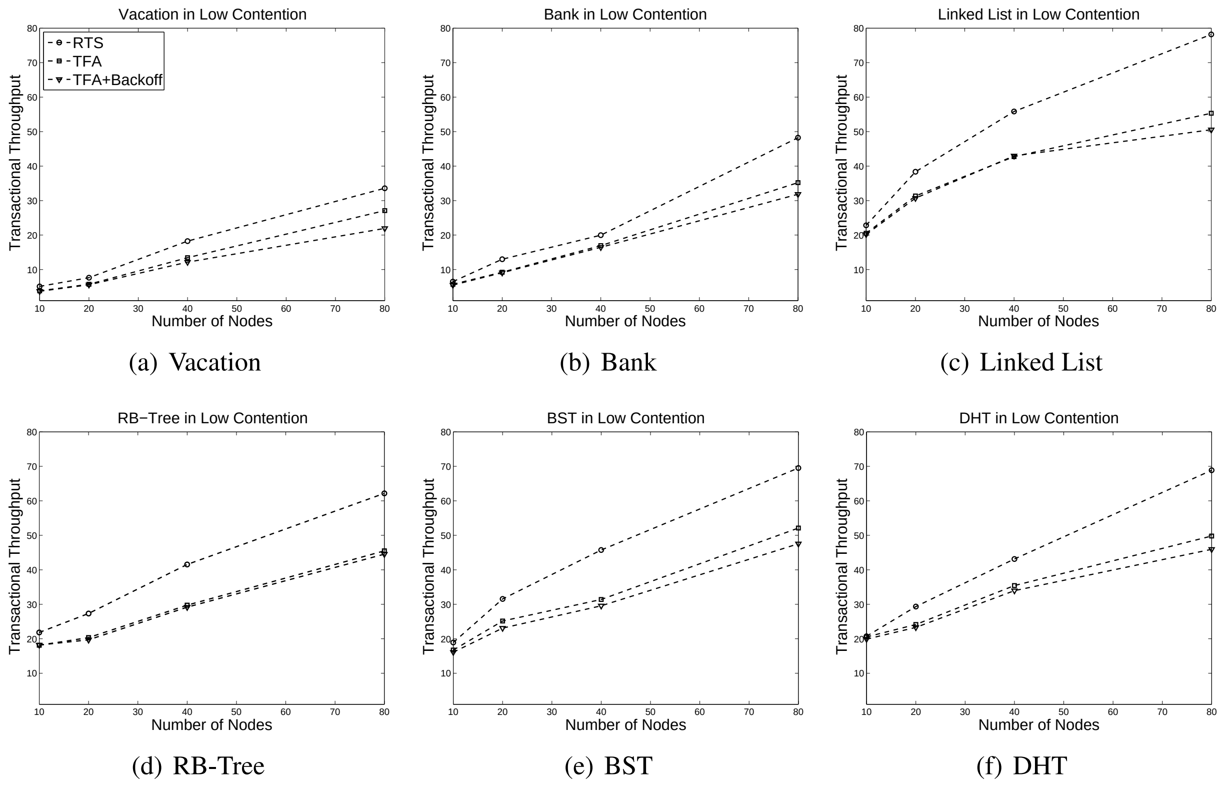
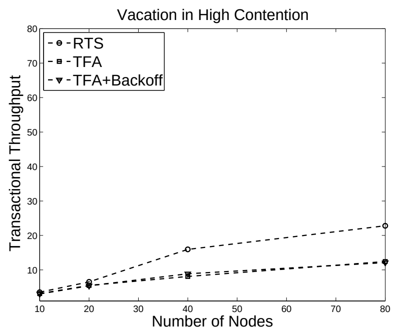


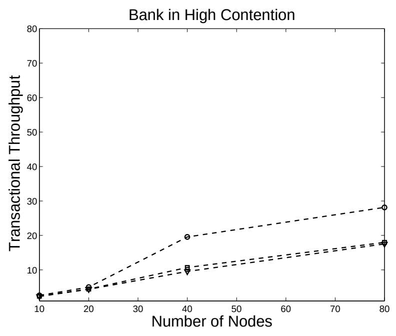
Figure 6.2: Transactional Throughput on Low Contention

Figure 6.2 shows the throughput at low contention (i.e., 90% read transactions) for each of the six benchmarks, running on 10 to 80 nodes. From Figure 6.2, we observe that RTS outperforms TFA and TFA+Backoff. Generally, TFA's throughput is better than TFA+Backoff's. If a parent transaction including multiple nested transactions aborts, it requests all the objects again under TFA+Backoff. Even if the parent transaction waits for a backoff time, the additional requests incur more contention, so the backoff time is not effective for nested transactions. Under TFA, an aborted transaction also requests all objects without any backoff, also incurring more contention. From Figures 6.2(a) and 6.2(b), we observe that Vacation and Bank benchmarks take longer execution time than others. The improvement of their throughput is less pronounced.

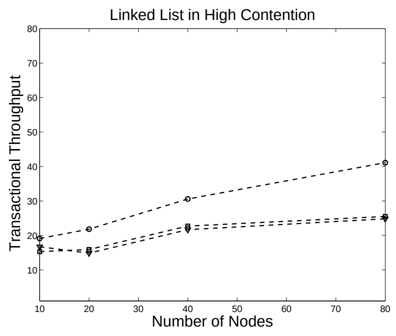
Figure 6.3 shows the throughput at high contention (i.e., 10% read transactions) for each of the six benchmarks. We observe that the throughput is less than that at low contention, but RTS's speedup over others increases. High contention leads to many conflicts, causing nested transactions to abort. Also, we observe that a long execution time caused by queuing live transactions incurs a high probability of conflicts. In Figures 6.3(c), 6.3(d), 6.3(e), and 6.3(f), the throughput is better than that of Bank and Vacation, because LL, RB Tree, BST, and DHT have relatively short local



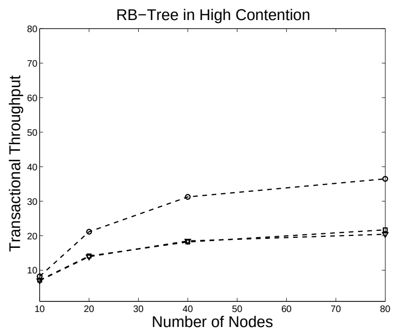
(a) Vacation



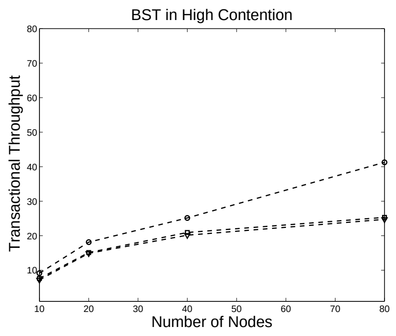
(b) Bank



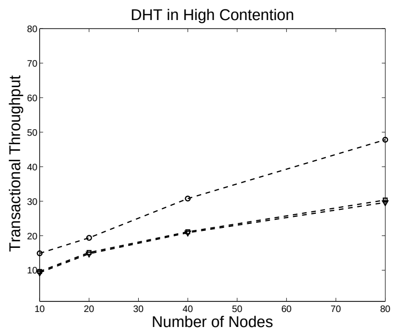
(c) Linked List



(d) RB-Tree



(e) BST



(f) DHT

Figure 6.3: Transactional Throughput on High Contention

execution times.

We computed the throughput speedup of RTS over TFA and TFA+Backoff – i.e., the ratio of RTS's throughput to that of the respective competitors. Figure 6.4 summarizes the speedup. Our experimental evaluations reveal that RTS improves throughput over DTM without RTS by as much as 1.53 (53%) $\sim$ 1.88 (88%) $\times$ speedup in low and high contention, respectively.

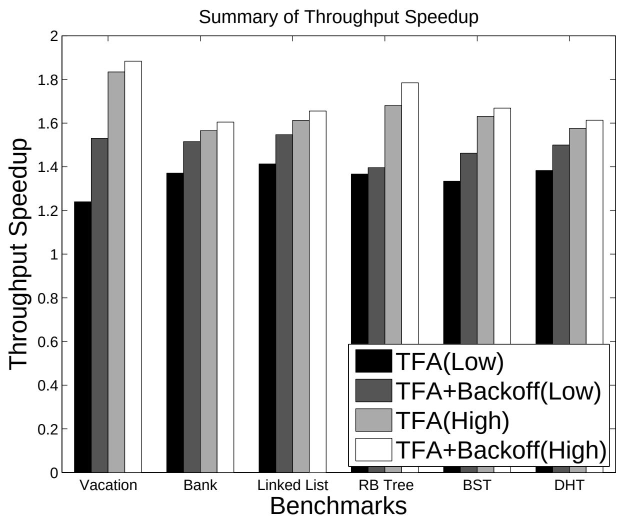


Figure 6.4: Summary of Throughput Speedup

Chapter 7

The Dependency-Aware Transactional Scheduler

7.1 Motivation

Figure 7.1 shows an example of open-nested transactions with compensating actions and abstract locks. Listings 7.1 and 7.2 in Figure 7.1 illustrate two outer transactions, T_1 and T_2, and an inner transaction in Listing 7.3. The inner transaction INSERT includes an $insert$ operation in a Linked List. T_1 has a $delete$ operation with a value. If the operation of T_1 executes successfully, its inner transaction INSERT executes. Conversely, regardless of the success of T_2's $delete$ operation, its inner transaction INSERT will execute. $OnCommit$ and $OnAbort$, which include a compensating action, are registered when the inner transaction commits. If the outer transaction (i.e., T_1 or T_2) commits, $OnCommit$ executes. When the inner transaction commits, its modification becomes immediately visible for other transactions. Thus, if the inner transaction commits, and its outer transaction T_1 or T_2 aborts, a $delete$ operation as a compensating action (described in $OnAbort$) executes. Let us assume that T_2 aborts, and $OnAbort$ executes. Even though T_2's inner transaction (INSERT) does not depend on its $delete$ operation, unlike T_1, $OnAbort$ will execute. Thus, the conflict of object "tree-2" in T_2 causes the execution of compensating action on object "tree-1" in INSERT. The INSERT operation acquires the abstract lock again when it restarts. Finally, whenever an outer transaction aborts, its inner transaction must execute a compensating action, regardless of the operation's dependencies.

This drawback is particularly evident in distributed settings. In fact, distributed transactions typically have an execution time several orders of magnitude bigger than in a centralized STM, due to communication delays that are incurred in requesting and acquiring objects [34]. If an outer transaction aborts, clearly the impact of the time needed for running compensating actions and for

Listing 7.1: Transaction T_1

```
new Atomic<Boolean >(){
  @Override boolean atomically(Txn t){
    List ll = (List)t.open(tree -2);
    deleted = ll.delete(7,t);
    if(deleted) INSERT(t,10); //inner transaction
      return deleted;
  }
}
```

Listing 7.2: Transaction T_2

```
new Atomic<Boolean >(){
  @Override boolean atomically(Txn t){
    List ll = (List)t.open(tree -2);
    deleted = ll.delete(9,t);
    INSERT(t,10); //inner transaction
    return deleted;
  }
}
```

Listing 7.3: Inner Transaction INSERT

```
public boolean INSERT(Txn t, int value){
  private boolean inserted = false;
  @Override boolean atomically(t){
    List ll = (List)t.open(tree -1);
    inserted = ll.insert(value,t);
    t.acquireAbstractLock (ll,value);
    return inserted;
  }
  @Override onAbort(t){
    List ll = (List)t.open(tree -1); //compensation action
    if(inserted)ll.delete(value,t);
    t.releaseAbstractLock(ll,7);
  }
  @Override onCommit(t){
    List ll = (List)t.open(tree -1);
    t.releaseAbsractLock(ll,value);
  }
}
```

Figure 7.1: Two open-nested transactions with abstract locks and compensating actions

acquiring abstract locks for distributed open-nested transactions is exacerbated due to the communication overhead. Moreover it increases the likelihood of conflicts, drastically reducing concurrency and degrading performance.

Motivated by these observations, we propose the DATS scheduler for open-nested DTM. DATS, for each outer transaction T_a, identifies the number of inner transactions depending from T_a and schedules the outer transactions with the greatest number of dependencies to validate first and (hopefully) commit. This behavior permits the transactions with high compensation overhead to commit; the remaining few outer transactions that are invalidated will be restarted excluding their independent inner transactions to avoid useless compensating actions and acquisition of abstract locks. In the next subsection the meaning of dependent transactions for DATS will be described.

7.2 Abstract and Object Level Dependencies

7.2.1 Abstract Level Dependency

Abstract level dependency (ALD) indicates the dependency between an outer transaction and its inner transactions at an abstract level. We define the *dependency level* (DL) as the number of inner transactions that will execute $OnAbort$ when the outer transactions abort. For example, T_1 illustrated in Figure 7.1 depends on its INSERT due to the $deleted$ variable. Thus, DATS detects a dependency between T_1 and its INSERT (its inner transaction) because the $delete$ operations in T_1 shares the variable $deleted$ with the conditional if statement declared for executing INSERT. In this case, the DL=1 for T_1. Conversely, T_2 executes INSERT without checking any pre-condition so its DL=0 because T_2 does not have dependencies with its inner transactions. The purpose of the abstract level dependency is to avoid unnecessary compensating actions and abstract locks. Even though T_2 aborts, $OnAbort$ in INSERT will not be executed because its DL=0, and the compensating action will not be processed. Meanwhile, executing $OnAbort$ implies running INSERT and acquiring the abstract lock again when T_2 restarts.

Summarizing, aborting outer transactions with smaller DLs leads to a reduced number of compensating actions and abstract lock acquisitions. Such identification can be done automatically at run-time by DATS using byte-code analysis or relying on explicit indication by the programmer. The first scenario is completely transparent from the application point of view but in some cases could add additional overhead. The second approach, although it requires the collaboration of the developer, is more flexible because it allows the programmer to bias the behavior of the scheduler. In fact, even though the logic of an outer transaction reveals a certain number of dependencies, the programmer may want to force running compensations in case of an abort. This can be done by simply changing the value of DL associated to the outer transaction.

7.2.2 Object Level Dependency

Object level dependency (OLD) indicates the dependency among two or more concurrent transactions accessing the same shared object. For example, in Figure 7.1, T_1 depends on T_2 because they share the same object "tree-2". If T_1 and T_2 work concurrently, a conflict between them occurs. However, $delete(7)$ of T_1 and $delete(9)$ of T_2 commute because they are two operations executing on the same object ("tree-2") but accessing different items (or fields when applicable) of the object (item "7" and item "9"). We recall that, two operations commute if applying them in either order they leave the object in the same state and return the same responses [43]. DATS detects object level dependency at transaction commit phase, splitting the validation phase into two. Say T_a is the transaction that is validating. In the first phase, T_a checks the consistency of the objects requested during the execution. If a concurrent transaction T_b has requested and already committed a new version of some object requested by T_a, then T_a aborts in order to avoid isolation corruption. After the successful completion of the first phase of T_a's validation, DATS detects the object level dependencies among concurrent transactions that are validating with T_a in the second phase. To do that, DATS relies on the notion of commutativity: Two transactions are defined as commutable if they conflict and they leave the state of the shared data-set consistent even if validated and committed concurrently.

A very intuitive example of commutativity is when two operations, $call1(X)$ and $call2(X)$, both access the same object X but different fields of X. Suppose T_a and T_b are conflicting transactions but simultaneously validating. If all of T_a's operations commute with all of T_b's operations, they can proceed to commit together avoiding a useless abort. Otherwise one of T_a or T_b must be aborted. This scheduler is in charge of the decision (see next sub-section).

In order to compute commutativity, DATS joins two supports. In the first, the programmer annotates each transaction class with the fields accessed. The second is a field-based timestamping mechanism, used for checking the field-level invalidation. The goal is to reduce the granularity of the timestamp from object to field. With a single object timestamp, it is impossible to detect commutativity because of fields modifications. In fact, writes to different fields of the same object are all reflected with the increment of the same object timestamp. In order to do that efficiently, DATS exploits the annotations provided by the developer on the fields accessed by the transaction to directly point only to the interested fields (instead of iterating on all the object fields, looking for the ones modified). On such fields, it uses field-based timestamping to detect object invalidation.

The purpose of the object level dependency is to enhance concurrency of outer transactions. Even though inner transactions terminate successfully, aborting their outer transactions affects these inner transactions (due to compensation). Thus, DATS checks for the commutativity of conflicting transactions and permits them to be validated, reducing the aborts.

Algorithm 7: Algorithms for checking AOL and OLD

```
Procedure Commit
Input: $txid$, $objects$
Output: $commit$, $abort$
foreach objects do
    if $txid$ is open nesting then
        ▷ Extract <operations,values,DL>
        Send <operations,values,DL,object.id>
                o object.owner
        Wait until receive status from object.owner
        if status=$noncommute$ then
            noncommutativity.put(object);
if noncommutativity=$\emptyset$ then
    ▷ All objects commute or no conflicts detected
    Retrieve the dependency queue from object.owner;
    Validate objects; ▷ Change the object ownership
    find highest $DL$ from dependency.get(object.id);
    Send object to the node with the highest $DL$;
    return $commit$;
foreach object $\in$ noncommutativity do
    ▷ Checking abstract level dependency (ADL)
    nestedTxId = CheckALD(object);
    ▷ Enqueue dependent nested transactions
    NestedTxs.put(object.id,nestedTxId);
Abort($txid$, $DependentObjects$);
return $abort$;
Procedure Retrieve_Object
Input: $operation$, $value$, $DL$, $oid$
object = findObject(oid);
if object=null then
    ▷ Object just validated, checking object level dependency (OLD)
    if CheckOLD(operation, value) then
        commutativity.put(object.id, new request(operation, values));
        return $commute$;
    ▷ Dependency queue to track updates.
    dependency.put(oid, $DL$);
    return $non-commiute$;
return $no-conflict$;
Procedure Abort
Input: $txid$, $objects$
if $txid$ is outer-transaction then
    foreach objects do
        nestedIds = NestedTxs.get(object.id);
        if nestedIds $\neq$ null then
            foreach nestedIds do
                ▷ Execute onAbort() for nestedId
                AbortNestedTx(nestedId);
AbortOuterTx($txid$);
```

7.3 Scheduler Design

We designed DATS using abstract level dependencies and object level dependencies. Algorithm 7 shows the pseudo-code with the procedures used by DATS for detecting ALD and OLD at validation/commit time. When outer transactions are invoked, the DL with their inner transactions is checked. When the outer transactions request an object from its owner, the requests with their DLs will be sent to the owner and moved into its scheduling queue. The object owner maintains the scheduling queue holding all the ongoing transactions that have requested the object with their DLs. When T_1 (one of the outer transactions) validates an object, we consider two possible scenarios. First, if another transaction T_2 tries to validate the same object, a conflict between T_1 and T_2 is detected on the object. Thus, DATS checks for the object level dependency. If T_1 and T_2 are independent (according to the object level dependency rules), DATS allows T_1 and T_2 to proceed with the validation. Otherwise, the transaction with lower DL will be aborted. In this way, dependent transactions with the minimal cost of abort and compensating actions are aborted and restarted, permitting transactions with a costly abort operation to commit.

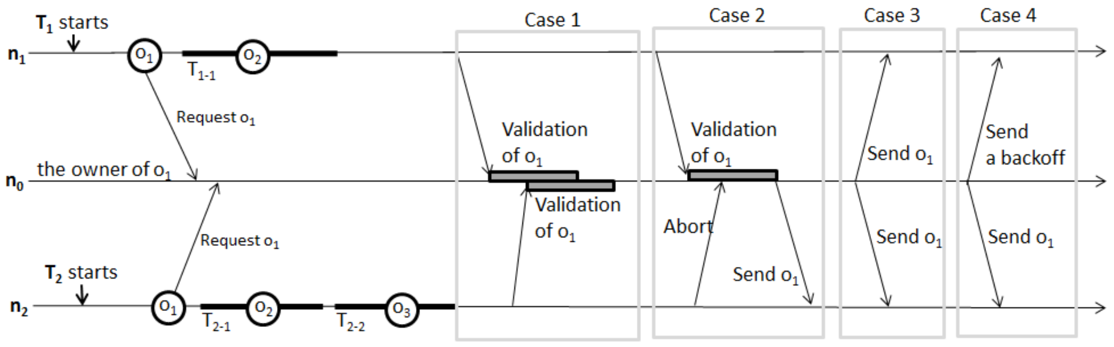


Figure 7.2: Four Different Cases for Two Transactions T_1 and T_2 in DATS

Figure 7.2 illustrates an example of DATS with two transactions T_1 and T_2 invoked on nodes n_1 and n_2, respectively. The transaction T_1 has a single inner-transaction and T_2 has two nested transactions. Let us assume that T_1's DL=1 and T_2's DL=2. The circles indicate written objects. The horizontal line corresponds to the status of each transaction described in the time domain. Figure 7.2 shows four different cases when T_1 and T_2 terminate. When T_1 and T_2 are invoked, DATS analyzes their DLs, operations, and values. When T_1 requests o_1 from n_0, the meta-data for DLs, operations and values of o_1 will be sent to n_0. These are moved to the scheduling queue of n_0. We consider four different cases regarding the termination of T_1 and T_2.

Case 1. T_1 and T_2 validate concurrently o_1. DATS checks for the object level dependency. If T_1 and T_2 are not dependent at the object level (i.e., the operations of T_1 and T_2 over o_1 commute), T_1 and T_2 commit concurrently.

Case 2. T_1 starts to validate and detects it is dependent with T_2 (that is still executing) at the object level on the object o_1. In this case T_2 will abort due to early validation. When T_1 commits, the updated o_1 is sent to n_2.

Case 3. Another transaction committed o_1 before T_1 and T_2 validate. If T_1 and T_2 are not dependent at the object level, o_1 is sent to n_1 and n_2 simultaneously as soon as the transaction commits.
Case 4. Another transaction committed o_1 before T_1 and T_2 validate. If T_1 and T_2 are dependent at the object level, DATS checks for the abstract level dependency, and o_1 is sent to n_2 because T_2's DL is larger than that of T_1. Aborting T_1, the scheduler is forced to run a single compensation (for T_{1-1}) instead of two compensations (T_{2-1} and T_{2-2}) in case of T_2's abort. Further, considering the case in which the DL of T_1 is 0, the abort of T_1 does not affect T_{1-1}. In fact, its execution will be preserved and only the operations of T_1 will be re-executed.

7.4 Evaluation

7.4.1 Experimental Setup

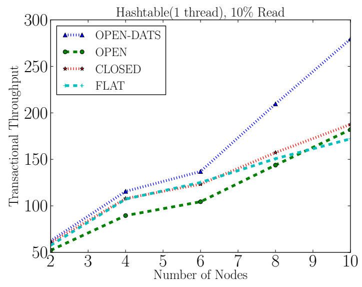
We implemented DATS in the HyFlow DTM framework [79]. We cannot compare our results with any competitor, as none of the DTMs that we are aware of support open nesting and scheduling. Thus, we compared DATS under TFA-ON (DATS) with only TFA-ON (OPEN) [78], closed nested transaction (CLOSED) [79], and flat nested transaction (FLAT). We contrast with CLOSED and FLAT to show that OPEN does not always perform better than them, while DATS consistently outperforms OPEN.

We assess the performance of DATS using Hash Table, Skip List and Linked List as micro-benchmarks, TPC-C [83] as a real-application benchmark. Our test-bed is comprised of 10 nodes, each one is an Intel Xeon 1.9GHz processor with 8 CPU cores. We varied the number of application threads performing operations for each node from 1 to 8, considering a spectrum between 2 and 80 concurrent threads in the system. We measured the *throughput* (number of committed transactions per second). All data-points reported are the result of multiple executions, so plots present for each data-point the mean value and the error-bar. In order to assess the goodness of DATS we also present the percentage of aborted transactions and the scheduler overhead.

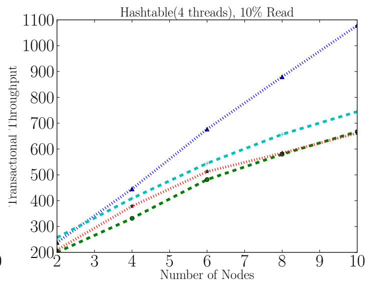
7.4.2 Benchmarks

The Skip List and Linked List benchmarks are data structures maintaining sorted and unsorted, lists of items, respectively, whereas Hash Table is an associative array mapping keys to values. We configured the benchmarks with the small number of objects and a large number of inner transactions – eight inner transactions per transaction and ten objects, incurring high contention.

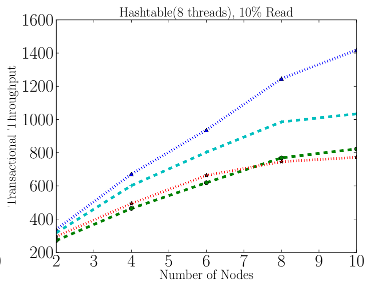
Regarding TPC-C, the write transactions consist of update, insert, and/or delete operations accessing a database of nine tables maintained in memory, where each row has a unique key. Multiple op-



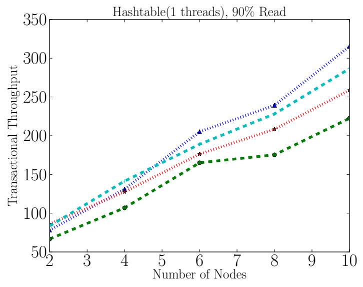
(a) HT (1 thread), 10% Read



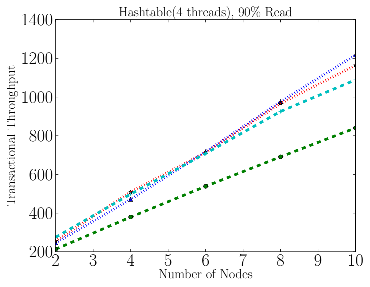
(b) HT (4 threads), 10% Read



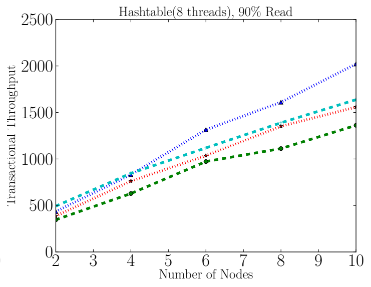
(c) HT (8 threads), 10% Read



(d) HT (1 thread), 90% Read



(e) HT (4 threads), 90% Read



(f) HT (8 threads), 90% Read

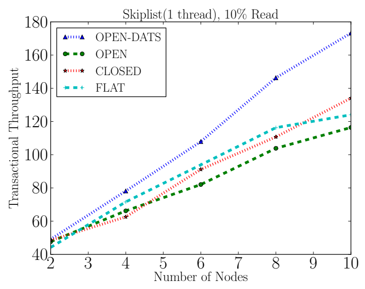
Figure 7.3: Performance of DATS Using Hash Table (HT).

erations commute if they access a row (or object) with the same key and modify different columns. We configured the benchmark with a limited number of warehouses (#3) in order to generate high conflicts. We recall that, in the data flow model, objects are not bound on fixed nodes but move, increasing likelihood of conflicts.

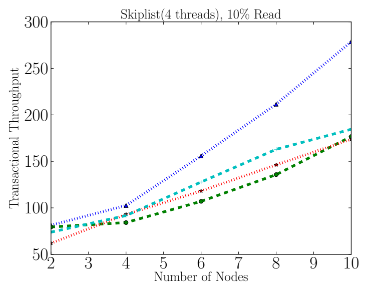
7.4.3 Evaluation

Figures 7.3, 7.4 and 7.5(a-f) show the throughput of micro-benchmarks under 10% and 90% of read transactions. The purpose of DATS is to reduce the overheads of compensating actions and abstract locks. In 10% read transactions, the number of aborts increases due to high contention. Outer transactions frequently abort, and corresponding compensating actions are executed; so DATS outperforms OPEN in throughput because it mitigates the abort of outer transactions and the corresponding compensating actions.

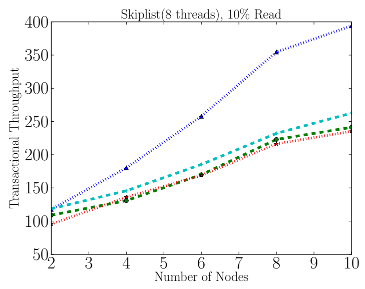
For the experiments with TPC-C in Figure 7.5(g),7.5(h),7.5(i), we used the amount of read and write transactions that its specification recommends. TPC-C benchmark accesses large tables to read and write values. Due to the non-negligible transaction execution time, the number of compensating actions and abstract locks in TPC-C significantly degrades the overall performance. Thus, DATS increases the performance in high contention (a large number of threads and nodes). By these results, it is evident how much unnecessary aborts of inner transactions affects performance and how much performance is improved through minimizing aborts. Even if DATS reduces the



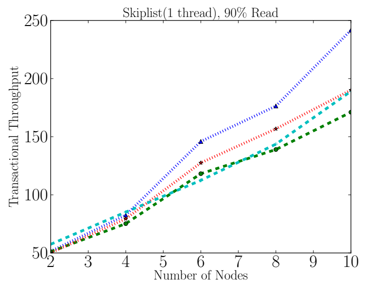
(a) SL (1 thread), 10% Read



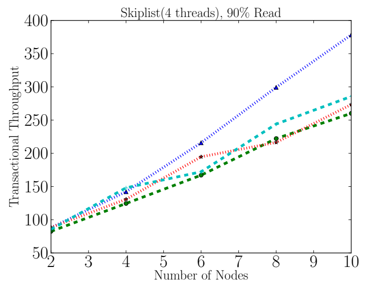
(b) SL (4 threads), 10% Read



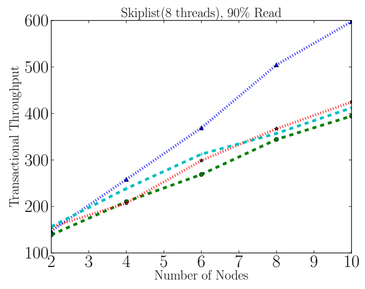
(c) SL (8 threads), 10% Read



(d) SL (1 thread), 90% Read



(e) SL (4 threads), 90% Read



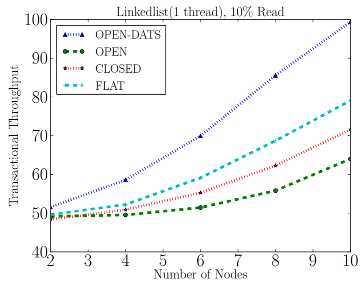
(f) SL (8 threads), 90% Read

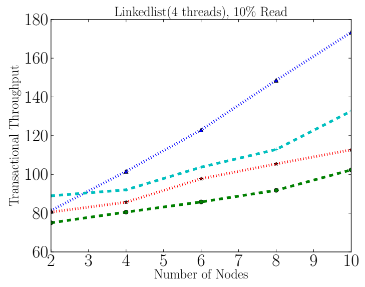
Figure 7.4: Performance of DATS Using Skip List (SL).

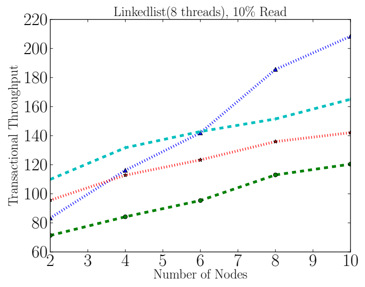
number of compensating actions and acquisition of abstract locks, the performance of OPEN is degraded because of the commit overheads of inner transactions [78]; so the throughput of DATS is slightly better than CLOSED and FLAT, but significantly better than OPEN.

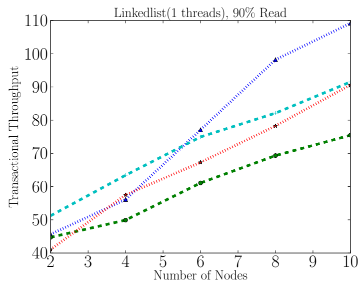
Figure 7.7 shows throughput speed-up relative to OPEN using Hash Table, Skip List, Linked List and TPC-C. Our results show that DATS performs up to 1.7× and 2.2× better than OPEN in micro-benchmarks and TPC-C, respectively.

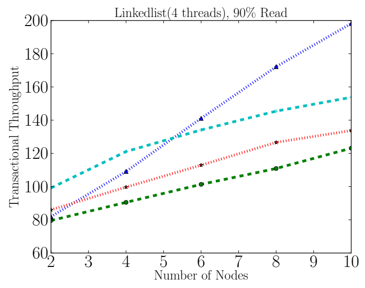
Figure 7.6 shows the analysis of scheduling overhead and abort reduction. Checking dependencies occurs when a transaction validates, so we measure the average execution time and the average validation time of committed transactions as illustrated in Figure 7.6(b). The gap between the two validation times of DATS and OPEN proves the scheduling overhead. Even though the validation time of DATS is up to two times more than OPEN's, a large number of transactions validated simultaneously according to the increment of nodes, results in a shorten transaction response time, reducing the average validation time and aborts. Figure 7.6(a) the comparison between the percentage of aborted transactions of OPEN and DATS. As long as the number of threads increases, the number of aborts in DATS and OPEN increases too. However, the increasing abort ratio in DATS is less than in OPEN, proving how much DATS reduces the abort rate.

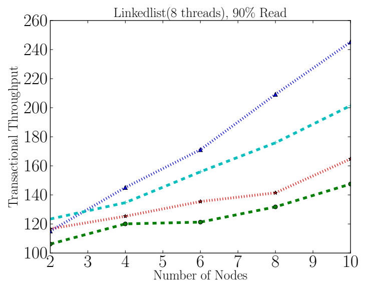

(a) LL (1 thread), 10% Read

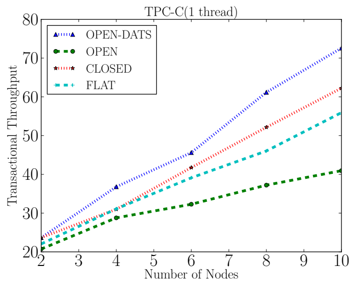

(b) LL (4 threads), 10% Read

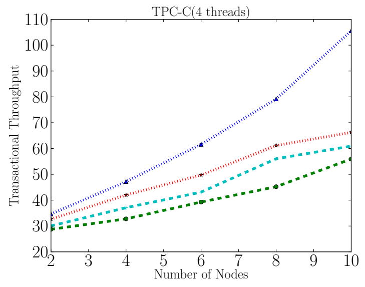

(c) LL (8 threads), 10% Read


(d) LL (1 thread), 90% Read


(e) LL (4 threads), 90% Read


(f) LL (8 threads), 90% Read


(g) TPC-C(1 thread)


(h) TPC-C(4 threads)

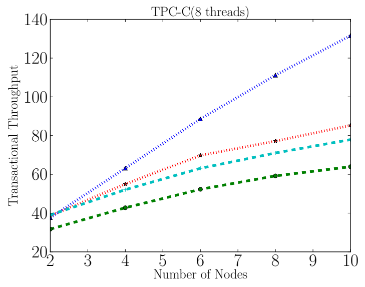
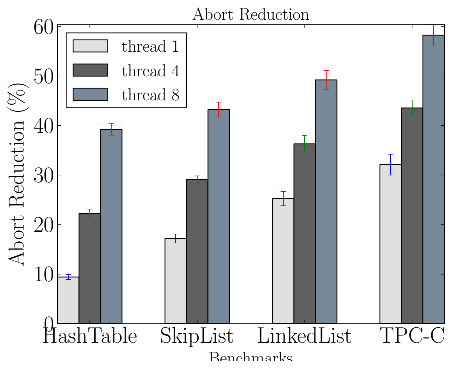

(i) TPC-C(8 threads)

Figure 7.5: Performance of DATS Using Linked List (LL) and TPC-C.


(a) % Aborted transactions

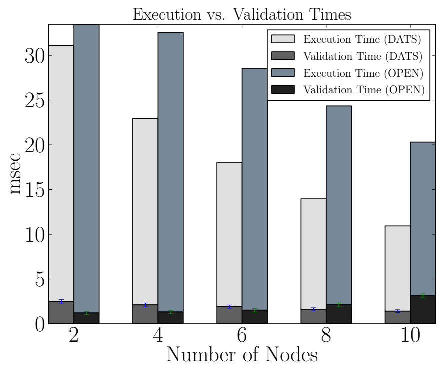
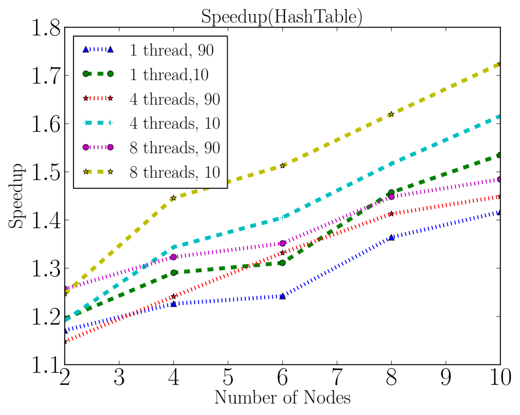
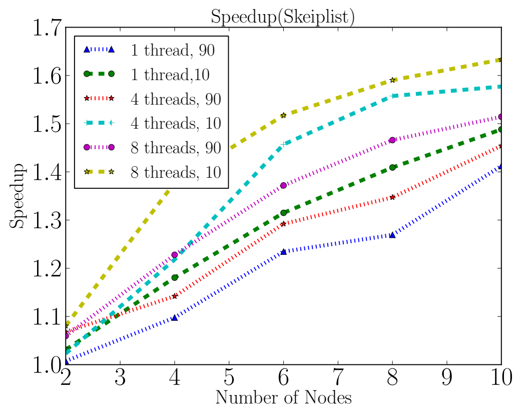

(b) Execution vs. Validation Times

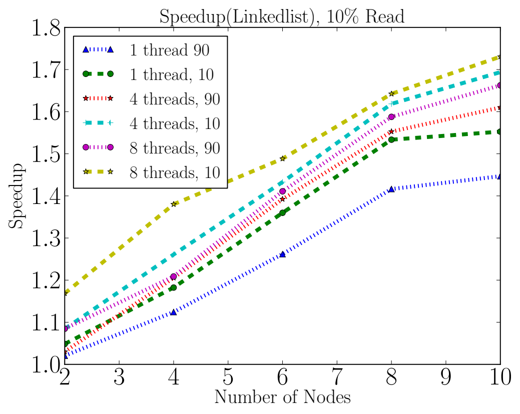
Figure 7.6: Analysis of Scheduling Overhead and Abort Reduction.



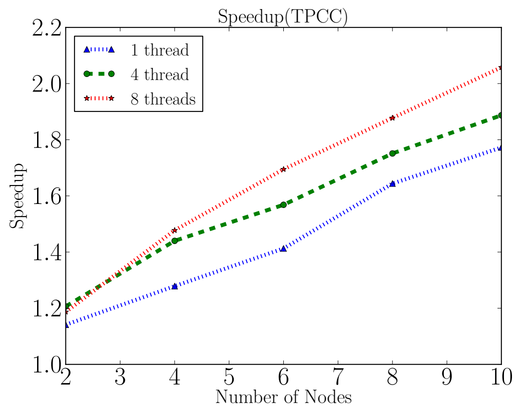
(a) Hash Table



(b) Skip List



(c) Linked List



(d) TPC-C

Figure 7.7: Speed-up (Throughput Relative to OPEN) in Hash Table, Skip List, Linked List, TPC-C.

Chapter 8

Scheduling-based Parallel Nesting

8.1 Motivation

The execution of nested inner transactions in the context of a parent transaction can be conceptually represented by a dynamic tree, called transaction tree, in which transactions represent the vertex of the tree and edges are used for defining the conflict relation between transactions. The topology of the tree is not defined a-priori. Originally, all the inner transactions belong to the same level of the tree and their parent represent the parent transaction. Sibling transactions (belonging to the same level of the transaction tree) are executed in parallel, assuming their conflict independence. The approach does not assume previous knowledge on transaction conflicts, therefore some (or all) of the sibling transactions cannot execute in parallel due to transaction dependencies. When a conflict happens, the aborted transactions is moved on a lower level with an edge representing the just detected dependency. In case all the inner transactions are not conflicting with the others, the parallel process allows to execute only the inner conflicting transaction on the critical path and the others in parallel.

In closed nesting, all the inner transactions must commit successfully for triggering the parent's commit. In case they are independent, they can be executed and committed in parallel allowing the parent transaction to commit just after the longest inner transaction completes its execution. However, if there is dependency among them, conflicts occur, so their parallel activation may not be effective. Figure 8.1 shows new order transaction T_1 [83] including multiple inner transactions. T_1 opens warehouse and district to extract a tax and stock to get a price, the first two inner transactions, respectively. These two inner transactions do not have dependency. Executing two transactions in parallel may lead to high performance.

Closed-nesting performs better than flat-nesting and the program model of closed-nesting differs

Listing 8.1: Transaction T_1

```
Atomic{
    // innerTxs: a number of inner transactions
    String w_id = rand(WAREHOUSES) + 1;
    String d_id = rand(DISTRICT) + 1;
    String c_id = rand(CUSTOMER) + 1;
    Atomic{
      // In the warehouse table: retrieve an object
      Object warehouse = (warehouse)open(w_id);
      W_TAX = warehouse.W_TAX
      // In the district table: retrieve D_TAX, get and inc D_NEXT_O_ID
      Object district = (district)open(w_id, d_id);
      D_TAX = district.D_TAX;
      // In the customer table: retrieve an object
    }
    for(i=0; i< GetItemList(); i++){
      Atomic{
        Object stock = (Stock)open(w_id, d_id, c_id, i);
        stock.quantity = getQuantity();
        stock.order_cnt++;
        Price = stock.price;
      }
      Atomic{
        Object customer = (customer)open(w_id, d_id, c_id);
        Discount = customer.Discount;
        // Create entries in ORDER
        Object order = new TpccOrder(w_id, d_id, o_id, i)
        order.Supply_W_ID = w_id;
        order.delivery = null;
        order.totalAmount = Price*(1-Discount)*(1+W_TAX+D_TAX);
      }
    }
  }
}
```

Figure 8.1: New Order Transaction with multiple Inner Transactions in TPC-C

from open-nesting's. Even though open nesting yields high concurrency, it has inherent overheads such as commit overheads or abstract locking overheads [78]. Thus, open-nesting does not always perform better than closed-nesting.

Motivated by this, we propose a scheduling-based parallel nested (SPN) transactional memory model focusing on how to identify whether or not inner transactions should be executed in parallel and how to enhance the performance of parallel inner transactions in DTM.

8.2 Scheduler Design

SPN consists of two steps: 1) converting the sequence of inner transactions to parallel inner transactions and running them simultaneously, 2) maintaining a "transaction table" of on-going parallel inner transactions. In the first step, nodes invoking transactions execute all inner transactions simultaneously and request their objects from object owners simultaneously. Each object request is composed by four elements – the order number of the inner transaction ($NiTx$), object id (oID), type of the transaction ($Type$), and outer transaction id ($TxID$). An order number is assigned from 1 to the total number of parallel inner transactions of the same parent transaction. Different inner transactions may request the same object. Thus, in the second step, the owners moves these elements (i.e., $NiTx$, oID, $Type$) to the transaction table and identifies which inner transactions can be executed in parallel.

The transaction table is updated when requesting and validating objects. At both times, object owners maintain the transaction table after storing the elements as follows.

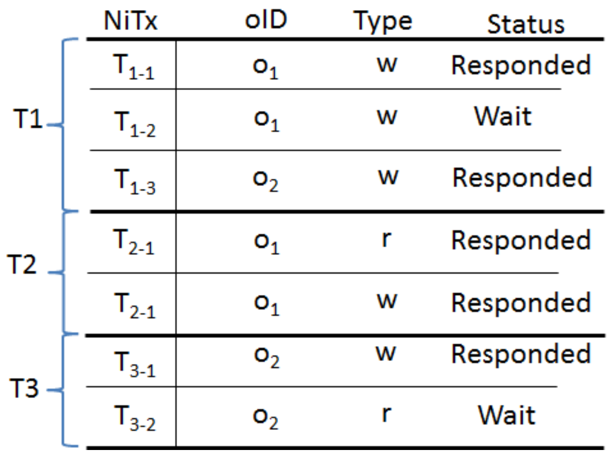
object owners maintains

- Requesting: If $NiTx$ is 1, sends an object corresponding to oID to the requester according to TFA rules and update its status as $Responded$. If $NiTx$ is not 1, the object owner checks whether the prior $NiTx$s of current requesters have requested the same oID. The prior $NiTx$s indicates lower numbers than $NiTx$ with the same $TxID$. If $NiTx$ is not 1 and no prior $NiTx$s have requested the same oID, the owner sends the object to the requester because of no conflict. If any prior $NiTx$s have requested the same oID, update its status as $Wait$ and send a backoff time to the requester.
- Validating: When one requesting transaction validates before others, allow the requested validation and remove corresponding $TxID$s from the transactional table. Other transactions that requested the same objects are aborted. Without requesting the object again, the aborted transactions will receive the updated objects.

A requester may receive multiple backoff times from an owner. Receiving a backoff time means that an inner transaction is using the same object. Different backoff times are assigned to different inner transactions accessing the same object. Thus, we represent how the owner decides a backoff time and the requester maintains the backoff time. When parallel inner transactions with different $NiTx$s request objects, backoff times are calculated using a number of $Wait$ statuses. Even if a prior status is not updated (or is delayed) for some reason, an owner checks whether to reply with an object or a backoff time using existing statuses. If a conflict is detected after updating prior statuses, conflicting on-going inner transactions receive backoff times and abort. In order to compute a backoff time, we use the number of $Wait$ statuses in the transaction table. If the $NiTx$'s status is $Wait$, its backoff time will be the execution time $\times$ the number of $Wait$ statuses. When the inner transaction commit internally, another inner transaction with the smallest backoff time is woken up and starts using the updated object.

Transactions' backoff time are stored in an hash table. The backoff time of a transaction corresponds to the average execution time of that transaction. The key of the table is the name of the transaction. If a *new order* transaction in TPC-C requests an object for example, its owner creates a bucket with key "new_order". When the transaction commits, the execution time is computed as its commit time - its staring time. Later, if SPN detects a conflict with another new order transaction, the execution time is assigned to the new order as a backoff time. As soon as an object is updated, a transaction receiving a backoff time is woken up to access the object. Thus, SPN does not need an exact backoff time, so SPN uses an approximated execution time.

SPN also identifies conflicts between write and read transactions. If write and write or write and read transactions access an object, the first write transactions' status will be $Responded$ and the second write or read transactions' status will be $Wait$. Read and write or read and read transactions accessing an object simultaneously receive the object.



	NiTx	oID	Type	Status
T1	$T_{1\text{-}1}$	o_1	w	Responded
	$T_{1\text{-}2}$	o_1	w	Wait
	$T_{1\text{-}3}$	o_2	w	Responded
T2	$T_{2\text{-}1}$	o_1	r	Responded
	$T_{2\text{-}1}$	o_1	w	Responded
T3	$T_{3\text{-}1}$	o_2	w	Responded
	$T_{3\text{-}2}$	o_2	r	Wait

Figure 8.2: An example for maintaining a transactional table

Figure 8.2 illustrates an example for a transaction table containing three outer transactions. T_1 contains three write inner transactions. T_2's inner transactions access object o_1, and T_3's inner transactions access object o_2. If the owner receives the requests from the node invoking T_{1-1}, T_{1-2}, and T_{1-3}, o_1, a backoff time, and o_2 are sent to T_{1-1}, T_{1-2}, and T_{1-3}, respectively. T_{1-2}

waits for the backoff time. As soon as T_{1-1} commits, T_{1-2} that waits for o_1 is woken up and use o_1 updated by T_{1-1}. T_{2-1} and T_{2-2} receive o_2 because of no conflict. If T_1 commits first, T_2 and T_3 will be aborted. If T_2 commits first, T_1 will be aborted.

The purpose of SPN is to maximize the parallelism inner transactions in DTM, executing them in parallel. SPN keeps track of inner transactions' access pattern and use it for resolving conflicts. Non conflicting inner transactions are executed in parallel. Conversely, conflicting inner transactions accessing the same objects are serialized.

Evidently, parallel nesting outperforms non-parallel nesting in throughput. We are interested in how much SPN outperforms CLOSED (non-parallel nesting). In order to analyze this, let the number of aborts of T_i be denoted as λ_i, and the local execution time of T_i containing N inner transactions be denoted as γ_N. According to Lemma 6.5.1, the total number of aborts on N transactions is $\sum_{i=1}^{N} \lambda_i \leq N - 1$. We have the following theorem.

Theorem 8.2.1. *Given N inner transactions and M objects, the RCR of SPN and $CLOSED$ (i.e., non-parallel closed nesting) on NR is less than 1, where $N \geq 2$ and $M \geq 2$.*

Proof. $makespan^{N}_{CLOSED}(NR)$ = $(N-1)(2\sum_{i=1}^{M} d(n_i, n_j)+\gamma_N)$, because each request of M objects from object owners is serialized. $makespan^{N}_{SPN}(NR)$ = $(N-1)(2\max(\forall_{i=1}^{M} d(n_i, n_j))+\gamma_N)$, because N inner transactions request M objects from object owners simultaneously. $\max(\forall_{i=1}^{M} d(n_i, n_j)) < \sum_{i=1}^{M} d(n_i, n_j)$, so the RCR of SPN and $CLOSED < 1$. The theorem follows. $\square$

Theorem 8.2.1 shows that SPN always outperforms CLOSED when $M \geq 2$. When $M = 1$, they produce the same performance.

8.3 Algorithm

Algorithm 8 consists two procedures, $Retrieve\_Request$ and $Validating$ procedures. whenever the owner receives a request, the $Retrieve\ \_Request$ procedure, that is invoked in an owner, returns one of $object$, $backoff$, and $abort$ to requester.

If a requesting object is locked due to validating, the procedure returns $abort$. Otherwise, the procedure checks conflicts among inner transactions using the transaction table. If all previous inner transactions accessing the object are read, the object is returned due to no conflict. If at least previous inner transaction is write, a backoff time is returned due to a conflict. The backoff time is computed using $exTime$ and a number of wait statuses. $exTime$ represents the execution time of an inner transaction. Whenever a transaction completes, we compute an average execution time and store it to $exTime$. If $Retrieve\_Request$ returns a backoff time to an inner transaction, it selects the corresponding $exTime$ of the inner transaction.

When an inner transaction tries to validate, the $Validating$ procedure is invoked. Even if the inner transaction commits, its modification does not affect other transactions, so object owners do

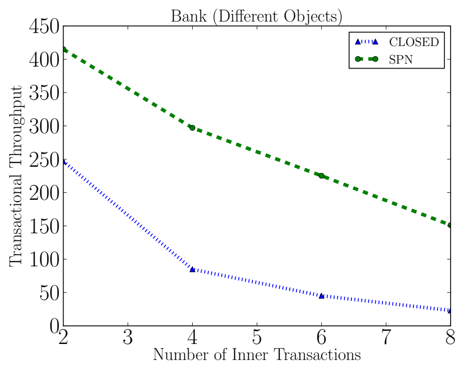
Algorithm 8: Algorithms for SPN

```
Procedure Retrieve_Request
Input: $TxID$, $NiTx$, $oID$, $type$
Output: $object$, $backoff$, $abort$
object = find_object($oID$);
if checkLock(object) then
    ▷ If a requesting object is locked due to validating
    return abort;
if $NiTx$ == 1 then
    transactionalTable.put($TxID$, $NiTx$, $type$, $oID$, "responded");
    ▷ transactionalTable is a hash table with key $TxID$
    return object;
else
    (TypeList, StatusList) = transactionalTable.getSameObj($TxID$, $oID$);
    ▷ The getSameObj procedure returns two lists of type and status of the inner transactions requesting $oID$ with
    $TxID$, respectively.
    ▷ Nstatus indicates a number of "wait" statuses.
    if TypeList is all reads then
        transactionalTable.put($TxID$, $NiTx$, $type$, $oID$, "responded"); return object;
    else
        transactionalTable.put($TxID$, $NiTx$, $type$, $oID$, "wait");
        Nstatus = StatusList.getPrior($Wait$);
        ▷ Nstatus indicates a number of "Wait" statuses.
        backoff = exTime $\times$ (Nstatus + 1); return backoff;
Procedure validation
Input: $oID$
Output: $commit$, $abort$
if validate($oID$) then
    ▷ validate checks whether the object of $oID$ is updated intermally.
    return abort;
List Txs = get_transactions($oID$);
▷ get_transactions returns a list of transactions waiting for $oID$
if Txs == null then
    return commit;
String aTx = get_minBackoff(Txs);
▷ get_minBackoff returns a transaction's id with the smallest backoff time
wakeup(aTx);
return commit;
```

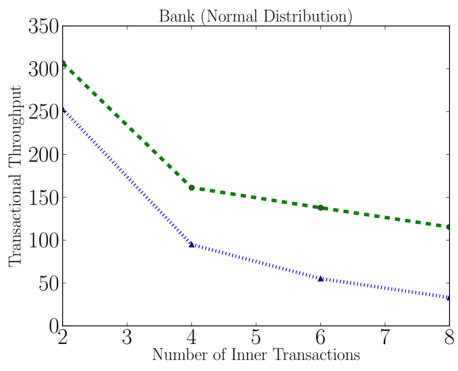
not know its modification due to the nature of closed nesting. If any objects to be validated are not modified, the procedure checks for which transactions are waiting for the objects and select a transaction with the shortest bakcoff time. If existing, the transaction is woken up and fetches an object from its local cache.

8.4 Evaluation

We implemented SPN in the HyFlow DTM framework [79]. We cannot compare our results with any competitor, as none of the DTMs that we are aware of support closed nesting and scheduling. Thus, we compared SPN under nested TFA (N-TFA) with only N-TFA [78].

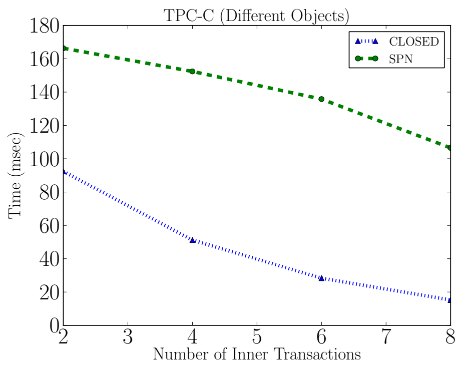


(a) Bank (Different Objects)

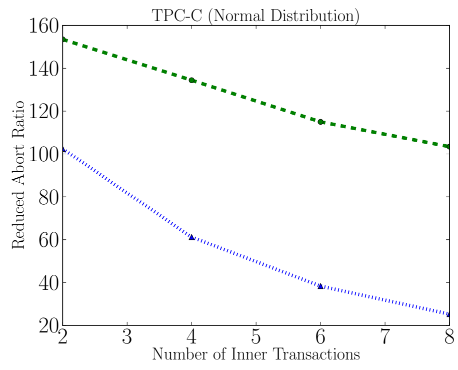


(b) Bank (Normal Distribution)

Figure 8.3: Throughput of Bank Benchmark



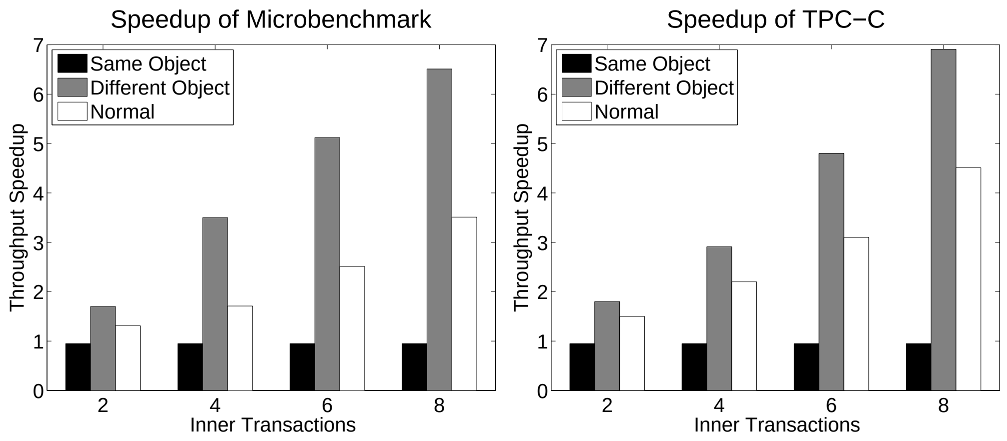
(a) TPC-C (Different Objects)



(b) TPC-C (Normal Distribution)

Figure 8.4: Throughput of TPC-C Benchmark

Figures 8.3 8.4 show the transactional throughput of bank and TPC-C benchmarks, respectively. In Figures 8.3(a) 8.4(a), all inner transactions access different objects, so all inner transactions are executed in parallel. In Figures 8.3(b) 8.4(b), the objects that inner transactions will access are randomly selected using a normal distribution. Its intent is to show the effectiveness of SPN under general workloads. SPN outperforms non-parallel closed nesting by up to 3.5 and 4.5$\times$ on micro-benchmark and TPC-C in general workloads, respectively.



(a) Speedup of Bank (b) Speedup of TPC-C

Figure 8.5: Throughput Speedup of Bank and TPC-C Benchmarks

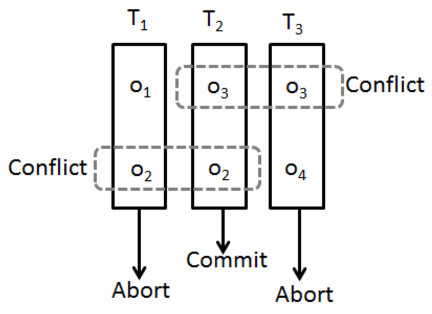
Chapter 9

The Cluster-based Transactional Scheduler

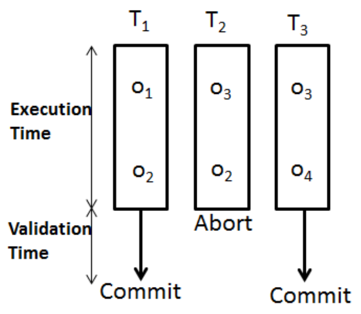
9.1 Motivation

Directory-based CC protocols (e.g., Arrow and Ballistic) [27, 21] in the single-copy model often keep track of the single writable copy. In practice, not all transactional requests are routed efficiently; possible locality is often overlooked, resulting in high communication delays. A distributed transaction consumes more execution time, which include the communication delays that are incurred in requesting and retrieving objects than a transaction on multiprocessors [34]. Thus, the probability for conflicts and aborts is higher. Even though a transaction in a full replication model does not request and retrieve objects, maintaining replicas of all objects at each node is costly. Increasing locality (and availability) by brute-force replication while ensuring one-copy serializability can lead to communication overhead. Motivated by this, we consider a k-cluster-based replication model for cc DTM. In this model, multiple copies of each object are distributed to k selected nodes to maximize locality and availability and to minimize communication overhead.

Moreover, a transaction may execute multiple operations with multiple objects, increasing the possibility of conflicts. Figure 9.1 shows a scenario two conflicts occurring with three concurrent transactions, T_1, T_2, and T_3 using two objects. Under TFA, a conflict over o_2 between T_1 and T_2 occurs and another conflict over o_3 between T_2 and T_3 occurs. If T_2 commits first, T_1 and T_3 will abort because T_2 will update o_3 and o_2 even though T_1 and T_3 do not contend. If T_2 aborts as shown in Figure 9.1(b), T_1 and T_3 will commit. Motivated by this, CTS aborts T_2 in advance and allows T_1 and T_2 to commit concurrently. A contention manager resolves a conflict between two transactions, but CTS avoids two conflicts among three transactions and guarantees the concurrency of two transactions of them.



(a) TFA



(b) CTS with TFA

Figure 9.1: Executing T_1, T_2, and T_3 Concurrently

9.2 Scheduler Design

In the case of an off-line scheduling algorithm (all concurrent transactions are known), a simple approach to minimize conflicts is to check the *conflict graph* of transactions and determine a *maximum independent set* of the graph, which is *NP-complete*. However, as an on-line scheduling algorithm, CTS checks for conflicts between a transaction and other ongoing transactions accessing an object whenever the transaction requests the object.

Let node n_x belong to cluster z. When transaction T_x at node n_x needs object o_y for an operation, it sends a request to the object owner of cluster z. When another transaction may have requested o_y but no transaction has validated o_y, there are two possible cases. The first case is when the operation is read. In this case, o_y is sent to n_x without enqueuing, because the read transaction does not modify o_y. In the second case, when the operation is write, CTS determines whether o_y is sent to the requester (i.e., n_x) or not by considering previously enqueued transactions and objects. Once CTS allows T_x to access o_y, CTS moves x and y representing T_x and o_y respectively to two scheduling queues. The object owners for each cluster maintain the following two queues, $\mathbb{O}$ and $\mathbb{T}$. Let $\mathbb{O}$ denote the set of enqueued objects and $\mathbb{T}$ denote the set of transactions enqueued by the object owners. If the object owner of cluster z enqueues x and y, it updates its scheduling queues to the other object owners'.

If $x \in \mathbb{T}$ and $y \notin \mathbb{O}$, x and y are enqueued and o_y is sent to n_x. This case indicates that T_x has requested another object from the object owner and o_y has not been requested yet. However, if $x \notin \mathbb{T}$ and $y \in \mathbb{O}$, CTS has to check for whether $\mathbb{T} \mid \beta$ includes more than two transactions or not, where $\beta = \mathbb{O} \mid \alpha$ and $\alpha = \mathbb{T} \mid y$. $\mathbb{O} \mid \alpha$ indicates objects requested by T_α and $\mathbb{T} \mid y$ represents transactions requesting o_y. This case shows when o_y is being used by other transactions and the transactions share an object with another transaction. CTS does not consider a conflict between two transactions because a contention manager aborts one of them when they validate. Thus, the transactions involved in $\mathbb{T} \mid y \cap \mathbb{T} \mid \beta$ abort, x and y are enqueued, and o_y is sent to n_x. The aborted transactions are dequeued.

If $x \in \mathbb{T}$ and $y \in \mathbb{O}$, CTS has to check for whether $\mathbb{T} \mid \gamma$ is distinct from $\mathbb{T} \mid y$ or not, where $\gamma = \mathbb{O} \mid x$. This case means that T_x has requested an object requested by another transaction and also o_y has been requested by another transaction. If two different transactions are using different objects that T_x has requested and is requesting, respectively, CTS aborts T_x to protect two transactions from aborting. Thus, if $\mathbb{T} \mid \gamma$ is distinct from $\mathbb{T} \mid y$, x and y also are enqueued and o_y is sent to n_x. Otherwise, o_y will not be sent to n_x, aborting T_x. In this case, the object owner knows that T_x aborts. Thus, the objects that T_x has requested will be sent to n_x after the objects are updated.

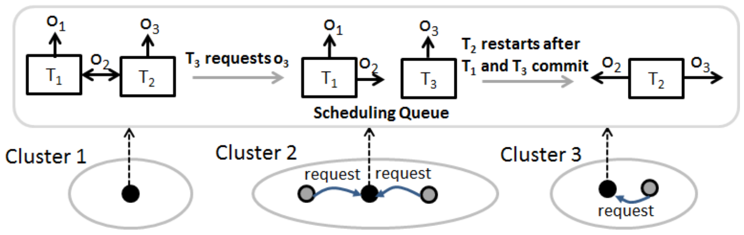


Figure 9.2: An Example of CTS

Figure 9.2 illustrates an example of CTS after applying the 3-clustering algorithm on a six-node network. The black circles represent object owners. The scheduling queue includes live transactions T_1 and T_2, and each transaction indicates its objects in use. If T_3 requests o_3, CTS checks for conflicts between T_3 and the enqueued transactions (i.e., T_1 and T_2). CTS aborts T_2 because of two conflicts among T_1, T_2 and T_3. T_2 restarts after T_1 and T_3 commit. The committed transactions are dequeued, and T_2 is enqueued.

We consider two effects of CTS on clusters. First, when a transaction requests an object, CTS checks for conflicts between the transaction and the previous requesting transactions and aborts some transactions in advance to prevent other transactions from aborting. This results in a reduced number of aborts. Second, in TFA, if a transaction aborts, the transaction will restart and request an object again, incurring communication delays. However, in CTS, object owners hold aborted transactions. When validation of an object completes, the object is sent to the nodes invoking the aborted transactions. Thus, CTS lets the aborted transactions use newly updated objects without requesting the object again, reducing communication delays.

9.3 Algorithms

We now present the algorithms for CTS. There are four algorithms: Algorithm 9 for $Open\_Object$, Algorithm 10 for $Retrieve\_Request$, CTS and $Retrieve\_Response$.

$Open\_Object$ described in Algorithm 9 is invoked when a transaction needs an object. After finding the owner of oid in a requester's cluster, the requester sends $type$, oid, and $txid$ to the owner. $type$ represents a read or write transaction. If the received object is null and the backoff time assigned by the owner is not 0, the requester saves the backoff time, so the transaction corresponding to $txid$ aborts and waits for the backoff time. If it expires, $Open\_Object$ is invoked again. Otherwise, the requester wakes up and receives the object. Even if $Open\_Object$ successfully receives an object, another transaction requesting oid may validate the object first. Thus, the status of $txid$ is checked before returning the object. If the status is abort, $Open\_Object$ returns $null$.

Algorithm 9: Algorithm of Open_Object

```
Procedure Open_Object
Input: Transaction_Type type, Transaction_id txid, Object_id oid
Output: $null$, $object$
if there is object corresponding to oid in the local cache then
    return $object$;
$owner$ = find_Owner($oid$); ▷ $owner$ is an address of $oid$'s owner
Send $type$, $oid$, and $txid$ to $owner$;
Wait until that Retrieve_Response is invoked;
Read $object$ and $backoff$ from Retrieve_Response;
if $object$ is null and $backoff$ is not 0 then
    Set $backoff$; return $null$;
else
    if txid is already aborted then
        return $null$;
    else
        return $object$;
```

Algorithm 10 describes $Retrieve\_Request$, which is invoked when an object owner receives a request and $Retrieve\_Response$, which is invoked when the object that $Object\_Open$ has requested is received. If the requested object is being validated or the CTS procedure returns $Abort$, a backoff time is assigned. To compute a backoff time, we use a transaction *stats table* that stores the average historical commit time of a transaction. Each table entry holds the most current successful commit times of write transactions. Unless CTS returns $Abort$, $Retrieve\_Request$ enqueues $txid$ and oid to $requestTable$ and $objectTable$, and sends the object to the requester. As soon as the object is updated, it is sent to a transaction dequeued from $abortTable$.

The CTS procedure determines whether a transactions aborts or not by using two hash tables: $requestTable$ and $objectTable$ accessed with the keys of $txid$ and oid, respectively. If CTS returns $Abort$, $txid$ and oid will be moved to $abortTable$ in order to maintain aborted transactions. Otherwise, these are moved to $requestTable$ and $objectTable$.

$Retrieve\_Response$ is invoked if another transaction starts or ends the validation of $object$. When the transaction starts validation, its $txid$ is dequeued from $requestTable$ and $objectTable$, and an abort message with a backoff time is sent to $Retrieve\_Response$. When the transaction ends validation, $Retrieve\_Response$ receives only $object$. If the transaction restarts due to expired backoff timer, $Open\_Object$ finds the object owner of oid again.

Algorithm 10: Algorithms of Retrieve_Request, CTS, and Retrieve_Response

```
Procedure Retrieve_Request
Input: $type$, $oid$, and $txid$
$object$ = get_Object($oid$); $address$ = get_Requester_Address();
Integer $backoff$ = 0;
if $object$ is not null then
    if $object$ is being validated or CTS($txid$, $oid$) == Abort then
        $backoff$ = computeBackoff(); $object$=null;
        abortTable.put($txid$, $oid$);
    else
        objectTable.put($oid$, $txid$); requestTable.put($txid$, $oid$);
Send $object$ and $backoff$ to $address$;
Procedure CTS
Input: $txid$ and $oid$
Output: $Abort$ or $notAbort$
$object$ = requestTable.get($txid$);
▷ $object$ holds objects that $txid$ has requested
$txn$ = objectTable.get($oid$);
▷ $txn$ holds transactions that have been requested $oid$
if $object$ == null or $txn$ == null then
    if $object \neq$ null then
        $tx$ = objectTable.get($object$);
        if objectTable.get(requestTable.get($tx$)).length $\geq$ 2 then
            $object$ = $object$ + requestTable.get($tx$);
            requestTable.remove($tx$); abortTable.put($tx$,$object$);
    return notAbort; ▷ $txid$ or $oid$ has not been enqueued.
else if $txn$ == objectTable.get($object$) then
    return Abort;
return notAbort;
Procedure Retreive_Response
Input: $object$, $txid$, $backoff$
if $txid$ is waiting in $Object\_Open$ then
    Send a signal to wake up and give $object$ and $backoff$;
else if $txid$ is ready to restart then
    Send a signal to restart and give $object$;
else if $txid$ is working then
    Set $backoff$ and abort $txid$;
```

Whenever an object is requested, CTS is invoked. The time complexity is $O(1)$ to lookup a transaction and an object. However, $objectTable$ may return multiple transactions. To check all enqueued transactions, the time complexity for an object is $O(the\ number\ of\ enqueued\ transactions)$.

9.4 Analysis

For the analysis of CTS, we use the method of the analysis described in Chapter 6.

Lemma 9.4.1. *Given scheduler B and N transactions, $makespan_B^N(NR) \leq 2(N-1)\sum_{i=1}^{N-1} d(n_i, n_j) + \Gamma_N$.*

Proof. Lemma 6.5.1 gives the total number of aborts on N transactions under scheduler B. If a transaction T_i requests an object, the communication delay will be $2 \times d(n_i, n_j)$ for both requesting and object retrieving times. Once T_i aborts, this delay is incurred again. To complete N transactions using scheduler B, the total communication delay will be $2(N-1)\sum_{i=1}^{N-1} d(n_i, n_j)$. The theorem follows. □

Lemma 9.4.2. *Given scheduler B, N transactions, k replications, $makespan_B^N(PR) \leq (N - k)\sum_{i=1}^{N-k} d(n_i, n_j) + (N-k+1)\sum_{i=1}^{N-1}\sum_{j=1}^{k-1} d(n_i, n_j) + \Gamma_N$.*

Proof. In PR, k transactions do not need to remotely request an object, because k nodes hold replicated objects. Thus, $\sum_{i=1}^{N-k} d(n_i, n_j)$ is the requesting time of N transactions and $\sum_{i=1}^{N-1}\sum_{j=1}^{k-1} d(n_i, n_j)$ is the validation time based on atomic multicasting for only k nodes of each cluster. The theorem follows. □

Lemma 9.4.3. *Given scheduler B and N transactions, $makespan_B^N(FR) \leq \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} d(n_i, n_j) + \Gamma_N$.*

Proof. Transactions request objects from their own nodes, so their requesting times do not occur in FR, even when the transactions abort. The basic idea of transactional schedulers is to minimize conflicts through enqueueing transactions when the transactions request objects. Thus, the transactional schedulers (i.e, B and CTS) do not affect $makespan_{x \in \{B, CTS\}}^N(FR)$. Thus, when a transaction commits, FR takes $\sum_{i=1}^{N-1}\sum_{j=1}^{N-1} d(n_i, n_j)$ for only atomic broadcasting to support one-copy serializability. □

Theorem 9.4.4. *Given scheduler B and N transactions, $makespan_B^N(FR) \leq makespan_B^N(PR) \leq makespan_B^N\ (NR)$.*

Proof. Given k PR, $\lim_{k\to1} makespan_B^N(PR) \leq 2(N-1)\sum_{i=1}^{N-1} d(n_i, n_j) + \Gamma_N$, and $\lim_{k\to N} makespan_B^N(PR) \leq \sum_{i=1}^{N-1}\sum_{j=1}^{N-1} d(n_i, n_j) + \Gamma_N$. The theorem follows. □

Theorem 9.4.5. *Given N transactions and M objects, the RCR of schedulers CTS on PR and scheduler B on FR is less than 1, where $N > 3$.*

Proof. Let $\sum_{i=1}^{N-1} d(n_i, n_j)$ denote δ_{N-1}. To show that the RCR of CTS on PR and B on FR is less than 1, $makespan_{CTS}^{N}(PR) < makespan_{B}^{N}(FR)$. CTS detects potential conflicts and aborts a transaction incurring the conflicts. The aborted transaction does not request objects again. Thus we derive $makespan_{CTS}^{N}(PR) \leq 2M\delta_{N-k} + M\sum_{i=1}^{N-1}\delta_{k-1} + M\Gamma_N$. $2\delta_{N-k} + (N-1)\delta_{k-1} \leq (N-1)\delta_{N-1}$, so that $2\delta_{N-k} \leq (N-1)\delta_{N-k}$. Only when $N \geq 3$, PR is feasible. Hence, $makespan_{CTS}^{N}(PR) < makespan_{B}^{N}(FR)$, where $N > 3$. The theorem follows. □

Theorem 9.4.5 shows that CTS in PR performs better than FR. Even though PR incurs requesting and object retrieving times for transactions, CTS minimizes these times, resulting in less overall time than the broadcasting time of FR.

9.5 Evaluation

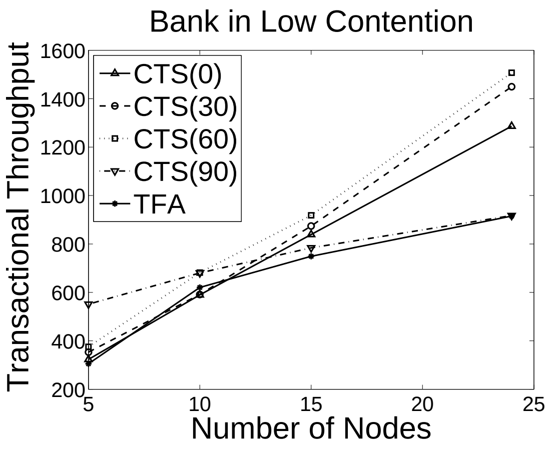
9.5.1 Experimental Setup

We implemented CTS in the HyFlow DTM framework [23], and developed six benchmarks for experimental studies. The benchmarks include two monetary applications (Bank and Loan) [23], distributed versions of the Vacation of the STAMP benchmark suite [69], and three distributed data structures including Counter, Red/Black Tree (RB-Tree) [38], and Distributed Hash Table (DHT).

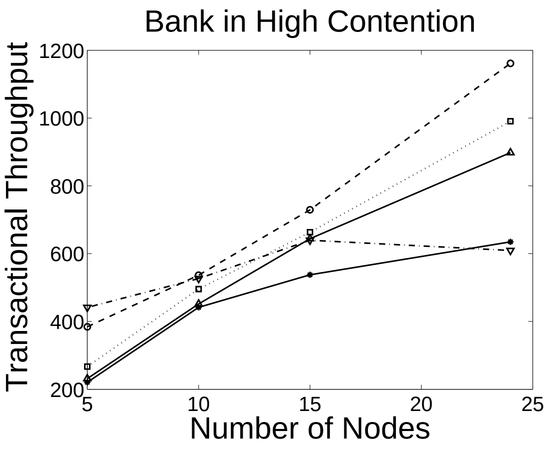
To select k nodes for distributing replicas of each object, we group nodes into clusters, such that nodes in a cluster are closer to each other, while those between clusters are far apart. Recall that the distance between a pair of nodes in a metric-space network determines the communication cost of sending a message between them. We use a k clustering algorithm based on METIS [88], to generate k clusters with small intra-cluster distances i.e., k nodes may hold the same objects. Our partial replication relies on the usage of a *total order multicast* (TOM) primitive to ensure agreement on correctness in a genuine multicast protocol [56]. The object owners for each cluster update objects through a TOM-based protocol.

We use *low* and *high contention*, which are defined as 90% and 10% read transactions of one million active concurrent transactions per node, respectively [38]. A read transaction includes only read operations, and a write transaction consists of only write operations [38]. Our experiments were conducted on 24-node testbed. Each node is an AMD Opteron processor clocked at 1.9GHz. We use Ubuntu Linux 10.04 server OS and a network with a private gigabit ethernet. Each experiment is the average of ten repetitions. The number of objects for a transaction is selected randomly from 2 to 20. We considered CTS(30) and CTS(60), meaning CTS over 30% and 60% object owners of the total nodes, respectively. For instance, CTS(30) under 10 nodes means CTS over 3-clustering algorithm. We measured the *transactional throughput* (number of committed transactions per second) under increasing number of requesting nodes and failed nodes.

9.5.2 Evaluation



(a) Bank (Low)



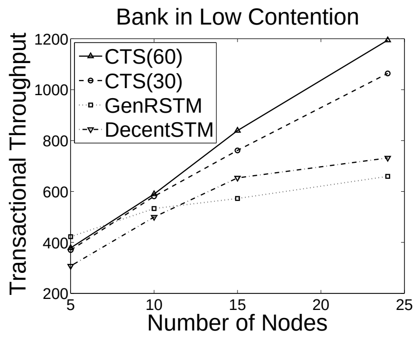
(b) Bank (High)

Figure 9.3: Throughput of Bank Benchmark with No Node Failures.

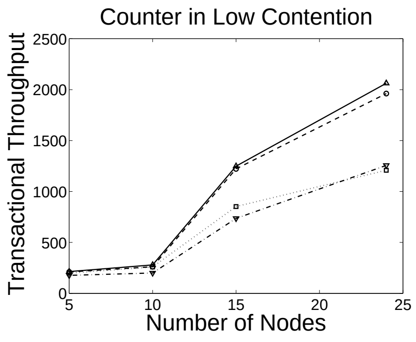
Figure 9.3 intends to show two effects of scheduling by CTS and the improvement of object availability by increasing object locality. To show the effectiveness of CTS, TFA is compared with CTS(0) – the combination of CTS and TFA with no replication. CTS(0) improves throughput over TFA as much as 1.5× under high contention because the number of conflicts decreases. CTS(0) outperforms CTS(90) in throughput, but it is non-fault-tolerant. The throughput produced by CTS(90) is degraded due to the large number of broadcasting messages needed to update all replicas. Due to high object availability on CTS(90), the requesting times of aborted transactions are less reduced. Meanwhile, due to low object availability on CTS(0), the requesting times are more reduced but object retrieving times increase. Thus, CTS(30) and CTS(60) achieve decreased object requesting and retrieving times, resulting in a better throughput than CTS(0) and CTS(90).

We considered two competitor DTM implementations: GenRSTM [51] and DecentSTM [52]. GenRSTM is a generic framework for replicated STMs and uses broadcasting to achieve transactional properties. DecentSTM implements a fully decentralized snapshot algorithm, minimizing aborts. We compared CTS with GenRSTM and DecentSTM.

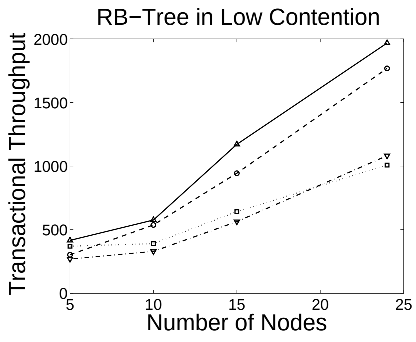
Figure 9.4 shows the throughput of three benchmarks for CTS(30), CTS(60), GenRSTM, and DecentSTM with 20% node failure under low and high contention, respectively. In these experiments, 20% of nodes randomly fail. GenRSTM broadcasts updates to all other replicas, which incurs a overhead. DecentSTM is based on a snapshot isolation algorithm, which requires searching the history of objects to find a valid snapshot. This algorithm also incurs a significant overhead. Due to those overheads, their performance degrades for more than 24 requesting nodes. Thus we observe



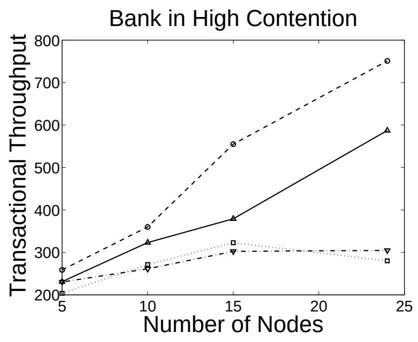
(a) Bank (Low)



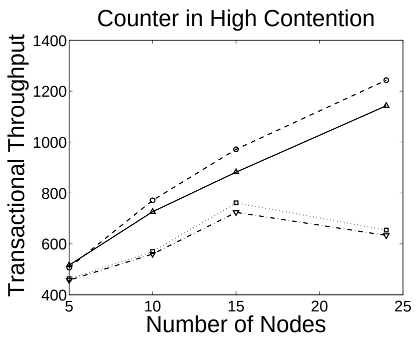
(b) Counter (Low)



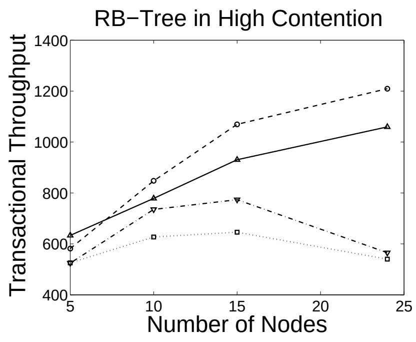
(c) RB Tree (Low)



(d) Bank (High)



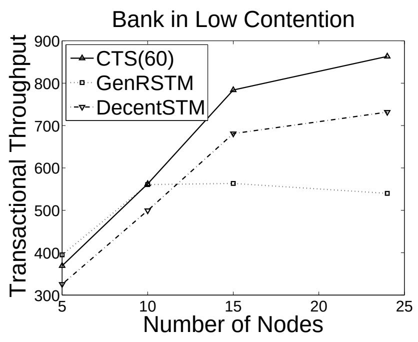
(e) Counter (High)



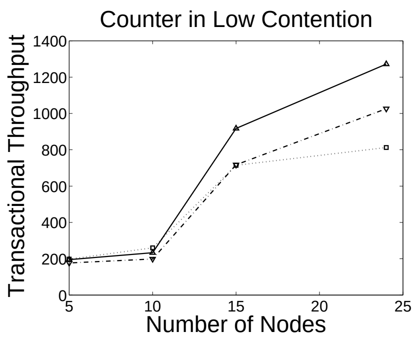
(f) RB Tree (High)

Figure 9.4: Throughput of 3 Benchmarks with 20% Node Failure under Low and High Contention (5 to 24 nodes).

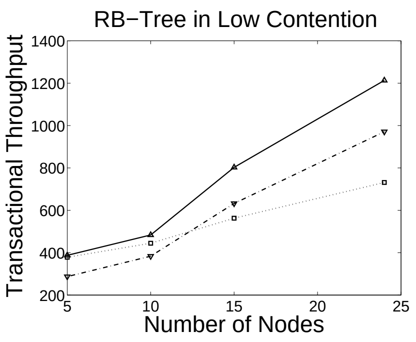
that CTS yields higher throughput than GenRSTM and DecentSTM. In particular, 60% of nodes are entitled to the ownership of an object based on CTS(60). CTS(60) maintains smaller clusters than CTS(30), so the communication delays to request and retrieve objects decrease, but the number of messages increases. Under high contention, CTS avoids the large number of conflicts, so CTS yields much higher throughput than GenRSTM and DecentSTM.



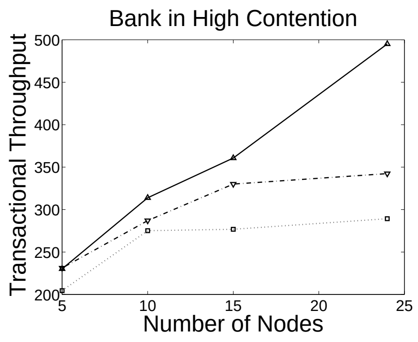
(a) Bank (Low)



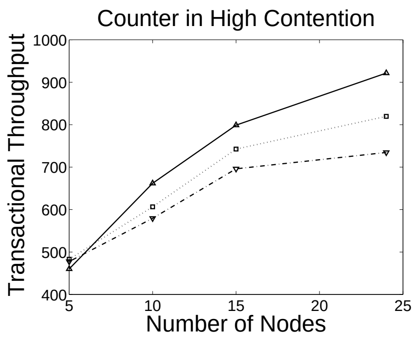
(b) Counter (Low)



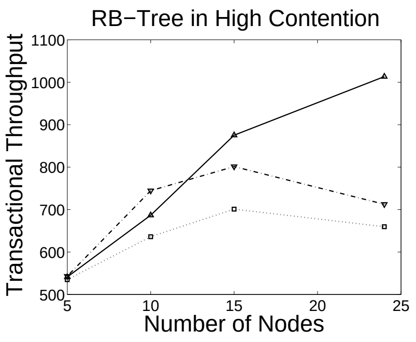
(c) RB Tree (Low)



(d) Bank (High)



(e) Counter (High)



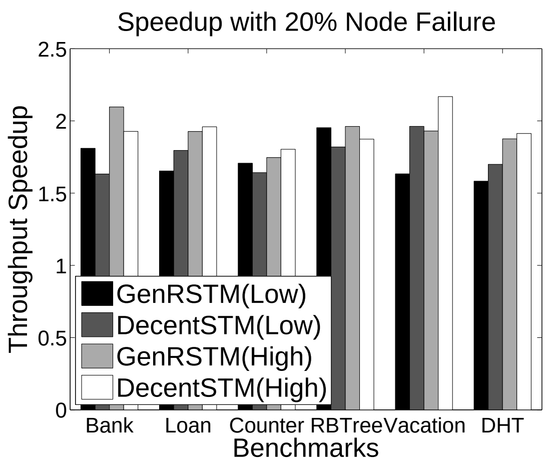
(f) RB Tree (High)

Figure 9.5: Throughput of 3 Benchmarks with 50% Node Failure under Low and High Contention (5 to 24 nodes).

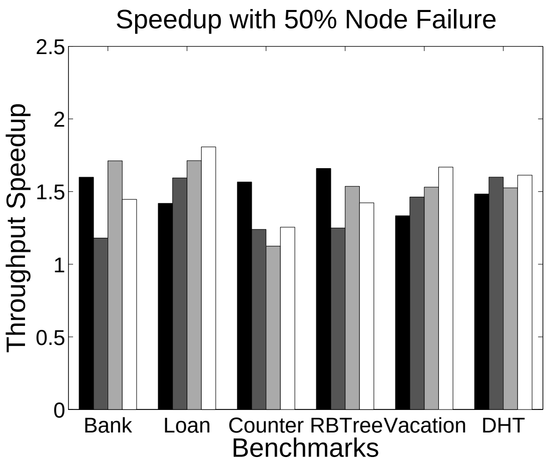
Figure 9.5 shows the throughput of three benchmarks for CTS(60), GenRSTM, and DecentSTM with 50% node failure under low and high contention, respectively. GenRSTM's and DecentSTM's throughput do not degrade as the number of failed nodes increases, because every node holds replicated objects. However, in CTS, this causes communication delays to increase, degrading throughput, because object owners may fail or scheduling lists may be lost. Over less than ten nodes with 50% failed nodes, GenRSTM yields higher throughput than CTS, because the number of messages decreases. As the number of nodes increases, CTS outperforms GenRSTM and DecentSTM in throughput.

We computed the throughput speedup of CTS(60) over GenR STM and DecentSTM i.e., the ratio of CTS's throughput to the throughput of the respective competitor.

Figure 9.6 summarizes the throughput speedup under 20% and 50% node failure. Our evaluations



(a) 20% Node Failure



(b) 50% Node Failure

Figure 9.6: Summary of Throughput Speedup

reveal that CTS(60) improves throughput over GenRSTM by as much as 1.9533 (95%) $\sim$ 2.0968 (109%) $\times$ speedup in low and high contention, respectively, and over DecentSTM by as much as 1.9622 (96%) $\sim$ 2.1683 (116%) $\times$ speedup in low and high contention, respectively. In other words, CTS improves throughput over two existing replicated DTM solutions (GenRSTM and DecentSTM) by as much as (average) 1.55$\times$ and 1.73$\times$ under low and high contention, respectively.

Chapter 10

The Locality-aware Transactional Scheduler

10.1 Motivation

The business logic of a transactional application determines whether its object accesses exhibit locality or not. In this dissertation, we focus on *time locality*, in which the probability that a transaction accesses an object increases when that object has been previously accessed. Any optimization for increasing the degree of locality can cause intrusive coupling between the application and the distributed concurrency control mechanism. A promising solution is therefore to design protocols with the capability for exploiting locality on application data accesses – e.g., moving objects closer to the node where transactional requests originate.

Figure 10.1 compares the performance of two versions of the TPC-C [83] benchmark. One is a local version (called LOCAL TPC-C), in which data access patterns are very skewed: each node selects the object to access from among a restricted subset of objects. Another is a non-local version (called NO-LOCAL TPC-C), in which each object accessed is randomly selected. Here, we use TFA [89] as the underlying DTM protocol, which is based on the data-flow model. TFA is purely distributed (i.e., not replicated), and object ownership is always changed in favor of the committing transaction.

The experiment is conducted on a system with up to 8 nodes, each of which has 8 application threads running TPC-C. The plot clearly reveals the speed-up under locality settings. After an initial phase in which the locality is not defined, the data-flow approach (i.e., LOCAL TPC-C) migrates the objects accessed to the requesting nodes, minimizing network interactions for the subsequent transactions. The scalability of NO-LOCAL TPC-C is limited by the expensive com-

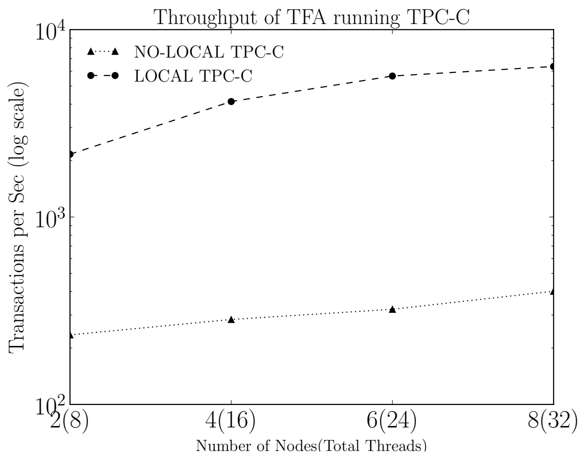


Figure 10.1: TPC-C and LOCAL TPC-C using TFA

munication costs due to the continuous moving of objects from node to node.

It is important to note that in any control-flow approach, similar performance can be achieved by only manually deploying objects using a-priori knowledge of application data access patterns. However, this knowledge becomes meaningless when application threads create new objects in the system. LOCAL-FLOW does not assume knowledge on application accesses and it does not need additional intrusive layers for managing objects placement. Our approach improves the basic data-flow model, making it appropriate for exploiting application locality.

10.2 Scheduler Design

In order to ensure the consistency of objects, we propose LOCAL-FLOW, *genuine* [55, 56] partial replication: only replicas involved during transactional execution participate in the commit phase. The protocol ensures *1-copy-serializability* [57] by acquiring locks on updated objects at commit time (using two-phase commit) and validating the accessed objects after lock acquisition.

LOCAL-FLOW uses a locality-aware transactional scheduler, called LTS, which is responsible for managing concurrent object requests from processing nodes. LTS monitors the key performance parameters needed for detecting the effectiveness of the current object placement and triggers the transition of ownership to exploit locality. In this section, we describe how LOCAL-FLOW and LTS work.

10.2.1 LOCAL-FLOW

LOCAL-FLOW is a partial replication protocol that ensures high locality and concurrency on distributed transactional processing. The protocol is suited for locality-aware applications deployed in scenarios with dynamic workloads (the initialization of a distributed system can be considered "dynamic" because there is no a-priori knowledge of transactional behavior).

When transactions mostly access remote objects instead of local objects, the impact of communication costs on total transaction execution time can be significantly high, resulting in poor performance. LOCAL-FLOW addresses this problem of detecting the best location for misplaced objects and moving them closer to their current *requesting nodes*. The requesting node n_r for an object o_i is a node that, according to the current transactions' data access pattern, is mostly performing operations on o_i. Transferring o_i to n_r means avoiding remote communications for those transactions executing on n_r that need o_i. In case o_i is needed by multiple requesting nodes, o_i is replicated and each requesting node is sent a replica (the object replication degree can be tuned based on the average number of requesting nodes per object). A local view of current object locations, called *CurrMap*, is stored at each node. Whenever an object transfer happens updates on CurrMap are propagated to all nodes. Conversely, when the workload is stable (i.e., most requests are local), CurrMap is occasionally updated and is locally queried for locating objects, minimizing the distributed interactions.

Transactions record their read/write accesses in private, transaction-local data structures (implemented as hash-maps), called *read-set* and *write-set*, respectively. Each object is marked with an integer value tracking its version, called *timestamp*. Different replicas of the same object have the same timestamp. The timestamp of an object is incremented whenever a transaction commits a modification on the object. The timestamp is used in the validation phase of a committing transaction: the timestamp of the committed version of the object is compared with the timestamp of the version stored in the transaction's read-set. This comparison reveals possible concurrent commits during the transaction's execution, which invalidate the object.

Consider a fully executed transaction Tx that enters its commit phase. Before proceeding further, it must ensure that all the objects in its read-set are still consistent and no other transactions are currently committing objects in its write-set. This is done in the following four steps:

(1) For each object o_{tx} in Tx's write-set, Tx contacts all of o_{tx}'s owners in order to acquire the locks corresponding to o_{tx}. When the request is received by o_{tx}'s owner it will try to acquire the lock. If the lock cannot be acquired then Tx's request is responded to with a negative acknowledge (NACK). Tx needs to collect all the replies from o_{tx}'s owners. If Tx receives all NACKs for o_{tx} then Tx is aborted and restarted (the case when Tx receives one positive acknowledgement (ACK) and other NACKs is described later in this subsection).

(2) After all object locks have been acquired, Tx validates the read-set using the object timestamps. This ensures that a transaction sees a consistent view of the accessed objects. Upon successful completion of this step, Tx can proceed to commit safely, otherwise an abort is issued.

(3) The timestamp of each written object is incremented. Subsequently, for local objects Tx's changes can be safely committed to shared memory, while for remote objects, the updated version is sent using a commit message.

(4) Local locks are instantly released and the remote objects are unlocked after receiving the commit message. This message, received by object owners, triggers the operation of LTS (see Sections 10.2.2 and 10.2.3).

Clearly, the first step can become a bottleneck in the presence of repeated adverse schedules, e.g. each transaction acquires a lock on a replica and finds the same object already locked on other replicas.

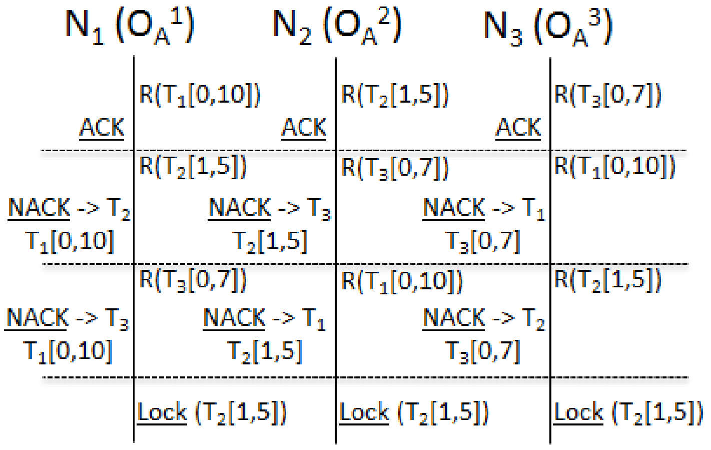


Figure 10.2: Example of commit phase

Figure 10.2 illustrates this situation. Three transactions T_1, T_2, and T_3 write a new version of an object o_A. o_A has three replicas o_A^1, o_A^2, and o_A^3, which are stored at nodes N_1, N_2, and N_3, respectively. Each transaction contacts N_1, N_2, and N_3 in order to acquire the lock. T_1 successfully locks o_A^1, T_2 locks o_A^2, and T_3 locks o_A^3. Each transaction considers o_A locked (or not) only after receiving all the replies (ACK or NACK) for its request. Clearly, in the described schedule, T_1, T_2, and T_3 receive one ACK from N_1, N_2, and N_3, respectively, and two NACKs each from the others. Here, in order to ensure correctness, T_1, T_2, and T_3 must be aborted and restarted. To overcome this, a deterministic rule is needed for the object owners to create a total order among the incoming requests, in order to deterministically select the transaction that can proceed with the locking process and abort the others. However, ensuring a total order among the requests implies paying the high cost of the atomic broadcast/multicast protocol.

LOCAL-FLOW minimizes situations in which all the concurrent transactions shown in the scenario in Figure 10.2 must be aborted. Each transaction embeds two pieces of information in its lock request messages: the number of retries done since its start (#retry) and the total number of locks needed in order to commit (#locks). Each object owner collects the pair <#retry,#locks> associated with the transaction receiving the ACK. This pair is then piggybacked to subsequent NACK messages that are issued to other transactions trying to acquire the same lock. A committing transaction needs to wait until replies from all object owners (storing object replicas) are received. In case it receives at least one ACK and other NACKs, the transaction compares its <#retry,#locks> pair with the other pairs extracted from the NACK messages (and associated with the transaction that successfully locked the object). The transaction with the maximum #retry is granted the lock. Ties on #retry are broken in favor of the transaction with the maximum #lock. Accordingly, other transactions are aborted and restarted, increasing their #retry. Thus, if a transaction receives a NACK, it is not doomed; only after the reception of all NACKs, the transaction aborts.

The rule just described is not deterministic. In adverse scenarios, all concurrent, validating transactions can have the same pairs <#retry,#locks>. In this case, each transaction independently aborts, because, by comparison, there is not only one transaction with the maximum #retry and #locks. The purpose of our approach is to prioritize concurrent committing transactions so as to reduce the probability for failure in case an abort to all transactions is triggered.

The current implementation batches the lock request messages per node. This minimizes the number of messages sent by the committing transaction to object owners.

10.2.2 Exploiting Locality

In order to exploit locality, we design LTS, a locality-aware transactional scheduler. The key goal of LTS is to help determine when objects are not correctly located in the system. LTS establishes a connection between objects and aborts caused by nodes that are running transactions accessing those objects.

We define a fixed time window, called *time-frame*, during which each node collects information on aborted transactions observed. Time-frame represents a local time interval. Specifically, each object owner (say own) records, for each object (say obj), two lists of pairs $<$ *node*,*abort_rate* $>$, where each pair represents the number of aborted transactions (*abort_rate*) in the last time-frame generated by the *node* accessing obj. The first list, called M, tracks the abort rate of obj's owners (including own). The second list, N, contains the abort rate of all the other nodes (non-owners). Accordingly, $N \cap M = \emptyset$.

Whenever a transaction commits a new version of an object o_A, its replicas are updated (see Section 10.2.1) and all transactions that concurrently requested o_A are aborted. Each object owner knows the transactions that are aborted due to contention on o_A. Therefore, when the commit message is received, the *abort_rate* of nodes running those aborted transactions are updated. If the time of receiving the commit message is within the current time-frame for updating the *abort_rate*, its value is simply incremented, otherwise it is overwritten.

The list M is maintained in the ascending order of *abort_rate*, whereas the list N is maintained in the descending order. When a new object version is committed and the lists are updated, LTS compares the *abort_rate* of M and N, node-by-node, starting from the first location of M, and generates the subset ($\acute{N}$) of nodes that have higher *abort_rate* than those in the same positions of

M (the size of $\acute{N}$ is less than the object replication degree). $\acute{N}$ represents candidate nodes for becoming new object owners. For each node $N_{ow} \in \acute{N}$, let ix represent its index in N. If the difference between N_{ow}'s *abort_rate* and the *abort_rate* of the node stored at position ix of M (named $N_M(ix)$) is higher than a threshold, then the ownership is changed from $N_M(ix)$ to N_{ow}. When the ownership is moved the object's value is transferred along with its lists N and M. The threshold represents the maximum number of aborted transactions that are allowed per time-frame without changing the ownership.

This mechanism captures workload changes. When the workload is stable, the number of aborted transactions started on non-object-owner nodes is negligible. Thus, there is no need to change ownership. The protocol manages this scenario using the aforementioned threshold. When the workload changes, a non-trivial number of transactions may request remote objects, resulting in several aborts. If the workload fluctuation is not temporary, this number will eventually exceed the threshold, triggering a change of ownership.

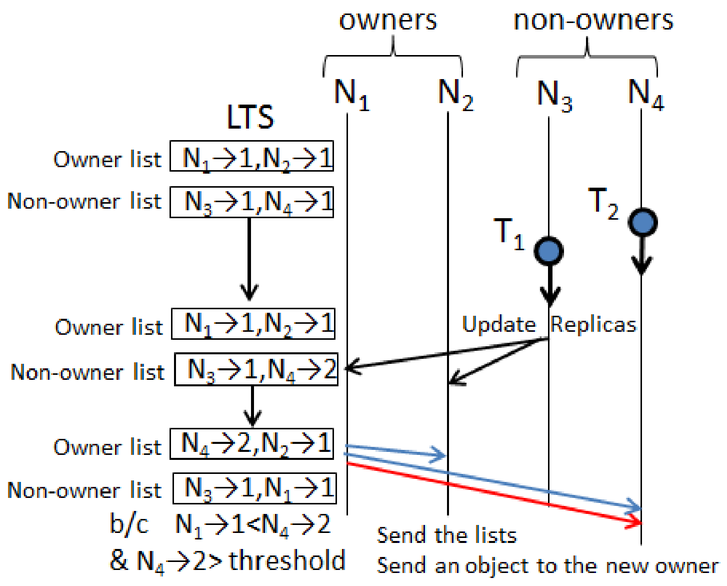


Figure 10.3: Example of LTS

Figure 10.3 illustrates an example of how LTS works with four nodes. The M list includes nodes N_1 and N_2 and the N list includes nodes N_3 and N_4. Let us assume that transactions T_1 and T_2, invoked at nodes N_3 and N_4, respectively, access an object o. If T_1 commits first, T_2 will be aborted. o's owners (i.e., N_1 and N_2) will increase N_4's abort ratio from 1 to 2 (i.e., $N_4 \rightarrow 2$)). When the owners receive the new version of o from T_1, they will build $\acute{N}$, which will include N_4 because the abort rate of N_1 is less than that of N_4, which is larger than a threshold. Node N_1 will now send the new M and N lists to N_2 and N_4. Object o updated by T_1 is sent to only N_4, because N_2 was a previous owner.

10.2.3 Increasing Concurrency

When shared objects are composed of multiple fields, the access probability for the fields can sometimes be skewed. Moreover, the data access pattern of different transactional profiles can logically group object fields into subsets that are concurrently accessed by different transactions. As a result, updating two fields of the same object from different transactions causes an abort due to object invalidation. However, such an abort does not compromise correctness, because the actual data accessed and modified are different.

An illustrative example of this scenario can be found in the TPC-C benchmark [83]. TPC-C provides each object with an unique key, which is needed by transactions to retrieve and manipulate the object [90]. If multiple transactions concurrently access different fields of the same object, a conflict occurs. In this case, only one transaction is allowed to commit, aborting the others. In the TPC-C specification, the transaction profiles *New Order* and *Payment* access an object called *District*. When both request the same *District*, a write-write conflict happens. However, *New Order* updates the field *D_NEXT_O_ID*, while *Payment* updates the field *D_YTD*. Hence, this is not a real conflict.

The contention probability can be alleviated by splitting heavily contended, "hot spot" objects, like the *District* object in the previous example, onto multiple nodes, minimizing the conflicts and thereby increasing concurrency. In order to do that, we define two parameters for managing the replication of shared objects. The first is the traditional *object replication degree* (ORD), which defines the number of owners for each object. The second is called the *field replication degree* (FRD), which defines the maximum number of replicas for each object field. For example, when ORD=3, each shared object is maintained by three different nodes in the system. When FRD=2, the object can be split, storing fields in different replicas. With three nodes available for maintaining each object (i.e., the object replicas), the fields can be stored in two of the three nodes, ensuring fault-tolerance (i.e., more than one copy is available).

Formally, in order to preserve object integrity in case of faults, FRD must be higher than one and lower than or equal to ORD (i.e., 1$< FRD < ORD$).

LTS determines whether to split one object into multiple objects, each of which has a subset of the original object fields. Toward that, LTS exploits the replicas already assigned for that object. If there are more than two replicas per object, LTS "guesses" the object contention probability and moves the most requested fields out from the original object container.

The object owners hold information regarding the aborted transactions, which is used for exploiting locality (see Subsection 10.2.2), and the fields that the aborted transactions attempted to update. The latter information is considered for moving fields. LTS determines whether object ownership needs to be changed and the best fields to move accordingly. If the access pattern exhibited by aborted transactions allows the identification of most requested fields, then LTS splits the object into multiple pieces, minimizing the contention.

LTS continuously monitors the state of current and new "hot spots" in the system, separating (or aggregating) fields according to the actual workload.

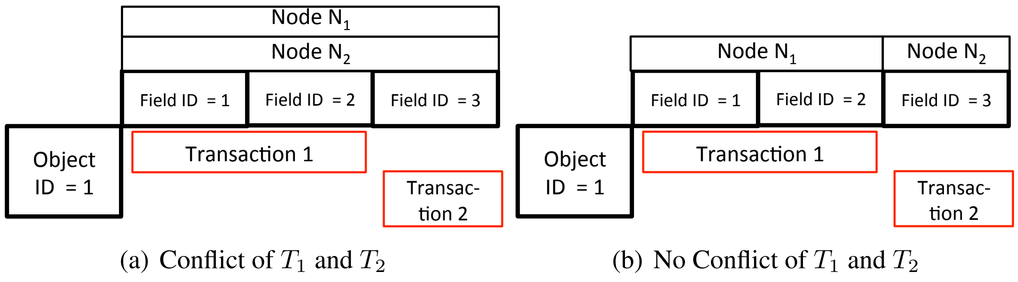


Figure 10.4: T_1 and T_2 access object ID 1.

Figure 10.4 shows how two aborted transactions T_1 and T_2 take advantage of object separation. For simplicity, we consider $ORD = 1$ and $FRD = 2$. The object $o_{id} = 1$ is composed of three fields (o_1, o_2, o_3) and it is stored in two nodes N_1 and N_2. We assume that T_1 executes on N_1 and T_2 executes on N_2. T_1 is interested in updating the fields o_1 and o_2, while T_2 is interested in updating the field o_3. When T_1 and T_2 concurrently access and update $o_{id} = 1$, they access the local copy, and will be aborted at commit time due to object invalidation (Figure 10.4(a)). However, no real invalidation occurs here. LTS detects this and splits $o_{id} = 1$, pushing o_1 and o_2 to N_1, and o_3 to N_2 (Figure 10.4(b)). In this way, it minimizes the contention on $o_{id} = 1$, reducing aborts.

10.2.4 Protocol Correctness

LOCAL-FLOW ensures 1-copy serializability (1CS) [57]. The proof is straightforward and is omitted, but we provide the basic reasoning:

A) The protocol does not rely on a multi-version scheme. Indeed, each object has only one global version available. If two operations performed by different transactions access the same object, and at least one is a write operation, they generate a conflict. Such conflicts are resolved by aborting at least one of the two concurrent transactions.

B) Before updating the written objects, the protocol implements mutual exclusion based on distributed locking (i.e., two-phase-commit [57]). After sending the lock request for an object, a transaction waits until all the replicas reply. Subsequently, the transaction compares the collected pairs <#retry,#locks> with its local pair. Only if the transaction has the highest #retry and highest #locks, it can proceed; otherwise the transaction is aborted and restarted. All concurrent committing transactions collect the same set of pairs sent by object owners along with the NACK message. If multiple transactions have the same <#retry,#locks>, then all are aborted and restarted (after an exponentially distributed back-off time). Only one transaction at a time can access a shared object in the write mode for updating. Deadlock detection is implemented using a timeout based strategy.

C) Once all the objects belonging to a transaction's write-set are locked, the protocol checks whether the transaction has observed an up-to-date view of the shared data, despite optimistic reads and concurrent updates. This validation compares the current timestamp of the read objects with the timestamp of the current committed version. Given that each transaction increases an object's timestamp every time it commits a new version, validation detects previous commits that have made the object inconsistent. Only after a successful read-set validation, a transaction can safely proceed to commit; otherwise, an abort is issued.

By locking all objects in the write-set and subsequently validating the read-set, each committing transaction acts as in centralized settings, blocking all the concurrent participants. The protocol ensures that only one conflicting transaction at a time proceeds to commit.

10.2.5 Algorithm Description

When a transaction starts, the $open$ function firstly checks the location of the corresponding object ID querying its local cache (called *CurrMap*). Only when the object is not locally maintained, a remote lookup is issued.

The $Validation$ procedure illustrated in Algorithm 11 is invoked when the transaction starts its commitment phase. A transaction needs to acquire locks on write-set first. For each object in write-set, the transaction sends a lock request to its owners, along with two parameters (i.e., $\#locks$, and $\#reruns$). Then it waits until it receives the corresponding ACK or $NACK$. In case all NACKs are received, the transaction aborts and restarts after a back-off time, increasing its $\#reruns$. When the transaction collects all ACKs, then it can safely consider the object successfully locked. When the transaction receives at least one ACK and other NACKs, $compareTwoFactors$ is responsible for comparing all the collected pairs of <#retry,#locks> associated with the transactions that received the ACK against their lock requests on the object. Only when the transaction has the pair with maximum $\#reruns$ or, in case of a tie for the maximum $\#reruns$, the maximum $\#locks$, then it can consider the lock granted, otherwise it has to abort and restart.

Only after the successful lock acquisition of all the objects in the write-set, the transaction calls $validateReads$ for checking the consistency of objects in its read-set. Unless the objects in read-set are modified, the replicas storing the objects in write-set will be updated, finalizing the transaction commitment phase.

Algorithm 12 shows the work of the locality-aware transactional scheduler invoked by an object owner when a transaction commits. We define a fixed time window, called *time-frame* (e.g., 10 sec) in which we measure the number of aborts observed by a node. Whenever an abort occurs, the abort rate for an element in $ownerList$ or $nonownerList$ is updated. We notice that the abort rate per node is maintained as a number of aborted transactions, started on that node, caused by a specific object in the last time-frame. Both $ownerList$ and $nonownerList$ consist of pairs $<$ key,value $>$ = $<$ $node$,$abortRate$ $>$. The elements in the lists are sorted according to the $abortRate$ value. Whenever the abort rate increases the $ownerList$'s sorting is refreshed in descending order of abort rate, and $nonownerList$ in ascending order of abort rate. Thus, the complexity of sorting

Algorithm 11: Local-Flow commitment phase

```
Procedure Validation
Input: write-set, read-set
Output: commit, abort
$local\_num\_locks$ = write-set.size();
foreach object obj ∈ write-set do
    send lock request with $< local\_num\_locks, local\_reruns >$ to $obj$'s owners;
    result = retrieve $< num\_locks, reruns >$ from all $obj$'s owners;
    if result == NACKs then
        ▷ All NACKs are received from the owners
        return abort;
    ▷ if at least one ACK is received from the owners
    if compareTwoFactors(result) == false then
        return abort;
▷ At this stage, all objects are successfully locked
if validateReads(read-set) == false then
    ▷ $validateReads$ checks for whether or not the objects in read-sets have been committed by previous concurrent
    transactions
    return abort;
foreach object obj ∈ write-set do
    updateReplicas(obj);
return commit;
Procedure Abort
Input: $local\_reruns$
$local\_reruns$ ++;
wait back-off time;
restart transaction;
```

the lists is $O(\log n)$, where n nodes invoking aborted transactions.

When a transaction updates an object to the owners, other transactions accessing the same object are aborted. The owners increase the abort rate of $ownerList$ and $nonownerList$ corresponding to the updated objects. In order to switch the ownership of an object, the abort rates of $ownerList$ and $nonownerList$ are compared only if the gap between the abort rate of $ownerList$ and $nonownerList$ is larger than a predefined threshold. When this happens, the node in $nonownerList$ becomes the candidate for getting the new ownership. Whenever LTS finds a candidate node, the replicas are sent to that node (the new owner).

Contextually, LTS decides whether or not the object fields are split. After the decision, the new list of object owners is updated to all nodes. This procedure is executed only when the new candidates for being object owners are found.

Whenever an abort is detected, the aborted transaction identifies the differences between original objects, that have been requested, and modified objects, that have been updated but not committed globally. The updated *fieldID*s and the *objectId*s are sent to current owners when a transaction is

Algorithm 12: Locality-aware transactional scheduling phase

```
Procedure LTS
Input: $ownerList$, $nonownerList$
$candidateList$ = null;
foreach {O_node,O_abortRate} in ownerList do
    {node, abortRate} = $nonownerList$.getFirst()
    ▷ getFirst returns the first key and element in the list
    if abortRate - O_abortRate > threshold then
        $nonownerList$.remove(node);
        $candidateList$.append(node);
if $candidateList$.length $\neq 0$ then
    splittingObject($candidateList$);
    updateOwnerList($candidateList$); ▷ updateOwnerList includes changing the ownership
Procedure splittingObject
Input: $candidateList$
$objects$ = null;
▷ $objects$ indicate an object list that will be sent to owners
ids = abortedTxList.get($candidateList$);
▷ get returns the field ids of objects that nodes in $candidateList$ try to update
foreach $node \in candidateList$ do
    if ids.isIndependentSet(node) = true then
        ▷ isIndependentSet returns true or false representing whether or not $node$'s field ids are not overlapped with
        the others
        $objects$ = split(ids($node$));
    else
        objects = ids($node$);
    updateRelicas($objects$, $node$); ▷ send objects to $node$
```

aborted. In the $splittingObject$ procedure in Algorithm 12, the owners maintain $abortedTxList$ used to find which fields aborted transactions tried to update. For each candidate (new object owner), the procedure checks for whether the fields that the candidate wants share with the other candidates'. If there are no candidates, only those fields are sent to object owner. Otherwise, the entire objects are sent. Finally, the new candidate list will be updated to all nodes.

10.3 Experimental Study

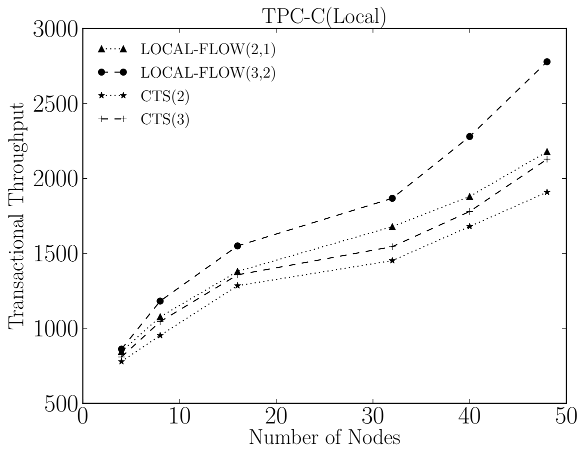
We implemented LOCAL-FLOW in HyFlow2 [91], a high-performance, open-source DTM framework for the JVM, written in Scala.

We conducted experiments using a private cluster and using the Future Grid public infrastructure[1]. Our private cluster consists of 48 nodes, where each node is comprised of 4 Intel Xeon 1.9GHz

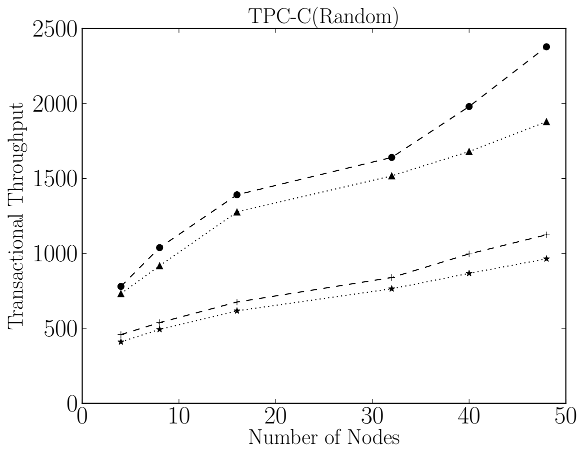
[1]www.futuregrid.org

processor cores, running Ubuntu Linux 10.04 OS. In Future Grid, we used up to 16 nodes (a Future Grid limitation). Future Grid was used to reproduce the same test-bed used in evaluating Score [42].

We compared LOCAL-FLOW against CTS [36] and Score [42]. CTS is a cluster-based transactional scheduler [36] that aborts some transactions in advance to enhance the concurrency of other transactions in a partial replication model. Our benchmarks included TPC-C [83] (representative of on-line transactional workloads) and micro-benchmarks including Linked List and Skip List data structures. To produce locality-aware behavior, we developed two versions of each benchmark: one where object accesses are skewed (called *local*) and the other where transactions access random objects (called *random*).



(a) TPC-C (Local)



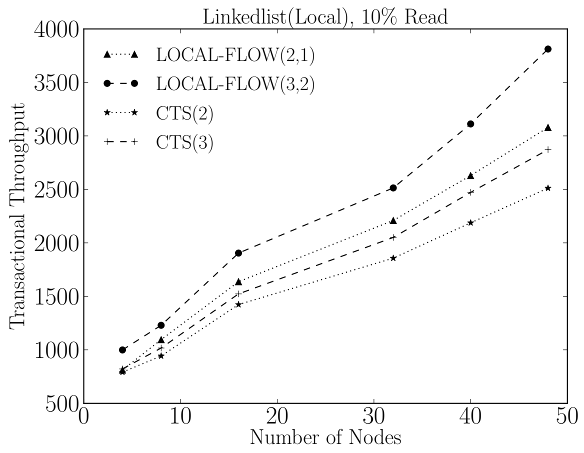
(b) TPC-C (Random)

Figure 10.5: Transactional throughput of TPC-C.

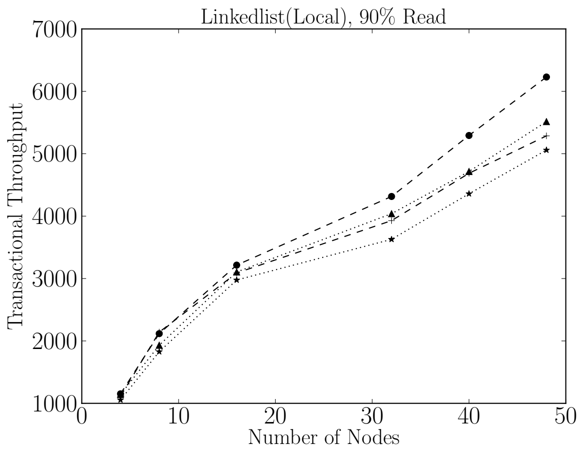
Figure 10.5 shows transactional throughput of TPC-C with local and randomly accessed objects. (The number of read and write transactions were configured according to the TPC-C specification.) Each plot has four curves because we considered configurations with object replication degrees of 2 and 3. For LOCAL-FLOW, this is reflected with ORD=2 and ORD=3. In order to enhance concurrency by object splitting, when ORD=3, we set FRD=2.

In the local setting, both LOCAL-FLOW and CTS exploit data access locality. However, splitting objects under LOCAL-FLOW increases concurrency, improving its performance. In the random scenario, even though an object does not exist in the local cache, LOCAL-FLOW exploits locality through LTS. Indeed, after a certain time, LTS detects the need to move objects closer to requests' source. However, in CTS, even if objects are fetched from a subset of nodes, communication costs are incurred for moving those objects from, even closest, nodes. LOCAL-FLOW improves throughput by as much as 2.2$\times$ over CTS.

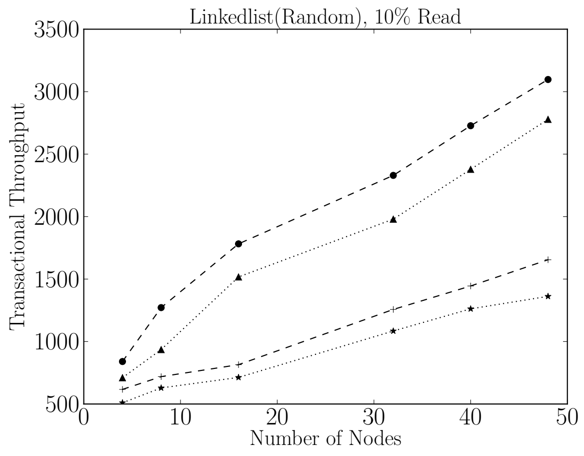
Note that, both CTS and LOCAL-FLOW perform better when replication degree increases. This is due to their ability to exploit locality. However, due to object splitting, LOCAL-FLOW has a lightweight commit cost, as it involves less replicas than CTS (which stores each object along with all its fields).



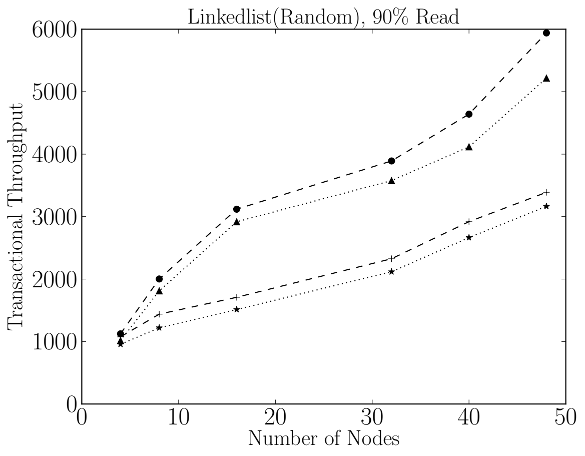
(a) LL (Local) 10%



(b) LL (Local) 90%



(c) LL (Random) 10%



(d) LL (Random) 90%

Figure 10.6: Transactional throughput of Linked List (LL) for different read %.

Figures 10.6 and 10.8 show throughput for Linked List and Skip List. For these micro-benchmarks, we use multiple operations in a transaction. Each operation randomly selects objects to access. Both LOCAL-FLOW and CTS may fetch objects from remote nodes. As the number of aborts increases, LTS determines the optimal location for each object. After several aborts, the scheduler moves objects toward nodes where transactions are suffering increasing aborts. This enables the subsequent incoming requests to fetch objects from their local cache instead of contacting remote owners. We observe that LOCAL-FLOW is effective for micro-benchmarks, improving throughput by as much as 2.6× over CTS.

To compare LOCAL-FLOW with Score [42], we used the same settings as in [42]: we deployed LOCAL-FLOW on Future Grid, and used TPC-C, configured with 50% write transactions and

50% read-only transactions. (Note that Score is integrated into Infinispan [53], and therefore only provides APIs like transactional *put* and *get* on a distributed map. This prohibits using micro-benchmarks such as Linked List or Skip List.)

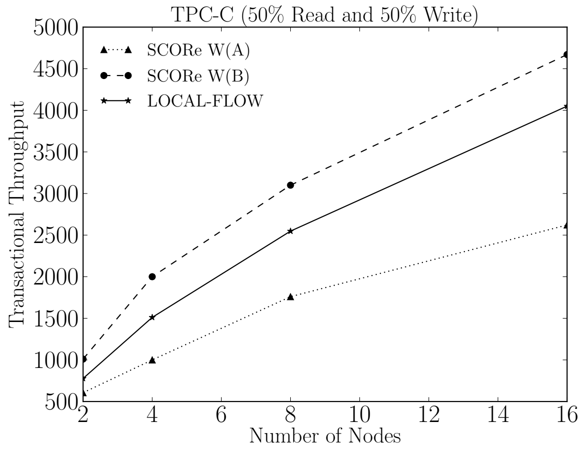
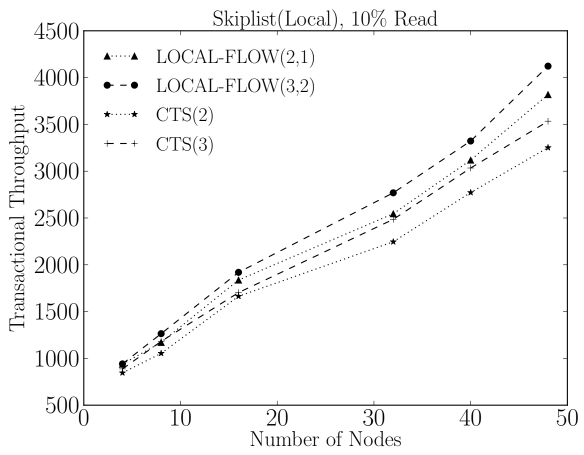


Figure 10.7: Score vs. LOCAL-FLOW.

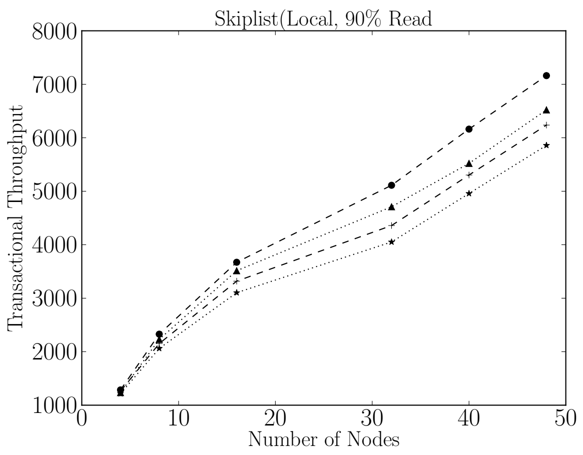
Figure 10.7 shows the results. The plot shows three curves: two for Score and one for LOCAL-FLOW. This is because, Score offers its best performance with manually placed objects. Also, being based on control-flow, it does not react when the workload changes. Therefore, we used two scenarios: *W(A)*, in which most of the transactional requests access local TPC-C objects, and *W(B)*, in which each node executes requests on objects that are mostly located on different nodes. The latter scenario mimics situations in which clients, responsible for doing operations, change their target node in favor of another.

LOCAL-FLOW's performance is not affected by changing the workload from scenario *W(A)* to *W(B)* after an initial phase in which it recognizes the need to move objects for exploiting the new application locality. In contrast, Score is overloaded by communication costs and its potential benefits are negated by the cost of remote operations. Here, LOCAL-FLOW outperforms Score (*W(B)*) by up to 55%. We notice that, the configuration used for this experiments is not the best for LOCAL-FLOW due to the large number of read-only transactions. We nevertheless used it for a fair comparison with the settings in [42].

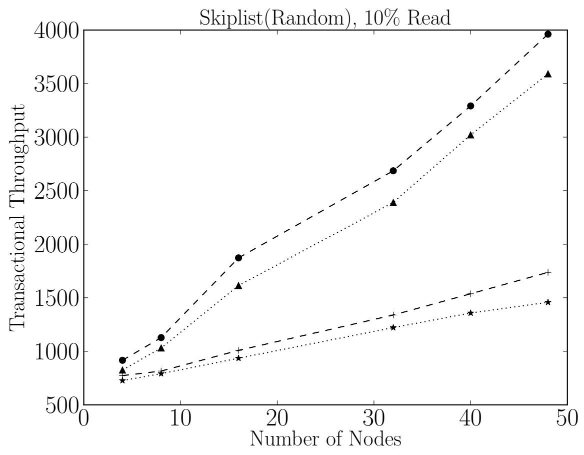
Score's performance in *W(A)* is slightly better than LOCAL-FLOW (by ≈15%). This is mostly due to the configuration of TPC-C. With 50% read-only transactions, Score is better because it does not validate such kind of transactions. In contrast, LOCAL-FLOW validates read-only transactions concurrently with write transactions, generating possible conflicts and object invalidations.



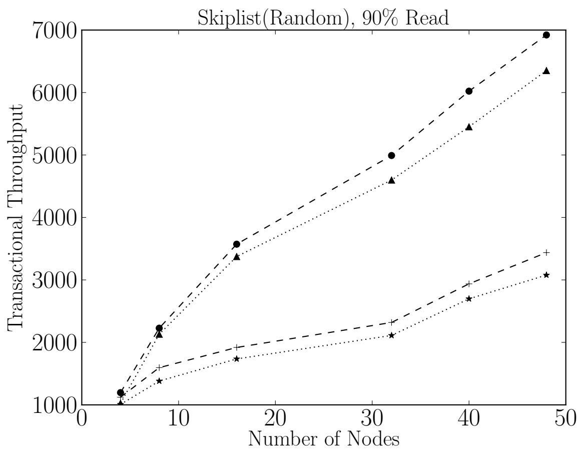
(a) SK (Local) 10%



(b) SK (Local) 90%



(c) SK (Random) 10%



(d) SK (Random) 90%

Figure 10.8: Transactional throughput of Skip List (SL) for different read %.

Chapter 11

Summary, Conclusions, and Future Work

11.1 Summary of Schedulers

In this dissertation, we studied seven different schedulers to improve throughput in data-flow cc DTM. First, Bi-interval categorizes requests into read and write intervals to exploit concurrency of read transactions. The key idea is to minimize object moving times and maximize concurrency of read transactions. Bi-interval enhances throughput by as much as 1.77$\sim$ 1.65$\times$ speedup under low and high contention, respectively.

Second, CRF scheduler maintains a scheduling queue to identify commutative and non-commutative transactions, and could decide to allow all commutative transactions to commit first than the others, maximizing their concurrency. However, despite the significant performance obtained by adopting the idea of commutativity transactions of CRF, there could be applications that do not admit such kind of commutativity. CRF addresses this issue by permitting the developer to explicitly specify non-commutative operations. CRF reveals that transactional throughput is improved by up to 5$\times$ over a state-of-the-art DTM solution.

Third, RTS focuses on scheduling closed-nested transactions. The scheduler heuristically determines transactional contention level to determine whether a live parent transaction aborts or enqueues. RTS is shown to enhance throughput at high and low contention, by as much as 1.53$\times$ and 1.88$\times$ speedup, respectively.

Forth, DATS schedules open-nested transactions. The key idea behind DATS is to avoid compensating actions regardless of conflicted objects and minimize the number of requesting abstract locks, improving performance. Our implementation and experimental evaluation shows that DATS enhances throughput for open-nested transactions by as much as 1.41$\times$ and 1.98$\times$ under low and high contention, respectively.

Fifth, SPN considers parallel closed-nested transactions in DTM. Object owners maintain a transactional table containing on-going inner transactions to identify which inner transactions can be executed in parallel. SPN exploits the parallelism of executing inner transactions and requesting objects in DTM. SPN reveals that throughput is improved by up to 3.5× in micro-benchmarks and up to 4.5× in TPC-C over closed-nested DTM without SPN.

Sixth, CTS focuses on the partial object replication model. The key idea of CTS is to avoid brute force replication of all objects over all nodes to minimize communication overhead. Instead, replicate objects across clusters of nodes, such that each cluster has at least one replica of each object, where clusters are formed based on node-to-node distances. Our implementation and experimental evaluation shows that CTS enhances throughput over GenRSTM and DecentSTM, by as much as (average) 1.55× and 1.73× under low and high contention, respectively.

Seventh, LTS exploits locality in a partial object replication model called LOCAL-FLOW. LTS monitors nodes invoking aborted transactions for detecting the effectiveness of the current object placement. Before the aborted transactions re-start, LTS moves objects' ownership to the nodes. When the aborted transactions restart, fetch the objects from their local cache without remote requesting. Also, transactions typically access only a subset of the field of an object, so the object is splinted into multiple fields according to the need of the transactions. Our results revealed that LOCAL-FLOW outperforms CTS by up to 2.6× on micro-benchmarks and by up to 2.2× on TPC-C. Moreover, LOCAL-FLOW outperforms Score by up to 1.5× on TPC-C when the workload changes.

Bi-interval and CRF focus on single- and multi-version DTM, respectively. CTS and LTS consider partially replicated DTM, which increases concurrency and availability. RTS, DATS, and SPN have been proposed for scheduling closed, open-nested, parallel-nested transactions, respectively.

All seven proposed transactional schedulers focus on data-flow DTM, but consider different object replication, version, and transaction models. In Table 11.1, we summarize the schedulers in terms of these models.

To help DTM programmers identify the scheduler that is best suited for a given application context, in Table 11.2, we characterize the schedulers in terms of their inputs, outputs, strengths, and limitations. Since read-set and write-set are common inputs to all schedulers, Table 11.2 omits those. As inputs, the schedulers consider two types of transactions: "live transactions" and "aborted transactions". Live transactions are those that are in progress, and aborted transactions are those that have been aborted.

Table 11.1: Summary of Proposed Distributed Transactional Schedulers

	Bi-interval	CTS	RTS	DATS	CRF	LTS	SPN
Object version	Single	Single	Single	Single	Multi version	Single	Single
Object copy	No	Partial	No	No	No	Partial	No
Transaction type	Flat	Flat	Closed	Open	Flat	Flat	Closed

Table 11.2: Properties of Proposed Distributed Transactional Schedulers

Bi-interval	Input	Aborted transactions and delay matrix consisting delays for the pair of nodes
	Output	The list of ordered write transactions
	Strength	Minimizing object moving time and maximizing the parallelism of read transactions
	Limitation	Overhead to sort aborted transactions according to their shortest paths
CTS	Input	Live transactions
	Output	A live transaction to be aborted
	Strength	Avoiding the conflicts of live transactions with high dependencies for minimizing overall number of abort
	Limitation	Working only if the size of write-set is at least 2
RTS	Input	Live outer-transactions and their committed inner transactions
	Output	A live outer-transaction to be aborted
	Strength	Minimizing the aborts of inner transactions committed internally
	Limitation	No performance gain if all outer transactions have the same number of committed inner-transactions
DATS	Input	Live outer-transactions and their commutable information
	Output	A live outer-transaction to be aborted
	Strength	Minimizing the compensation action and abstract locks of inner-transactions committed globally
	Limitation	No performance gain if all inner-transactions depend on their outer transaction
CRF	Input	The commutativity of live write transactions
	Output	The list of commutative write transactions
	Strength	Enhancing the concurrency of write transactions through identifying commutative transactions on the same shared objects
	Limitation	No performance gain if all write operations do not commute
LTS	Input	Two lists of owners and non-owner for each object and aborted transactions
	Output	The list of nodes to receive ownership
	Strength	Minimizing remote fetch for each transaction
	Limitation	Incurring object moving time to switch ownership in high contention
SPN	Input	Parallel live inner transactions
	Output	Objects and backoff times
	Strength	Enhancing the parallelism of inner transactions
	Limitation	No performance gain when all the inner transactions have data-dependency

11.2 Conclusions

Bi-interval's design, implementation, and evaluation shows that the idea of grouping concurrent requests into read and write intervals to exploit concurrency of read transactions – originally developed in BIMODAL for multiprocessor TM – can also be successfully exploited for DTM. Doing so poses a fundamental trade-off, however, one between object moving times and concurrency of read transactions. Bi-interval's design shows how this trade-off can be effectively exploited towards optimizing throughput.

CRF focuses on how to enhance concurrency of write transactions in multiversioning schemes ensuring snapshot isolation, where write transactions are exposed to a high probability of conflicts. CRF's key idea is to detect conflicts between commutative and non-commutative write transactions, and allow commutative transactions to commit concurrently before the others. CRF's design shows how commutativity-based scheduling impacts throughput in DTM. Experimental evaluation shows CRF's effectiveness: CRF enhances throughput over a state-of-the-art DTM solution by $3 - 5\times$.

With closed-nested transactions, when an outer transaction is aborted and re-issued, the inner transactions will have to retrieve objects again, increasing communication delays. RTS reduces the aborts of outer transactions, including their inner transactions.

When transactions with committed open-nested transactions conflict later and are re-issued, compensating actions for the open-nested transactions can reduce throughput. DATS avoids this by reducing unnecessary compensating actions, and minimizing inner transactions' remote abstract lock acquisitions through object dependency analysis. DATS's design illustrates the importance of scheduling open-nested transactions in order to reduce the number of compensating actions and abstract locks in case of abort.

SPN focuses on parallel nesting in DTM. With SPN, all inner transactions request their objects from object owners in parallel. This removes the initial overhead of parallelism and reduces overall communication delay by parallel object requests, resulting in high performance. SPN's design illustrates that it is important to hide the communication costs in DTM for improving throughput, and that parallel nesting is an effective way to do so.

CTS uses multiple clusters to support partial replication in fault-tolerant DTM. The clusters are built such that inter-node communication within each cluster is small. To reduce object requesting times, CTS partitions object replicas into each cluster (one per cluster), and enqueues and assigns backoff times for aborted transactions. CTS's design shows that such an approach yields significant throughput improvement.

LTS's design shows that locality matters, and that it can be effectively exploited in a distributed transactional setting. When in-memory operations dominate transaction execution, time spent retrieving objects from remote nodes degrades performance. LOCAL-FLOW's design, implementation, and evaluation demonstrates that transactional protocols for the data-flow model are effective for improving performance in locality-aware transactional applications.

To summarize, transactions may be aborted in DTM due to a number of reasons in different DTM models (i.e., multi-version, replicated, nested), reducing throughput. Our work shows that, identifying the underlying causes (e.g., repeated abort of large write transactions, repeated acquisition of remote objects/abstract locks, the initial overhead of parallel nesting, and low locality on object accesses) and eliminating them can yield significant throughput improvement. The dissertation's results show that the proposed scheduling techniques – which at their core, identify and eliminate those causes – are highly effective.

There are many different DTM models that the dissertation do not consider (e.g., control-flow execution model, full replication model, dynamic system model). However, the dissertation's proposed techniques can likely be adapted and optimized for achieving high performance in those models.

11.3 Future Work

Based on the dissertation's results, we propose the following directions for future research.

An interesting direction is to evaluate some of our proposed schedulers in other models – active replication and speculative execution. These models also target high performance in DTM, so our approaches can enhance their purpose.

- The active replication paradigm exploits full replication to avoid service interruption in case of node failures, allowing transactions to execute locally, costing them only a single network communication step during transaction execution. When a replica is updated globally, a conflict can be detected. Thus, CRF can be applied to the active replication model. Whenever a conflict occurs, CRF checks the replica for commutativity.
- Under speculative execution, transactions accessing objects guess final commit order. Before the commit, other transactions can access the object. Bi-interval, RTS, and CTS give aborted transactions different backoff times to prevent them from aborting again. These schedulers can exploit speculative execution, moving objects that belongs to the write-set of transactions before committing to nodes invoking aborted transactions.

Scheduling transactions in a control-flow DTM is another interesting direction. Past transactional schedulers have been implemented in the data-flow DTM model. Control-flow DTM (i.e., transactions move from node to node and objects are immobile) outperforms data-flow DTM in throughput with low contention. Thus, scheduling transactions in high contention may performs better than data-flow models.

Bibliography

[1] 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*, PPoPP '06, (New York, NY, USA), pp. 187–197, ACM, 2006.

[2] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 102, 2004.

[3] M. Herlihy, V. Luchangco, and M. Moir, "Obstruction-free synchronization: Double-ended queues as an example," in *ICDCS*, (Los Alamitos, CA, USA), p. 522, IEEE Computer Society, 2003.

[4] J. F. Martínez and J. Torrellas, "Speculative synchronization: applying thread-level speculation to explicitly parallel applications," in *ASPLOS-X*, (New York, NY, USA), pp. 18–29, ACM, 2002.

[5] J. Oplinger and M. S. Lam, "Enhancing software reliability with speculative threads," in *ASPLOS-X*, (New York, NY, USA), pp. 184–196, ACM, 2002.

[6] R. Rajwar and J. R. Goodman, "Transactional lock-free execution of lock-based programs," vol. 36, (New York, NY, USA), pp. 5–17, ACM, Oct. 2002.

[7] S. Dolev, D. Hendler, and A. Suissa, "CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory," in *Proceedings of the twenty-seventh ACM symposium on Principles of distributed computing*, PODC '08, (New York, NY, USA), pp. 125–134, ACM, 2008.

[8] T. Harris and K. Fraser, "Language support for lightweight transactions," in *Object-Oriented Programming, Systems, Languages, and Applications*, pp. 388–402, Oct 2003.

[9] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer, "Software transactional memory for dynamic-sized data structures," in *PODC*, pp. 92–101, Jul 2003.

[10] V. J. Marathe, W. N. Scherer III, and M. L. Scott, "Adaptive software transactional memory," in *Proceedings of the 19th International Symposium on Distributed Computing*, (Cracow, Poland), Sep 2005. Earlier but expanded version available as TR 868, University of Rochester Computer Science Dept., May2005.

[11] B. Saha, A.-R. Adl-Tabatabai, and Q. Jacobson, "Architectural support for software transactional memory," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, (Washington, DC, USA), pp. 185–196, IEEE Computer Society, 2006.

[12] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum, "Hybrid transactional memory," in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS XII, (New York, NY, USA), pp. 336–346, ACM, 2006.

[13] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen, "Hybrid transactional memory," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, (New York, NY, USA), pp. 209–220, ACM, 2006.

[14] C. Cao Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *ISCA*, Jun 2007.

[15] A. Shriraman, M. F. Spear, H. Hossain, V. J. Marathe, S. Dwarkadas, and M. L. Scott, "An integrated hardware-software approach to flexible transactional memory," *SIGARCH Comput. Archit. News*, vol. 35, no. 2, pp. 104–115, 2007.

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

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

[18] D. Perelman, A. Bishevsky, *et al.*, "SMV: Selective multi-versioning STM," in *ACM SIGPLAN workshop on Transactional Computing*, 2010.

[19] S. M. Fernandes and J. a. Cachopo, "Lock-free and scalable multi-version software transactional memory," in *Proceedings of the 16th ACM symposium on Principles and practice of parallel programming*, PPoPP '11, (New York, NY, USA), pp. 179–188, ACM, 2011.

[20] J. E. B. Moss, "Open nested transactions: Semantics and support," in *Workshop on Memory Performance Issues,*, 2005.

[21] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," in *Proceedings of the 19th international conference on Distributed Computing*, DISC'05, (Berlin, Heidelberg), pp. 324–338, Springer-Verlag, 2005.

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

[23] M. M. Saad and B. Ravindran, "Supporting STM in distributed systems: Mechanisms and a Java framework," in *Sixth ACM SIGPLAN workshop on Transactional Computing*, 2011.

[24] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues, "D2STM: Dependable distributed software transactional memory," in *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, (Washington, DC, USA), pp. 307–313, IEEE Computer Society, 2009.

[25] P. Romano, N. Carvalho, M. Couceiro, L. Rodrigues, and J. Cachopo, "Towards the integration of distributed transactional memories in application servers clusters," in *Quality of Service in Heterogeneous Networks*, vol. 22, pp. 755–769, Springer Berlin Heidelberg, 2009. (Invited paper).

[26] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, "Cloud-TM: harnessing the cloud with distributed transactional memories," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 1–6, April 2010.

[27] M. J. Demmer and M. Herlihy, "The arrow distributed directory protocol," in *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, (London, UK, UK), pp. 119–133, Springer-Verlag, 1998.

[28] J. Kim and B. Ravindran, "On transactional scheduling in distributed transactional memory ystems," in *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, SSS'10, (Berlin, Heidelberg), pp. 347–361, Springer-Verlag, 2010.

[29] M. M. Saad and B. Ravindran, "Snake: control flow distributed software transactional memory," in *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, (Berlin, Heidelberg), pp. 238–252, Springer-Verlag, 2011.

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

[31] R. M. Yoo and H.-H. S. Lee, "Adaptive transaction scheduling for transactional memory systems," in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, (New York, NY, USA), pp. 169–178, ACM, 2008.

[32] G. Blake, R. G. Dreslinski, and T. Mudge, "Proactive transaction scheduling for contention management," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 42, (New York, NY, USA), pp. 156–167, ACM, 2009.

[33] J. Kim, R. Palmieri, and B. Ravindran, "Enhancing concurrency in distributed transactional memory through commutativity," in *Proceedings of the 19th international conference on Parallel Processing*, Euro-Par'13, (Berlin, Heidelberg), pp. 40–51, Springer-Verlag, 2013.

[34] J. Kim and B. Ravindran, "Scheduling closed-nested transactions in distributed transactional memory," in *Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International*, pp. 179–188, 2012.

[35] J. Kim, R. Palmieri, and B. Ravindran, "Scheduling open-nested transactions in distributed transactional memory," in *Proceedings of the 15th International Conference on Coordination Models and Languages (COORDINATION'13)*, pp. 105–120, 2013.

[36] J. Kim and B. Ravindran, "Scheduling transactions in replicated distribute software transactional memory," in *Proceedings of the 2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCgrid 2013)*, CCGRID '13, (Delft , The Netherlands), IEEE Computer Society, 2013.

[37] J. Kim, R. Palmieri, and B. Ravindran, "LOCAL-FLOW: A locality-aware distributed transactional protocol," in *Techincal Report, Virginia Tech, ECE*, 2013.

[38] R. Guerraoui, M. Kapalka, and J. Vitek, "STMBench7: a benchmark for software transactional memory," *SIGOPS Oper. Syst. Rev.*, vol. 41, no. 3, pp. 315–324, 2007.

[39] B. Zhang and B. Ravindran, "Brief announcement: on enhancing concurrency in distributed transactional memory," in *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, (New York, NY, USA), pp. 73–74, ACM, 2010.

[40] T. Riegel, "Snapshot isolation for software transactional memory," in *Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT06)*, 2006.

[41] T. Riegel, C. Fetzer, and P. Felber, "Time-based transactional memory with scalable time bases," in *Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '07, (New York, NY, USA), pp. 221–228, ACM, 2007.

[42] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Middleware*, pp. 456–475, 2012.

[43] 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*, PPoPP '08, (New York, NY, USA), pp. 207–216, ACM, 2008.

[44] K. Agrawal, C. E. Leiserson, and J. Sukha, "Memory models for open-nested transactions," in *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, (New York, NY, USA), pp. 70–81, ACM, 2006.

[45] 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*, PPoPP '07, (New York, NY, USA), pp. 68–78, ACM, 2007.

[46] W. Baek, N. Bronson, C. Kozyrakis, and K. Olukotun, "Implementing and evaluating nested parallel transactions in software transactional memory," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, (New York, NY, USA), pp. 253–262, ACM, 2010.

[47] K. Agrawal, J. T. Fineman, and J. Sukha, "Nested parallelism in transactional memory," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, (New York, NY, USA), pp. 163–174, ACM, 2008.

[48] J. a. Barreto, A. Dragojević, P. Ferreira, R. Guerraoui, and M. Kapalka, "Leveraging parallel nesting in transactional memory," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 91–100, ACM, 2010.

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

[50] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson, "DiSTM: A software transactional memory framework for clusters," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ICPP '08, (Washington, DC, USA), pp. 51–58, IEEE Computer Society, 2008.

[51] N. Carvalho, P. Romano, and L. Rodrigues, "A generic framework for replicated software transactional memories," in *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pp. 271 –274, aug. 2011.

[52] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized stm algorithm," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, 2010.

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

[54] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 35–40, 2010.

[55] 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*, pp. 455–465, 2012.

[56] 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*, pp. 214–224, 2010.

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

[58] B. Zhang and B. Ravindran, “Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks,” in *Reliable Distributed Systems, 2009. SRDS ’09. 28th IEEE International Symposium on*, pp. 268–277, 2009.

[59] R. Guerraoui, M. Herlihy, and B. Pochon, “Toward a theory of transactional contention managers,” in *Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, PODC ’05, (New York, NY, USA), pp. 258–264, ACM, 2005.

[60] K. Manassiev, M. Mihailescu, and C. Amza, “Exploiting distributed version concurrency in a transactional memory cluster,” in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP ’06, (New York, NY, USA), pp. 198–208, ACM, 2006.

[61] H. E. Ramadan, C. J. Rossbach, and E. Witchel, “Dependence-aware transactional memory for increased concurrency,” in *Proceedings of the 41st annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 41, (Washington, DC, USA), pp. 246–257, IEEE Computer Society, 2008.

[62] R. Guerraoui, T. A. Henzinger, and V. Singh, “Permissiveness in transactional memories,” in *Proceedings of the 22nd international symposium on Distributed Computing*, DISC ’08, (Berlin, Heidelberg), pp. 305–319, Springer-Verlag, 2008.

[63] E. B. Moss, “Nested transactions: An approach to reliable distributed computing,” tech. rep., Cambridge, MA, USA, 1981.

[64] H. Garcia-Molina, “Using semantic knowledge for transaction processing in a distributed database,” *ACM Trans. Database Syst.*, vol. 8, pp. 186–213, June 1983.

[65] G. Weikum, “Principles and realization strategies of multilevel transaction management,” *ACM Trans. Database Syst.*, vol. 16, pp. 132–180, Mar. 1991.

[66] J. E. B. Moss and A. L. Hosking, “Nested transactional memory: model and architecture sketches,” *Sci. Comput. Program.*, vol. 63, pp. 186–201, December 2006.

[67] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood, “Supporting nested transactional memory in logTM,” *SIGPLAN Not.*, vol. 41, no. 11, pp. 359–370, 2006.

[68] K. Agrawal, I.-T. A. Lee, and J. Sukha, “Safe open-nested transactions through ownership,” in *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA ’08, (New York, NY, USA), pp. 110–112, ACM, 2008.

[69] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, “STAMP: Stanford transactional applications for multi-processing,” in *Workload Characterization, 2008. IISWC 2008. IEEE International Symposium on*, pp. 35–46, 2008.

[70] B. Zhang and B. Ravindran, “A quorum-based replication framework for distributed software transactional memory,” in *Proceedings of the 15th international conference on Principles of Distributed Systems*, OPODIS’11, (Berlin, Heidelberg), pp. 18–33, Springer-Verlag, 2011.

[71] R. Guerraoui and L. Rodrigues, *Introduction to Reliable Distributed Programming*. 2006.

[72] N. Carvalho, P. Romano, and L. Rodrigues, “Asynchronous lease-based replication of software transactional memory,” in *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware ’10, (Berlin, Heidelberg), pp. 376–396, Springer-Verlag, 2010.

[73] C. Gray and D. Cheriton, “Leases: an efficient fault-tolerant mechanism for distributed file cache consistency,” vol. 23, (New York, NY, USA), pp. 202–210, ACM, Nov. 1989.

[74] J. Baker, C. Bond, J. C. Corbett, J. Furman, A. Khorlin, J. Larson, J.-M. Leon, Y. Li, A. Lloyd, and V. Yushprakh, “Megastore: Providing scalable, highly available storage for interactive services,” in *Proceedings of the Conference on Innovative Data system Research (CIDR)*, pp. 223–234, 2011.

[75] W. Maldonado, P. Marlier, P. Felber, A. Suissa, D. Hendler, A. Fedorova, J. L. Lawall, and G. Muller, “Scheduling support for transactional memory contention management,” in *PPoPP ’10: Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, (New York, NY, USA), pp. 79–90, ACM, 2010.

[76] A. Dragojević, R. Guerraoui, A. V. Singh, and V. Singh, “Preventing versus curing: avoiding conflicts in transactional memories,” in *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC ’09, (New York, NY, USA), pp. 7–16, ACM, 2009.

[77] M. Ansari, M. Luján, C. Kotselidis, K. Jarvis, C. C. Kirkham, and I. Watson, “Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering,” in *HiPEAC* (A. Seznec, J. S. Emer, *et al.*, eds.), vol. 5409 of *LNCS*, pp. 4–18, Springer, 2009.

[78] A. Turcu and B. Ravindran, “On open nesting in distributed transactional memory,” in *Proceedings of the 5th Annual International Systems and Storage Conference*, SYSTOR ’12, (New York, NY, USA), pp. 12:1–12:12, ACM, 2012.

[79] A. Turcu and B. Ravindran, "On closed nesting in distributed transactional memory," in *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

[80] D. J. Rosenkrantz, R. E. Stearns, and P. M. L. II, "An analysis of several heuristics for the traveling salesman problem," *SIAM J. Comput.*, vol. 6, no. 3, pp. 563–581, 1977.

[81] T. Riegel, C. Fetzer, H. Sturzrehm, and P. Felber, "From causal to z-linearizable transactional memory," in *Proceedings of the Twenty-Sixth Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp. 340–341, 2007.

[82] C. A. R. Hoare, "Communicating sequential processes," *Commun. ACM*, 1978.

[83] T. Council, "TPC-C benchmark, revision 5.11," Feb 2010.

[84] S. Büttcher and C. L. A. Clarke, "Indexing time vs. query time: trade-offs in dynamic information retrieval systems," CIKM '05, pp. 317–318, ACM, 2005.

[85] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Commun. ACM*, vol. 13, pp. 422–426, July 1970.

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

[87] R. Guerraoui and M. Kapalka, "Transactional memory: Glimmer of a theory," in *CAV*, vol. 5643 of *LNCS*, pp. 1–15, Springer Berlin, 2009.

[88] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, pp. 359–392, 1998.

[89] M. M. Saad and B. Ravindran, "Transactional forwarding: Supporting highly-concurrent stm in asynchronous distributed systems," in *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pp. 219–226, 2012.

[90] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland, "The end of an architectural era: (it's time for a complete rewrite)," in *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pp. 1150–1160, VLDB Endowment, 2007.

[91] A. Turcu, B. Ravindran, and R. Palmieri, "Hyflow2: A high performance distributed transactional memory framework in scala," in *10th International Conference on Principles and Practices of Programming on JAVA platform: virtual machines, languages, and tools*, PPPJ '13, September 2013.