Scheduling Memory Transactions in Distributed Systems

Junwhan Kim

Preliminary Examination Proposal 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

Binoy Ravindran, Chair
Robert P. Broadwater
Paul E. Plassmann
Anil Vullikanti
Maurice Herlihy

May 1, 2012
Blacksburg, Virginia

Keywords: Software Transactional Memory, Distributed Systems, Scheduling

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 complimentary 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*, PTS, CTS, RTS, and $DATS$. 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, on average.

PTS considers multi-versioned DTM. Traditional multi-versioned TM models use multiple object versions to guarantee commits of read transactions, but limits concurrency of write transactions. PTS detects conflicts of write transactions at an object level. Instead of aborting a transaction due to an object-level conflict, PTS assigns backoff times for conflicting transactions. Our implementation reveals that PTS improves throughput over competitors including GenRSTM and DecentSTM by as much as 3.4×, on average.

CTS considers replicated DTM: object replicas are distributed to 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 and assigns backoff times for aborted transactions due to early validation over clusters, reducing communication overhead. Implementation reveals that CTS improves throughput over competitor replicated D-STM solutions including GenRSTM and DecentSTM by as much as 1.64×, on average.

RTS and DATS consider transactional nesting in DTM, and respectively focuses on the *closed* and *open* nesting models. 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 detects object dependencies between open-nested transactions and their outer transaction. When conflicts occur, only the involved operations are restarted to avoid executing unnecessary compensating actions and minimize inner transactions' attempt for acquiring remote abstract locks. Implementations reveal effectiveness: RTS and DATS improve throughput (over the no-scheduler case), by as much as 88% and 98%, respectively.

Our major proposed post-preliminary research is to develop schedulers which satisfy consistency criteria that are weaker than the opacity criteria targeted by the pre-preliminary schedulers, toward improving concurrency. Example such criteria include update serializability (US) and strong even-

tual consistency (SEC). While US guarantees commit for read-only transactions, SEC ensures that object states will eventually converge, improving concurrency at the expense of weaker consistency. Additional directions include evaluating the schedulers using industrial/production-strength benchmarks (e.g., TPC-B, Berkeley DB, and Yahoo cloud serving benchmark).

This work is supported in part by NSF CNS 0915895, NSF CNS 1116190, and NSF CNS 1130180.

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 it is 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 in-memory data objects that promises to alleviate the difficulties of lock-based synchronization (i.e., scalability, programmability, and composability issues). With TM, code that read/write shared objects is organized as *memory transactions*, which speculatively execute, while logging changes made to 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 [65] and is composable. TM for multiprocessors has been proposed in hardware (HTM) [29,

33, 46, 53, 57], in software (STM) [20, 32, 34, 45, 66], and in hardware/software combination [17, 42, 13, 72].

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 [56]. 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 [37]. 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 [55, 56, 22].

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* [49]. 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.

Compared to open-nesting, the flat and closed nested models may limit concurrency: when a large transaction is aborted, all its flat/closed-nested transactions are also aborted and rolled back, even if they don't conflict with any other transaction. (Of course, closed nesting potentially offers higher concurrency than flat nesting.) In contrast, 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 [51]. In contrast, if the inner transaction were to be closed- or flat-nested, then those object changes are not made visible (i.e., the objects are not "released") until the outer transaction commits, potentially causing conflicts to other transactions who may want to use those objects. Thus, open-nesting potentially offers higher

concurrency than closed and flat nesting.

To achieve high concurrency in open nesting, inner transactions have to implement *abstract serializability* [52]. 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 committed to 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 checking for conflicts, but I must acquire an *abstract lock* and propagate this lock for A. Two non-commutative operations [1] would try to acquire the same abstract lock, so open nesting performs concurrency control at an abstract level.

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 (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 (cc DTM) [35, 78, 63], in which a set of nodes communicate with each other by message-passing links over a communication network, and a cluster model (cluster DTM) [16, 60, 61], in which a group of linked computers works closely together to form a single computer. The most important difference between the two is communication cost. cc DTM assumes a *metric-space* network (i.e., the communication cost between nodes form a metric), whereas cluster DTM differentiates between local cluster memory and remote memory at other clusters.

Most cc DTM works consider Herlihy and Sun's dataflow execution model [35], 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 [18, 35, 78], 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 [16, 60, 61, 38, 63, 64].

1.3 Transactional Scheduling

As mentioned before, a complimentary approach for dealing with transactional conflicts is transactional scheduling. Broadly, a transactional scheduler determines the ordering of concurrent trans-

[1] Two method invocations commute if applying them in either order leaves the object in the same state and returns the same response.

actions so that conflicts are either avoided altogether or minimized. Two kinds of transactional schedulers have been studied in the past: reactive [20, 5] and proactive [77, 7]. 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 [20, 5]. *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 [77, 7]. 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 five different transactional schedulers that we have developed. The schedulers target data-flow cc DTM and are called *Bi-interval*, *PTS*, *CTS*, *RTS*, and *DATS*.

Scheduling in single-version DTM. We first consider the 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 [7]. We present a novel transactional scheduler called Bi-interval [38] 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 [28]. 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 parallelism 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 DTM (i.e., MV-STM). Unless a conflict between operations to access a shared object occurs, MV-STM allows the corresponding transactions to read the object's old versions. Thus, MV can potentially enhance concurrency in distributed TM. However, in data-flow cc DTM, where objects are migrated, object versions may be "scattered" in the network. This causes difficulty in maintaining the consistency of the versions, and may incur high communication delay for doing so. Even with MV-STM, when conflict between two operations occurs, aborting an involved transaction is inevitable. Furthermore, a distributed transaction consumes longer execution time including communication delays to request and acquire objects than a transaction on multiprocessors, so the probability for conflicts increases. In addition, since MV-STM cannot hold all possible versions of an object forever, *garbage collection* (GC) is needed for removing obsolete versions. This raises difficulties on determining when and how the versions can be removed. Unless some versions are kept, the transactions needing those versions will abort due to the limited number of versions.

To boost performance in (multi-version) DTM, a transactional scheduler is therefore compelling to consider, as it effectively stalls contending transactions when conflicts occur. However, in MV DTM, reactive and proactive transactional schedulers may not be effective. This is due to several

reasons.

First, MV-STM inherently guarantees commits of all read-only transactions [56]. Past transactional schedulers [5, 38] abort loosing read-only transactions due to conflicts and simultaneously restart the aborted read-only transactions to maximize their concurrency. However, conflicts with read-only transactions do not occur in MV-STM due to multiple object versions. Thus, the concurrency of read-only transactions cannot be exploited by traditional scheduling approaches in MV-STM.

Second, a transaction may request multiple objects. However, a conflict occurs and is only detected on a single object [10]. Even though other objects used by the transaction may not be subject to a conflict, the transaction is still aborted. Once a transaction is aborted, it will suffer from additional communication delays to request and retrieve all its objects again in data-flow cc DTM. Due to such delays, determining backoff times for aborted transactions (under proactive schedulers) or serializing enqueued, aborted transactions (under reactive schedulers) is generally difficult.

We overcome these difficulties by designing a transactional scheduler, called *progressive transactional scheduler* (or PTS). PTS considers an event-based cc DTM model: when transactions request and acquire an object (version), events that track the object versions are recorded. The events indicating which transaction reads or updates an object are used to detect which object is subject to the conflict. After a conflict is detected, PTS assigns different backoff times for conflicting transactions. If a new object version is created and conflicting transactions needing it exist, PTS sends the version to the requesting nodes.

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) [9, 40, 16, 14, 6]. 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 [63]. 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. We develop a transactional scheduler for this model, called *cluster-based transactional scheduler* (or CTS). CTS enqueues and assigns backoff times for aborted transactions due to early validation over clusters, reducing communication overhead. If an object is created and enqueued transactions needing it exist, CTS sends the object to the enqueued transactions, reducing communication delays and increasing the concurrency of read transactions.

Scheduling nested transactions. We now turn our attention to 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.

We first consider scheduling closed-nested transactions, which is more efficient than flat nesting and guarantees *serialization* [3]. We present a transactional scheduler for closed-nested transactions, called the *reactive transactional scheduler* (or RTS), which 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 the enqueued transaction a backoff time to boost throughput.

We then consider scheduling open-nested transactions. Unlike abstract serializability for open-nested transactions [52], an outer transaction uses a different strategy to guarantee serializability. 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, the commit protocol for each open-nested transaction must acquire and release the locks, degrading performance.

In the mean time, 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 [3]. 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, we present a scheduler, called the *dependency-aware transactional scheduler* (or DATS). The basic idea of DATS is to reduce conflicts of outer transactions and the number of abstract locks of inner transactions, which in turn, minimizes communication overheads and compensating actions. When an outer transaction aborts, DATS identifies conflicting objects and executes compensating actions only for those involved in the conflicts. If inner transactions acquire abstract locks regardless of the conflicted objects, their locks (issued by the abstract lock mechanism) will be preserved to boost throughput.

1.4 Summary of Current Research Contributions

We now summarize our contributions.

The design of Bi-interval shows that the idea of grouping concurrent requests into read and write intervals to exploit concurrency of read transactions — originally developed in the BIMODAL scheduler for multiprocessor TM [5] — 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 exploited towards optimizing throughput. We show that 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. We implemented Bi-interval in the HyFlow Java DTM framework [63] and experimentally evaluated using micro benchmarks (e.g., Red/Back Tree) and macro benchmarks (e.g., Bank, Loan, distributed versions of Vacation from the STAMP benchmark suite [12]). Our evaluation shows that Bi-interval improves throughput over the no-scheduler case by as much as 1.77$\sim$ 1.65$\times$ speedup under low and high contention, respectively.

PTS targets multi-version DTM. As mentioned before, when transactions request and acquire an object version, events that track the object version are recorded to detect conflicts. Our key idea is to detect which object version is subject to the conflict in event-based cc DTM and assign backoff times to conflicting transactions to minimize communication overhead. We also show that PTS is *opaque* and *strongly progressive* [26]. Our implementation and experimental evaluation using two monetary applications (Bank and Loan) and two distributed data structures including Counter and Red/Black Tree (RB-Tree) [12] as micro benchmarks shows that PTS enhances throughput over replicated DTM solutions, GenRSTM [14] and DecentSTM [6], by as much as (average) 3.4$\times$ and 3.3$\times$ under low and high contention, respectively.

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. For this model, CTS schedules concurrent requests needing a replicated object when a conflict occurs at the object level, by assigning backoff times to enqueued aborted transactions. Our implementation and experimental evaluation shows that CTS enhances throughput over GenRSTM [14] and DecentSTM [6], by as much as (average) 1.55$\times$ and 1.73$\times$ under low and high contention, respectively.

In closed-nested DTM, RTS heuristically determines transactional contention level to decide whether a live parent transaction aborts or enqueues, and a backoff time that determines how long a live parent transaction waits. Our experimental evaluation validates our idea: RTS is shown to enhance throughput (over the no-scheduler case) at high and low contention, by as much as 1.53 (53%) $\sim$ 1.88 (88%) $\times$ speedup, respectively.

DATS's design goal is to improve throughput of open-nested DTM . When open-nested transac-

tions acquire and release abstract locks of objects, communication delays are incurred. If outer transactions abort, compensating actions are executed. The key idea behind DATS is to avoid compensating actions regardless of conflicted objects and minimize the number of abstract lock requests, improving performance. Our implementation and experimental evaluation shows that DATS enhances throughput for open-nested transactions (over the no-scheduler case), by as much as 1.41× and 1.98× under low and high contention, respectively.

1.5 Proposed Post Preliminary-Exam Work

Based on these research results, we propose the following research directions for post preliminary exam work:

- Design transactional schedulers that satisfy consistency criteria weaker than opacity such as a) *update serializability* [31] and b) *strong eventual consistency* [71].
- Evaluate the performance of the pre- and post-preliminary transactional schedulers using industrial-strength benchmarks including TPC-B, Berkeley DB, and Yahoo cloud serving benchmark.

1.6 Proposal Outline

The rest of the proposal 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, and 8 describe the Bi-inteval, PTS, CTS, RTS, and DATS schedulers, respectively. Each chapter discusses motivation, designs the scheduler (informally), provides an illustrative example, presents the corresponding algorithms, analyzes the scheduler properties, and evaluates performance. We conclude and describe the proposed post-exam work in Chapter 9.

Chapter 2

Past and Related Work

2.1 Distributed Transactional Memory

Herlihy and Sun proposed distributed STM [35]. 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 [35], 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 [19] 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 [78, 80], which improves scalability by using a peer-to-peer structure. They also present a class of location-aware cache-coherence (or LAC) protocols [79], which improve the makespan competitive ratio, with the optimal Greedy contention manager [25].

While these efforts focused on distributed STM's theoretical properties, other efforts developed implementations. In [9], 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 [44], 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 [16]. 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 [60], and Cloud platforms [61]. In [60], 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 [61], 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) [58], 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.* [24]. 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 [37], does not allow transactions to abort unless live transactions violate π.

In [56], 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 [37], 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 [48]. This work focused on the popular two-phase locking protocol and extended it to support nesting. Also, it proposed algorithms for distributed transaction man-

agement, object state restoration, and distributed deadlock detection.

Open nesting also originates in the database community [23], and was extensively analyzed in the context of undo-log transactions [76]. 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 [51]. 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 [50], explaining how using multiple levels of abstractions can help in differentiating between fundamental and false conflicts and therefore improve concurrency.

Moravan et al. [47] 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. combine closed and open nesting by introducing the concept of transaction ownership [2]. 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.

2.4 Transactional Scheduling

Transactional scheduling has been explored in a number of multiprocessor STM efforts [43, 21, 4, 77, 20, 5]. In [21], Dragojević *et. al.* describe an approach that dynamically schedules transactions based on their predicted read/write access sets. In [4], 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) [77] 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 [20], 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 in [7]. Their scheme

detects hot spots of contention that can degrade performance, and proactively schedules affected transactions around the hot spots. Evaluation on the STAMP benchmark suite [12] shows their scheduler outperforming a backoff-based policy by an average of 85%.

Attiya and Milani present the BIMODAL scheduler [5], 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. Kim and Ravindran extend the BIMODAL scheduler for cc DTM in [38]. Their scheduler, called Bi-interval, groups concurrent requests into read and write intervals, and exploits the tradeoff between object moving times (incurred in dataflow cc DTM) and concurrency of read transactions, yielding high throughput.

Steal-On-Abort, CAR-STM, and BIMODAL enqueue aborted transactions to minimize future conflicts in SV-STM. In contrast, PTS only enqueues live transactions that conflict with other transactions. The purpose of enqueuing is to prevent contending transactions from requesting all objects again. Of course, enqueuing live transactions may lead to deadlock or livelock. Thus, PTS assigns different backoff times for each enqueued live transaction.

ATS and Proactive Transactional Scheduler determine contention intensity and use it for contention management. Unlike these schedulers which are designed for multiprocessors, predicting contention intensity will incur communication delays in distributed systems. Thus, our proposed schemes collect average running times – a history of how long transactions run – to compute a backoff time. Unlike multiprocessor STM, two communication delays will be incurred to obtain an object, one for requesting an object and the other for retrieving it. Enqueuing a live transaction during the backoff time saves those communication delays.

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 [35], 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 [35], 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.

3.2 Atomicity, Consistency, and Isolation

We use the *Transactional Forwarding Algorithm* (TFA) [63] 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).

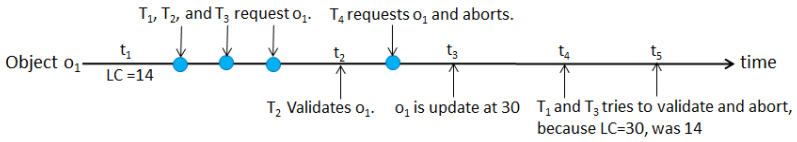


Figure 3.1: An Example of TFA

For completeness, we illustrate TFA with an example. In Figure 3.1, 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, PTS, and CTS are associated with TFA. RTS is associated with nested TFA (N-TFA) [73], that is an extension of TFA to implement closed nesting in DTM. DATS is associated with TFA with Open Nesting (TFA-ON) [74], which extends the TFA algorithm [63], to manage open-nested transactions. N-TFA and TFA-ON change the scope of object validations.

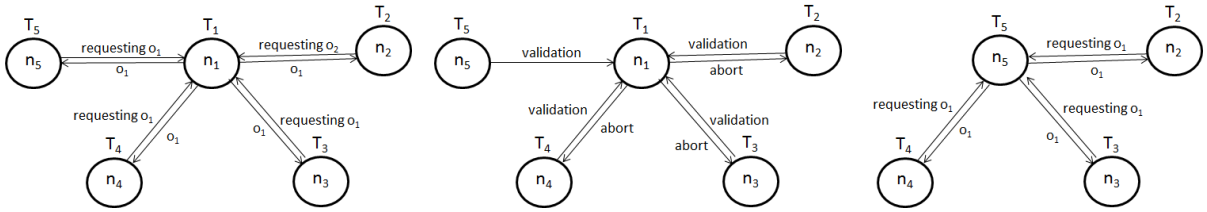
The behavior of open-nested transactions under TFA-ON is similar to the behavior of regular transactions under TFA. TFA-ON manages the abstract locks and the execution of commit and compensating actions [74]. To provide conflict detection at the abstract level, an abstract locking mechanism is integrated 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 [74].

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 [5] 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.

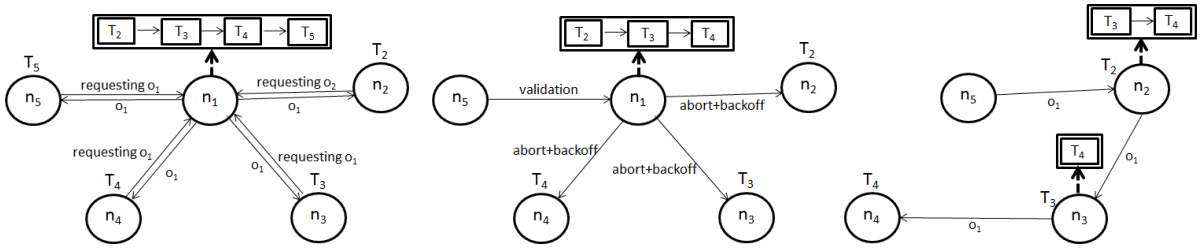
Bi-interval maintains a scheduling queue for read and write transactions for each object. If an enqueued transaction is a read transaction, it is moved to the head of the scheduling queue. If it is a write transaction, it is inserted into the scheduling queue according to the shortest path visiting each 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

```
1  Class Requester {
2     Address address;
3     Transaction_ID txid;
4  }
5  Class Requester_List {
6     List<Requester> Requesters = new LinkedList<Requester>();
7     Integer BackoffTime;
8     void addRequester(backoff_time, Requester);
9     void removeDuplicate(Address);
10 }
11 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

```
1  Procedure Retrieve_Request
2  Input: oid. txid, local_exectuion_time, elapsed_time
3  object = get_Object(oid);
4  address = get_Requester_Address();
5  Integer backoff = 0;
6  if object is not null and in use then
7      Requester_List reqlist = scheduling_List.get(oid);
8      if reqlist is null then
9          reqlist = new Requester_List();
10     else
11         reqlist.removeDuplicate(address);
12     if reqlist.BackoffTime < elapsed_time then
13         backoff = reqlist.BackoffTime;
14         reqlist.addRequest(local_exectuion_time-elapsed_time, new Requester(address, txid));
15         scheduling_List.put(oid, reqlist);
16 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

```
1  Procedure bi_interval
2  Input: scheduling_List, oid
3  Output: Address
4  reqlist = scheduling_List(oid);
5  NextNode = null;dist=∞
6  if reqlist is not null then
7      foreach Requester ∈ reqlist do
8          if Requester is for write transaction then
9              if dist > Distance(Requester.Address) then
10                 dist = Distance(Requester.Address);
11                 NextNode = Requester.Address;
12         else
13             readRequesters.addRequester(Requester);
14 return NextNode;
```

4.5 Analysis

Definition 1. *Given a scheduler A, $makespan_i(A)$ is the time that A needs to complete all the transactions in $\mathbb{T}_N^j$ which require accesses to an object o_j.*

Definition 2. *The competitive ratio (CR) of a scheduler A for $\mathbb{T}_N^j$ is $\frac{makespan_j(A)}{makespan_j(OPT)}$, where OPT is the optimal scheduler.*

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 we define two types of makespans: (1) *traveling makespan*, which is the total communication delay to move an object; and (2) *execution makespan*(or, local execution time γ), which is the time duration of transactions' executions including all aborted transactions, but not communication delays.

Definition 3. *In a given graph G, the object moving cost $\eta_G^A(u, V)$ is the total communication delay for visiting each node from node u holding an object to all nodes in V, under scheduler A.*

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 $\sum_{m=1}^{N-k+1} \gamma_m$.

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

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

Theorem 4.5.2. *Bi-interval's traveling makespan competitive ratio 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., $Stretch_\eta(v_T, V^{R_i}_{T_{N-1}}) = \max \frac{\eta_G(v_T, V^{R_i}_{T_{N-1}})}{d_G(v_T, V^{R_i}_{T_{N-1}})} \leq \frac{1}{2}\log(N-1) + \frac{1}{2}$ from [62]. Hence, $CR^d_{Biinterval} \leq \log(N + I_r - k - 1)$. The theorem follows. □

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

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

$$makespan_{OPT} = \sum_{m=1}^{N-k+1} \gamma_m + \min d_G(v_T, V^{R_i}_{T_{N-k+1}}) \tag{4.1}$$

$$makespan_{Biinterval} = \sum_{m=1}^{N-1} \gamma_m + \log(N-1) \max d_G(v_T, V^{R_i}_{T_{N-1}}) \tag{4.2}$$

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

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

Theorem 4.5.4. *When* $I_r < k$*, Bi-interval improves the traveling makespan* $(\mathrm{makespan}^{\mathrm{d}}_{\mathrm{i}}(\mathrm{Biinterval}))$ *as much as* $O(|\log(1 - (\frac{k-I_r}{N-1})|)$.

Proof.

$$\begin{aligned} \max \quad & \frac{\eta_G(v_T, V^{R_i}_{T_{N+I_r-k-1}})}{d_G(v_T, V^{R_i}_{T_{N-1}})} \\ = \quad & \max\Big(\frac{\eta_G(v_T, V^{R_i}_{T_{N-1}})}{d_G(v_T, V^{R_i}_{T_{N-1}})} + \frac{\varepsilon}{d_G(v_T, V_{T_{N-1}})}\Big) \\ \leq \quad & \frac{1}{2}\log(N-k+I_r-1) + \frac{1}{2} \end{aligned} \tag{4.3}$$

When $I_r < k$, a read interval has at least two read transactions. We are interested in the difference between $\eta_G(v_T, V^{R_i}_{T_{N-1}})$ and $\eta_G(v_T, V^{R_i}_{T_{N+I_r-k-1}})$. Thus, we define ε as the difference between two η_G values.

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

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. □

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.

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

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* $\Theta(\log(n-k))$.

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.

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

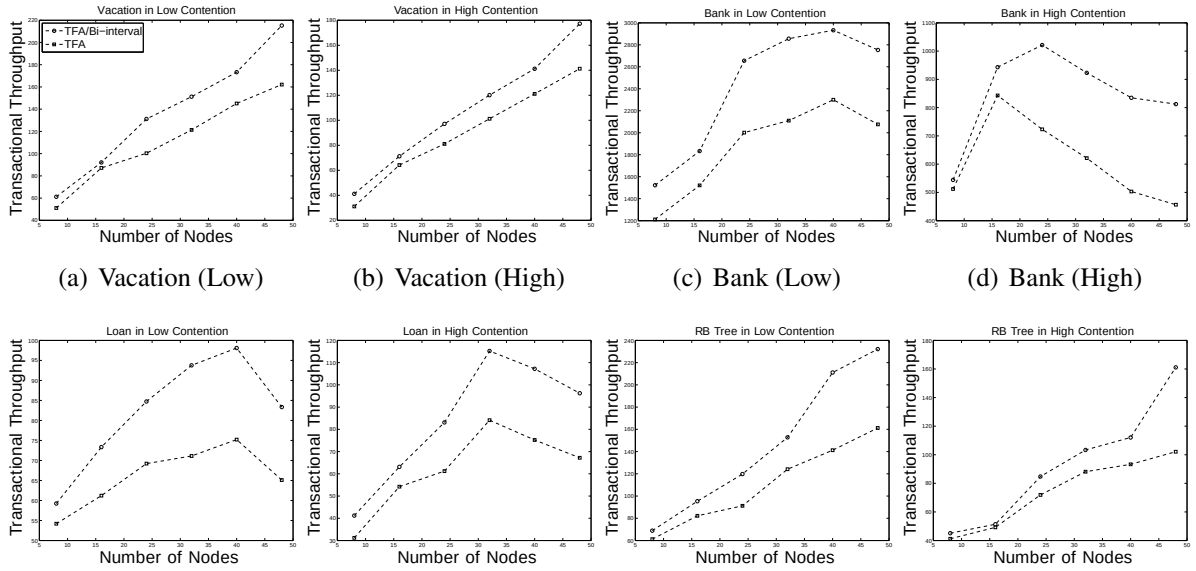
Since $E(I_r)$ is smaller than k, $CR^{Average}_{Biinterval} = \Theta(\log(N-k))$. The theorem follows. □

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 [12], two monetary applications (Bank and Loan). and Red/Black Tree (RB-Tree) [28] 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 of concurrent transactions. This delay also includes waiting time in a scheduling queue and Bi-interval's computational time.



(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

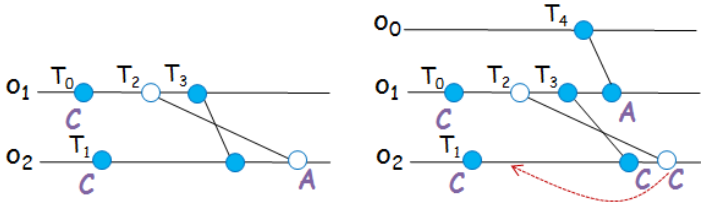
In low contention, Bi-interval produces high concurrency due to the large number of read-only transactions. In high contention, Bi-interval reduces object moving time. In both cases, Bi-interval improve throughput, but concurrency of read-only transactions improves more throughput than reduced object 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 Progressive Transactional Scheduler

5.1 Motivation

Consider the scenario of an SV and MV-STM depicted in Figure 5.1. We use the same style in the figure as that of [59]. The solid circles indicate write operations and the empty circles represent read operations. Transactions are represented on horizontal lines with the circles. The horizontal line corresponding to the status of each object describes the time domain. The letters **C** and **A** indicate commit and abort events, respectively.



(a) T_2 aborts under SV-STM (b) T_2 commits under MV-STM

Figure 5.1: Single vs. Multiple Version Models

In Figure 5.1(a), transaction T_2 reads o_1^0 and transaction T_3 updates objects o_1^1 and o_2^1. Under SV-STM, T_2 reads o_2^1 updated by T_3, which causes T_2 to abort. However, in Figure 5.1(b), T_2 reads o_2^0 updated by T_1, which results in the commit of T_2.

In the scenario depicted in Figure 5.1(b), T_3 and T_4 consist of all write operations. Since T_0 has updated o_1^0 and committed last, a node invoking T_0 becomes the owner of o_1. T_3 and T_4 request o_1

from the node, and then T_3 validates and commits o_1 first, aborting T_4. MV-STM is particularly useful for avoiding aborts of read transactions [55]. However, due to the communication delay among nodes in a distributed system, write transactions (e.g., T_4) may suffer from repeated aborts. For example, when T_3 commits, T_4 aborts due to the conflict over only o_1 and has to request all its objects (i.e., o_0 and o_1) again. When that happens, we cannot guarantee that all objects are accessible for T_4 when it restarts. o_0 may be validated by another transaction, so T_4 may abort due to a different conflict. Even if all objects are available for T_4, it may suffer from communication overhead to request and retrieve o_0 and o_1.

Existing transactional schedulers (reactive or proactive) serialize aborted transactions to avoid repeated aborts. Even though a backoff-based policy is typically used in proactive scheduling, predicting an exact backoff time is difficult due to the communication delays caused by the CC protocol. Also, a random backoff time may delay transactions, because T_4 may restart after T_3 commits, or T_4 may request other objects before o_1 is requested (for example). Under a reactive scheduler, when the conflict between T_3 and T_4 occurs, suppose that T_4 is enqueued and o_1^1 updated by T_3 moves to T_4. We cannot guarantee that T_4 obtains o_1^1. This may cause T_4 to "starve" on o_1^1, as a large number of (write) transactions may be enqueued before T_4 is enqueued.

Motivated by these observations, we identify which object version is subject to the conflict and restart conflicting transactions instead of aborting them. This increases performance because of the following: a transaction may conduct a sequence of operations on multiple objects. Each operation may need one or more objects. Each object may be requested from different object owners. Whenever a conflict is detected, only the corresponding object versions are discarded, and the transaction is not aborted.

5.2 Scheduler Design

5.2.1 Distributed Event-based TM Model

We present an event-based model for detecting conflicts at an object level. The basic idea is to write an event in an event queue whenever it happens. Let n_i denote a node invoking a transaction T_i. We define two types of events: (1) *Request*($\mathrm{Req(n_i, o_j)}$). This event represents the request of object o_j from node n_i. (2) *Acquisition*($\mathrm{Acq(n_i, o_j)}$). This event indicates when node n_i acquires object o_j. When the node that holds the versions of an object receives a request, it first determines which requesting nodes must be sent the versions. Then the node locally records those requesting nodes as having the object versions. After a node receives a version, a transaction (on the node) conducts its read/write operation. We use the notation $E \mid n_i$ to denote the subsequence of events E that contains only events performed by node n_i.

Figure 5.2 shows an example execution scenario of the event model. 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 and T_3 request objects o_1 and o_2 from nodes n_0 and n_1, respectively. Node n_1 holds the list

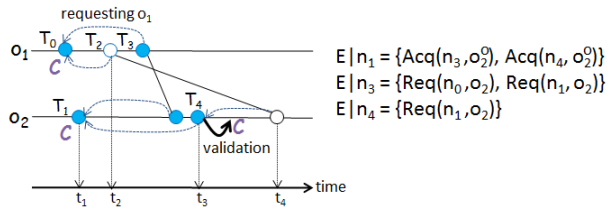


Figure 5.2: Example Scenario of Event-based DTM Model

of versions of o_2. After T_3 requests o_1 from n_1, T_4 requests o_1 from n_1. Thus, n_1 records the events $Acq(n_3, o_2^0)$ and $Acq(n_4, o_2^0)$. When T_4 validates o_2^1 to commit, $Acq(n_4, o_2^0)$ is removed, and T_3 aborts, because $Acq(n_3, o_2^0)$ shows that T_3 has not terminated yet. This means that T_4 commits before T_3 validates o_2. However, in the proposed model, T_3 does not abort and wait until o_2^1 generated by T_3 is available, because o_1^0 still is an up-to-date version. If T_4 commits, node n_4, which invokes T_4 receives the versions o_2^0 and o_2^1 of object o_2. So, T_2 requests o_2 with the value $\mid t_4$ - $t_2 \mid$ from n_4, and n_4 sends o_2^0 updated at time t_1 to T_2, because $\mid t_4 - t_3 \mid < \mid t_4$ - $t_2 \mid < \mid t_4 - t_1 \mid$. Once a write transaction commits, the node invoking it becomes the object owner. The last committed transaction's node holds the list of committed versions.

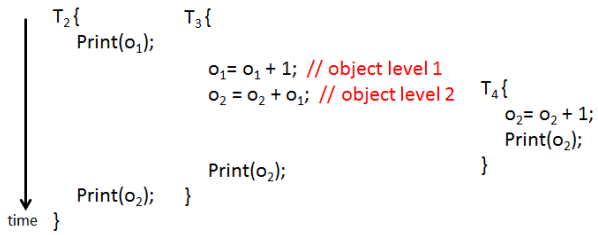


Figure 5.3: Code of Transactions in Figure 5.2

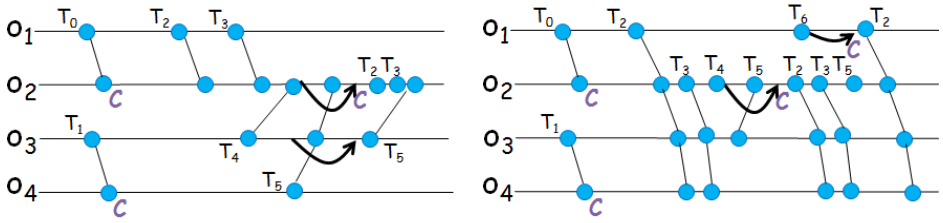
Figure 5.3 shows the code example of transactions T_2, T_3, and T_4 illustrated in Figure 5.2. When T_4 commits, T_3 rolls-back object level 2 and restarts with o_2^1 updated by T_4 (instead of aborting). The event model describes which transactions must be aborted on early validation. There are three cases when a transaction must abort. First, due to limited memory, obsolete object versions are reclaimed by an automatic garbage collection service. We consider a threshold-based garbage collector [11] for this purpose (discussed in Section 5.6). Thus, transactions needing obsolete versions must abort. Second, if a transaction T_i validates an object first, and there are Acq events for that object, then the write transactions that have been involved in those events abort due to early validation. In this case, the transactions may have requested multiple objects and may have already conducted some operations on those objects. So they abort and restart. However, we record events indicating which objects are acquired and requested. Thus, the transactions roll-back the object and restart with a new version updated by T_i, instead of aborting. The third case is aborting transactions that request a version while T_i validates that version. In this case, transactions that

request a read or write operation restart with a new version updated by T_i, instead of aborting. In the second and third cases, PTS is invoked.

There are two purposes for maintaining events. First, if a transaction T_i requesting multiple objects aborts, T_i will invoke the CC protocol to find the location of the objects again, incurring communication delays. An object owner holds a list of events indicating which node acquired which object, and sends a newly updated version to nodes involved in the events. As soon as the versions are available, the transactions that have been involved in the events may receive the versions without the help of the CC protocol, reducing communication delays. Second, if a transaction T_i has conducted some operations on multiple objects and one of those objects has been validated by another transaction, the events indicate the object on which the conflict occurs. So T_i rolls-back the conflicting object. This implies that all operations that T_i has conducted do not need to abort.

5.2.2 Progressive Transactional Scheduling

The event model describes how to maintain events and detect which object version is subject to the conflict. When conflicts are detected, PTS assigns different backoff times to the conflicting transactions and simultaneously sends the requested object versions to nodes invoking those conflicting transactions. The backoff times indicate when the transactions are likely to receive the versions. Thus, a transaction T_i that has validated an object version earlier has a responsibility to simultaneously send that version to nodes requesting that object. This results in concurrency for read transactions and validation of a write transaction with reduced execution time. Unless the transactions receive the updated version before the backoff times, they will request it.



(a) T_4 commits. T_2, T_3, and T_5 are scheduled.

(b) T_4 and T_6 commit. T_2, T_3, and T_5 are scheduled.

Figure 5.4: Examples of Progressive Transactional Scheduling

In Figure 5.4, an arrow represents a time interval that starts when a transaction begins validation to when it commits. A validation in distributed systems includes global registration of version ownership. Figure 5.4 shows two examples illustrating PTS. Figure 5.4(a) indicates a conflict among write transactions T_2, T_3, T_4, and T_5. Transactions T_2 and T_3 request objects o_1 and o_2 from node n_0, respectively. Transaction T_4 requests objects o_2 and o_3 from nodes n_0 and n_1, respectively. Consequently, n_0 records five events: $Acq(n_2, o_1^0)$, $Acq(n_2, o_2^0)$, $Acq(n_3, o_1^0)$, $Acq(n_3, o_2^0)$, $Acq($

n_4, o_2^0). When T_4 starts validation of o_2 and o_3, the event $Acq(n_4, o_2^0)$ is removed from the six events of n_0, and the two events $Acq(n_2, o_2^0)$ and $Acq(n_3, o_2^0)$ are moved to n_4 invoking T_4, because the Acq events indicate that o_2^0 is to be validated. While T_4 validates o_2 and o_3, T_5 requests o_2 from n_0. Also, the request is moved to n_4. Finally, when T_4 ends validation (i.e., commits), n_4 records three events (i.e., $Acq(n_2, o_2^1)$, $Acq(n_3, o_2^1)$, and $Acq(n_5, o_2^1)$) and sends o_2^1 to nodes n_2, n_3, and n_5. In the same way, n_1 holds three events (i.e., $Acq(n_4, o_3^0)$, $Acq(n_5, o_3^0)$, and $Acq(n_5, o_4^0)$). When T_4 commits, n_4 records the event $Acq(n_5, o_3^1)$ and sends o_3^1 to n_5. In the meantime, transactions T_2 and T_3 still acquire up-to-date o_1^0. Thus, T_2 and T_3 roll-back o_2 and restart when o_2^1 is received.

Figure 5.1(b) also shows two conflicts of o_0 and o_1. Let us assume that T_4 validates o_2^0 first. At this time, T_2 and T_3 have completed some operations on objects o_3 and o_4. When T_4 commits, o_3 and o_4 may have a dependency with o_2. Thus, objects o_3 and o_4 that have been requested after o_2 must be rolled-back. Therefore, T_2 starts at the o_2 level, and T_3 aborts, because o_2 is located at the first object level. When T_6 commits, T_2 also aborts.

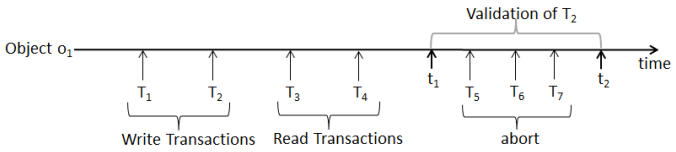


Figure 5.5: An Example of Assigning a Backoff Time

Figure 5.5 shows an example of assigning a backoff time over object o_2 illustrated in Figure 5.1(a). Before T_4 starts validation at time t_1, write transactions T_2 and T_3 acquire o_1. After T_2 starts validation at t_1, T_4 sends backoff messages to nodes n_2 and n_3 for T_2 and T_3, respectively. The backoff messages include $\mid t_2 - t_1 \mid$. T_2 and T_3 roll-back o_2 and wait for $\mid t_2 - t_1 \mid$ time units. Transaction T_5 that requests between times t_1 and t_2 receives a backoff message, which includes $\mid t_2$ – the requesting time of $T_5 \mid$ as the backoff time. T_5 will now wait for the backoff time to expire.

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* [8] representation of the most current successful commit times of write transactions. Whenever a transaction starts, an expected commit time is recorded in the table.

5.3 Illustrative Example

We show an example to implement a write transaction. Listing 5.1 shows a general write transaction including two write operations of a bank benchmark. Each object indicates a bank account. A write operation involves depositing an amount of money to the account. When the *deposit* function generates an exception error, the transaction aborts.

Listing 5.1: An Example of a Write Transaction on TFA

```
Atomic{
  try {
    deposit(object1, amount);
    deposit(object2, amount);
    validate(object1, object2);
  }catch (TransactionException ex){
    commit = false;
  }
}
```

Listing 5.2 shows a write transaction performed as Listing 5.1. The write transaction conducts two deposits. The *deposit* function accepts three parameters: object (i.e., account), amount, and an object level for inputs. When the *deposit* function generates an exception error, PTS updates $object\_level$, which is the object level described in the input. If a backoff message is received at object level 2, the transaction waits for the backoff time. PTS assigns 2 to $object\_level$, and returns an error $deposit(object2, amount, 2)$. The transaction starts at $case\ 2$ of the $switch$ statement.

Listing 5.2: An Example of a Write Transaction on PTS

```
switch(object_level){
  case 1: //object level 1
    try {
      deposit(object1, amount, 1);
    }catch (TransactionException ex){
      commit = false;
      break;
    }
  case 2: //object level 2
    try {
      deposit(object2, amount, 2);
      validate(object1, object2);
    }catch (TransactionException ex){
      commit = false;
      continue;
    }
}
```

5.4 Algorithms

We now present the algorithms for PTS. There are four algorithms: Algorithm 4 for $Open\_Object$, Algorithm 5 for $Retrieve\_Request$, Algorithm 6 for $Commit$, and Algorithm 7 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 $Open\_Object$. The $Commit$ procedure is invoked whenever a transaction successfully terminates. 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, 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 assigned backoff time is not 0, the requester waits for the backoff time. This implies that the requester requests an object, which is being validated by another transaction and does not need to roll back. If the backoff time expires, $Open\_Object$ returns null. Otherwise, the requester wakes up and receives the object. $TransactionQueue$ is used to check the status of live transactions.

Algorithm 4: Algorithm of Open_Object

```
1  Procedure Open_Object
2  Input: Transaction_Type type, Transaction_ID txid, Object_ID oid
3  Output: null, object
4  owner = find_Owner(oid);
5  Send type, oid, and txid to owner;
6  Wait until that Retrieve_Response is invoked;
7  Read object and backoff 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             Record an Req event including owner;
15             return object;
16         else
17             TransactionQueue.remove(txid);
18     return null;
19 else
20     Record an Req event including owner; return object;
```

Algorithm 5 describes $Retrieve\_Request$, which is invoked when an object owner receives a request. $get\_Version$ returns the version of oid corresponding to $txid$. If $get\_Version$ returns null, it is not the owner of oid. Thus, 0 is assigned as the backoff time, and the requester must retry to

find a new owner. If the corresponding version is being validated, $Retrieve\_Request$ has to calculate a backoff time. $expected\_commit\_time$ represents when the transaction that is validating the version commits (approximately). $current\_time$ represents when the object owner receives a request. Thus, $expected\_commit\_time$ - $current\_time$ is determined as the backoff time and sent to the requester.

Algorithm 5: Algorithm of Retrieve_Request

```
1  Procedure Retrieve_Request
2  Input: type, oid. txid
3  version = get_Version(type, oid);
4  address = get_Requester_Address();
5  Integer backoff = 0;
6  if version is not null and type is write then
7      Record an Acq event including version and address;
8      if version is being validated then
9          backoff = expected_commit_time - current_time; version=null;
10 Send version and backoff to address;
```

Algorithm 6 may have multiple objects to commit. After finding owners for each object in Req event lists, Algorithm 6 sends a message to each owner. Owners find who has acquired the objects from the Acq event lists and sends all addresses of nodes acquiring the objects to Algorithm 6. If the addresses are received, an expected commit time is sent to the nodes as the backoff time. The nodes let their transactions know which object has been conflicted. Changes to the ownership for each object occur at validation.

Algorithm 6: Algorithm of Commit

```
1  Procedure Commit
2  Input: objects
3  foreach objects do
4      owner = find_Owner(object) from the Req event list;
5      Send owner a message to obtain the addresses of the nodes acquiring object ;
6      if received addresses is not null then
7          Send expected_commit_time to addresses for a backoff;
8      Validate object;
9      if received addresses is not null then
10         Send object to addresses;
```

In Algorithm 7, $Retrieve\_Response$ sends $Open\_Object$ a signal to wake up if a transaction is waiting for an object. If $txid$ is found in $TransactionQueue$, the transaction corresponding to $txid$ waits for a backoff time. Otherwise, the object is given to the transaction.

Whenever an object is requested, PTS performs Algorithms 4, 5, and 7. We use a hash table to maintain Acq and Req events. The time complexity is $O(the\ number\ of\ events)$ to send an object version.

Algorithm 7: Algorithm of Retrieve_Response

```
1 Procedure Retreive_Response
2 Input: object, txid, 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     Give object to Open_Object;
```

5.5 Properties

We now prove PTS's correctness and progress properties. We consider the opacity correctness criterion [27], which requires that 1) committed transactions must appear to execute sequentially in real-time order, 2) no transaction observes modifications to shared state done by aborted or live transactions, and 3) all transactions, including aborted and live ones, must observe a consistent state.

Strong progressiveness was recently proposed as a progress property [26]. A TM system is strongly progressive if 1) a transaction that encounters no conflict is guaranteed to commit, and 2) if a set of transactions conflicts on a single transactional variable, then at least one of them is guaranteed to commit. Strong progressiveness does not mean the strongest progress property. The strongest progress property mandates that no transaction is ever forcefully aborted, which is impractical to implement due to its high complexity and overhead [26].

Theorem 5.5.1. *PTS ensures opacity.*

Proof. We have to show that PTS satisfies opacity's aforementioned three conditions. We start with the real-time order condition. We say that a transaction T_j reads object o from transaction T_i, if T_i updates o and then commits. Let us assume that the set of transactions $T = \{T_1, T_2, \cdots\}$ successfully commits. If all transactions access a single object, TFA ensures real-time order. We assume that all transactions access a set of objects $O = \{o_1, o_2, \cdots\}$. Assume that T violates real-time order. For this to happen, a transaction $T_i \in T$ commits O and a transaction $T_j \in T$ read O must not abort. However, PTS aborts T_j. If T_j reads a subset of O, T_j rolls-back the subset, resulting in real-time order, which yields a contradiction.

The second condition is guaranteed by the underlying TFA itself, which uses the write-buffer mechanism. This mechanism exposes changes of objects only at commit time.

The last condition ensures consistent state for both live and aborted transactions. If a transaction T_i commits the set of objects, all transactions that have acquired the objects roll-back to ensure consistency. Theorem follows. □

Theorem 5.5.2. *PTS is strongly progressive.*

Proof. Assume, by way of contradiction, that PTS is not strongly progressive. We consider two cases. First, assume that no transaction conflicts with another on an object. This means that none of the transactions have successfully committed. Due to early validation, some transaction must abort or roll-back. Then there must exist a conflicting object version, which yields a contradiction.

Second, assume that there are some conflicts and all transactions fail to commit. Conflicting transactions abort or roll-back. When a new object version is created, conflicting transactions restart, which yields a contradiction. Theorem follows. □

5.6 Evaluation

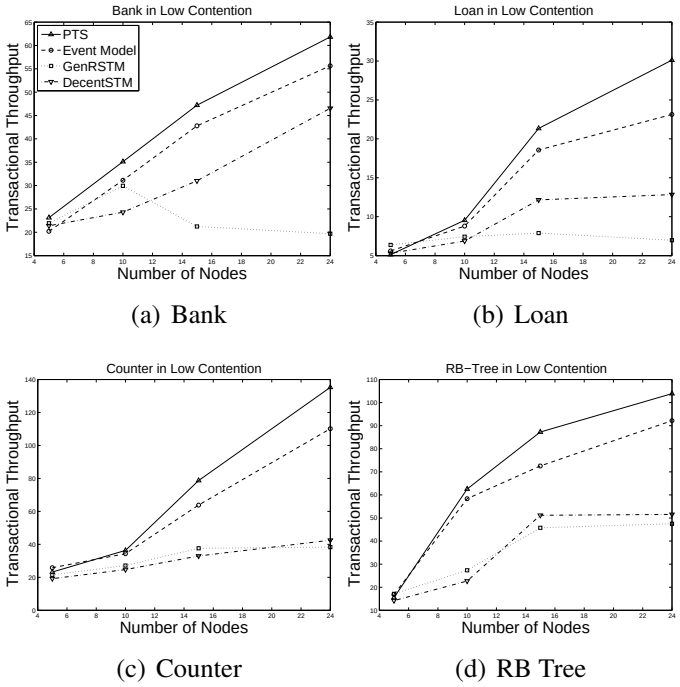
We implemented PTS in the HyFlow DTM framework [63], and developed four benchmarks for experimental studies. The benchmarks include two monetary applications (Bank and Loan), distributed versions of the Vacation of the STAMP benchmark suite [12], and three distributed data structures including Counter, Red/Black Tree (RB-Tree) [28], and Distributed Hash Table (DHT).

We considered two competitor DTM implementations: GenRSTM [14] and DecentSTM [6]. GenRSTM is a generic framework for replicated STMs and uses broadcasting to achieve transactional properties. DecentSTM implements a fully decentralized snapshot algorithm based on multi-version. We compared PTS with just the basic event model (i.e., without PTS), GenRSTM, and DecentSTM.

To manage garbage collection, versions that are no longer accessible need to be identified. Unlike multiprocessors, determining old versions for live transactions in distributed systems incurs communication overheads. Thus, we consider a threshold-based garbage collector [11], which checks the number of versions and disposes the oldest version if the number of versions exceeds a pre-defined threshold. For example, threshold 2 means that at most 2 versions for each object can be kept. We consider threshold 4 for measuring the basic event model's throughput, because we observed that, the speedup is relatively less increased after threshold 4.

Figures 5.6 and 5.7 show the throughput of four benchmarks for PTS, Event Model, GenRSTM, and DecentSTM, under low and high contention, respectively. (Event Model is the basic event model without PTS.) In these experiments, a random number of objects between 1 and 10 is accessed by each transaction. Due to the large number of messages, GenRSTM, and DecentSTM's performance degrades for more than 24 requesting nodes. We observe that PTS yields higher throughput than GenRSTM and DecentSTM.

Figures 5.8 and 5.9 show the throughput of four benchmarks for PTS, Event Model, GenRSTM, and DecentSTM, with 5 objects, under low and high contention, respectively. We observe that PTS performs significantly better than Event Model in Figures 5.8 and 5.9, when compared to Figures 5.6 and 5.7. This is because conflicts between transactions that use a small number of objects and a large number of objects frequently occur. The resulting increase in the abort of large transactions (i.e., those using large number of objects) degrades the performance of throughput.



(a) Bank (b) Loan

(c) Counter (d) RB Tree

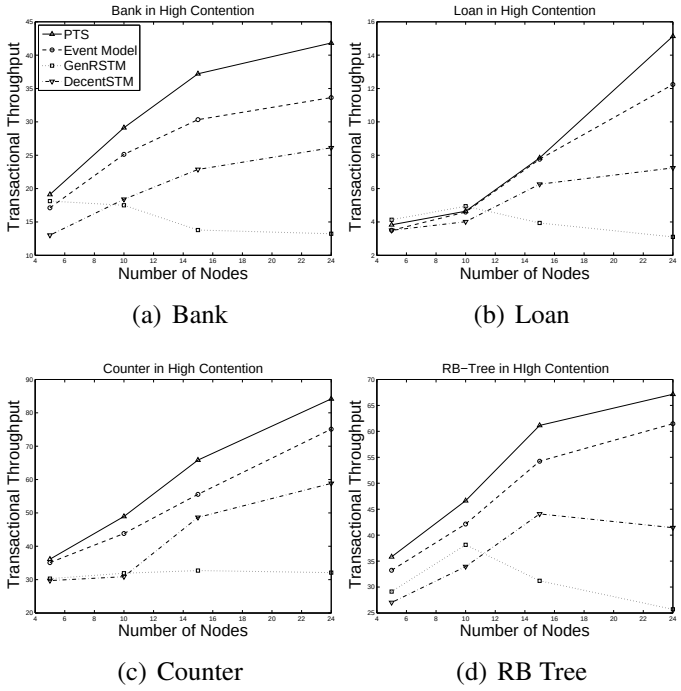
Figure 5.6: Throughput of 4 Benchmarks with 1-10 Random Objects under Low Contention.

Figures 5.10 and 5.11 show the throughput of four benchmarks for PTS, Event Model, GenRSTM, and DecentSTM, with 10 objects, under low and high contention, respectively. A transaction accessing 10 objects has a relatively long execution time. Aborting such transactions degrades throughput. Thus, PTS performs better than GenRSTM and DecentSTM.

Table 5.1: Summary of Throughput Speedup with 5 Objects

	Low Contention		High Contention	
	GenRSTM	DecentSTM	GenRSTM	DecentSTM
Bank	3.2798	2.13501	3.2061	2.6140
Loan	6.3244	3.7468	6.0103	2.5522
Counter	2.4659	2.2166	3.7931	1.9404
RB Tree	4.3471	3.2998	3.2032	1.6282

We computed the throughput speedup of PTS over Event Model, GenRSTM, and DecentSTM – i.e., the ratio of PTS's throughput to the throughput of the respective competitor. Tables 5.1, and 5.2 summarize the speedup of five objects and ten objects, respectively. Our evaluations reveal that PTS improves throughput over GenRSTM and DecentSTM by as much as (average) 3.4× and 3.3× under low and high contention, respectively. We also observe that PTS's throughput on five objects is higher than that on ten objects, but its throughput speedup on ten objects is higher than that on five objects.



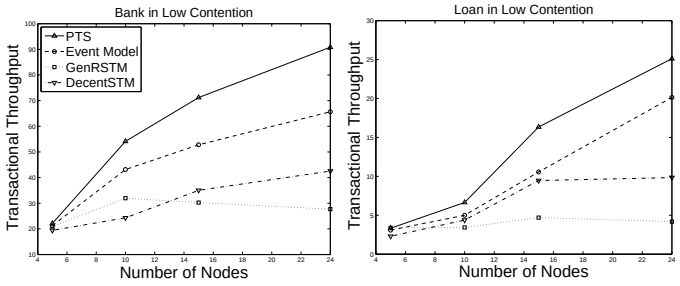
(a) Bank (b) Loan

(c) Counter (d) RB Tree

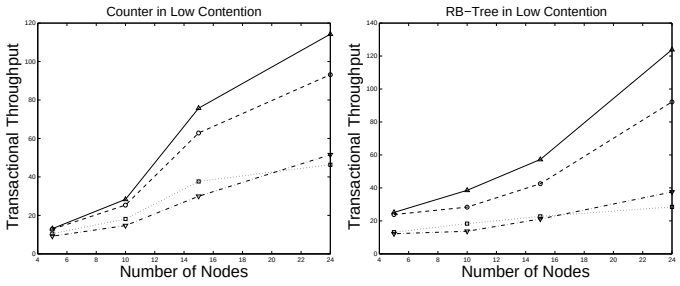
Figure 5.7: Throughput of 4 Benchmarks with 1-10 Random Objects under High Contention.

Table 5.2: Summary of Throughput Speedup with 10 Objects

	Low Contention		High Contention	
	GenRSTM	DecentSTM	GenRSTM	DecentSTM
Bank	3.8580	2.3435	2.884	2.4494
Loan	4.5778	3.3921	6.6436	2.3884
Counter	2.4480	3.6146	4.6852	2.4247
RB Tree	3.7258	3.1778	5.1670	2.2899

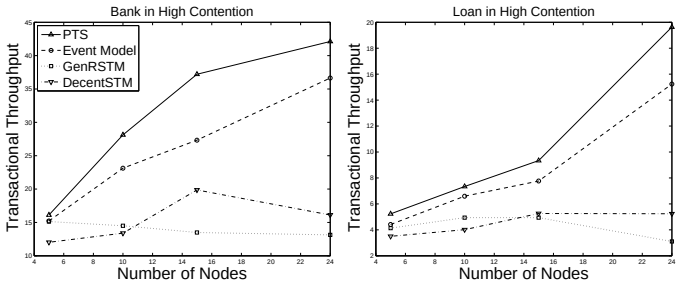


(a) Bank (b) Loan

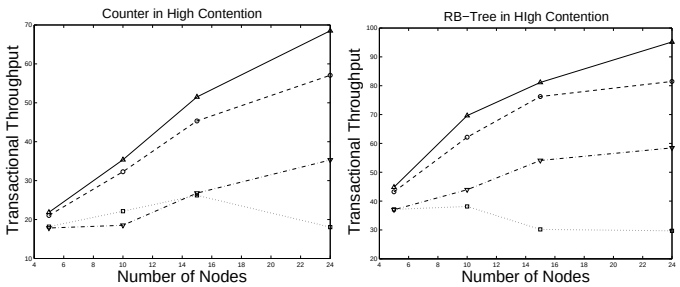


(c) Counter (d) RB Tree

Figure 5.8: Throughput of 4 Benchmarks with 5 Objects under Low Contention.

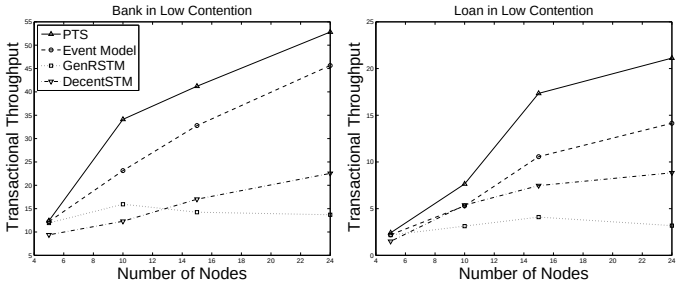


(a) Bank (b) Loan

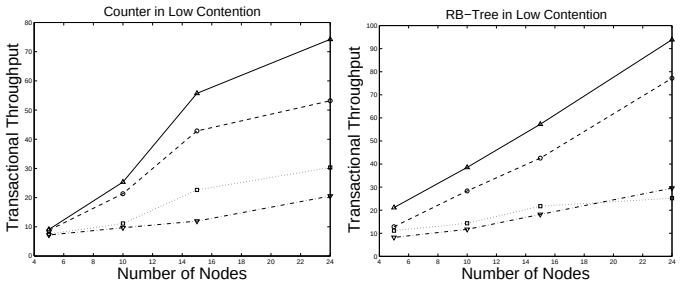


(c) Counter (d) RB Tree

Figure 5.9: Throughput of 4 Benchmarks with 5 Objects under High Contention.

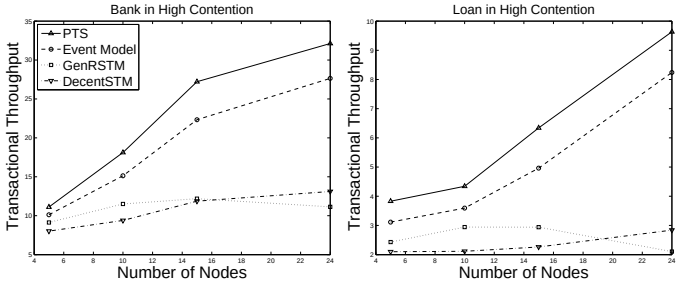


(a) Bank (b) Loan

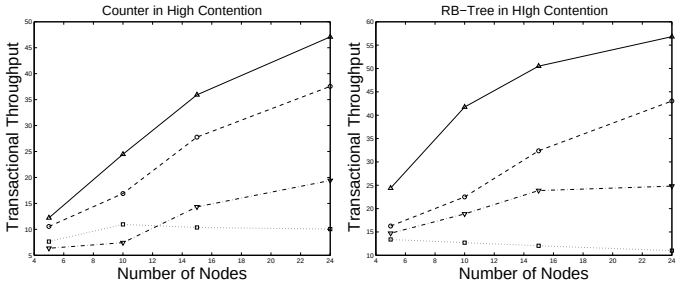


(c) Counter (d) RB Tree

Figure 5.10: Throughput of 4 Benchmarks with 10 Objects under Low Contention.



(a) Bank (b) Loan



(c) Counter (d) RB Tree

Figure 5.11: Throughput of 4 Benchmarks with 10 Objects under High Contention.

Chapter 6

The Cluster-based Transactional Scheduler

6.1 Motivation

There are two cases when transactions are aborted under TFA [63]. First, when a transaction starts validation on an object, transactions that have requested the object but not validated yet are aborted (due to early validation). Second, if transactions request an object while another transaction is being validated on that object, the requesting transactions are aborted to ensure object consistency.

A distributed transaction consumes more execution time, which include the communication delays that are incurred in requesting and acquiring objects than a transaction on multiprocessors. Thus, the probability for conflicts and aborts is higher. In order to reduce the number of aborts, transaction execution time should be minimized. One way to do so is by decreasing communication delays, which depends on object locality: remote access is several orders of magnitude slower than local access.

Directory-based CC protocols (e.g., Arrow, Ballistic, Relay) [18, 35, 78] 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. Even though objects can be replicated to increase locality (and also availability), each node has limited memory. Thus, 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 high memory and 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 memory usage and communication overhead. We develop the CTS scheduler for this model. CTS assigns backoff times to enqueued aborted transactions. Due to the failure of

nodes or links, objects may not be delivered to enqueued transactions. When nodes fail, transactions that they have invoked also fail. When links fail, moving objects may be missing. A backoff time indicates when the enqueued transactions abort and restart.

6.2 Scheduler Design

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-means clustering algorithm [36], to generate k clusters with small intra-cluster distances — i.e., k nodes may hold the same objects.

When a transaction T_1 at node n_1 needs object o_1 for an operation, it sends a request to the object holder belonging to the cluster of n_1. We consider two possible cases in terms of o_1. The first case is when another transaction is validating o_1. In this case, T_1 is enqueued and aborted. The second case is when another transaction may have requested o_1 but no transaction has validated o_1. In this case, the object holder has to determine whether to enqueue T_1. If the operation is read, o_1 is sent to n_1. If the operation is write, T_1 is enqueued because it may be aborted, and o_1 is sent to n_1. If another transaction starts validation of o_1, T_1 must abort. When the transaction ends validation of o_1 (e.g., commit), a new copy of o_1 is sent to all the object holders of o_1 in other clusters to ensure one-copy serializability. Also o_1 is sent to n_1. When n_1 receives o_1, T_1 immediately restarts. If T_1 commits, n_1 becomes a new object holder of o_1 in the cluster belonging to n_1.

Object owners for each cluster maintain their own scheduling queue. When T_1 validates o_1, the object owner of a cluster belonging to T_1 sends the list of enqueued aborted transactions to n_1. o_1 updated by T_1 is simultaneously sent to the enqueued aborted transactions. In the meantime, object owners of o_1 in other clusters may hold different enqueued aborted transactions. Once the object owners receive o_1 from n_1, o_1 is also simultaneously sent to the different enqueued aborted transactions.

The main purpose of enqueuing aborted transactions is to decrease the execution time of a transaction. We consider two effects through the enqueuing strategy. First, if a transaction aborts, the CC protocol will be invoked to find the location of the objects again, incurring communication delays. An object owner holds a queue consisting of the enqueued aborted transactions, and sends a newly updated object to the nodes invoking the transactions. As soon as validation of the object completes, the enqueued aborted transactions may receive the object without the help of the CC protocol, reducing communication delays.

Second, when validation of an object completes, the object is simultaneously sent to all enqueued aborted transactions if they exist. This results in concurrency for read transactions and validation of a write transaction with reduced execution time. If one of the enqueued aborted write transactions starts validation first, the transaction will have the smallest execution time of the enqueued aborted

write transactions in all clusters. Its intent is to give a transaction with the smallest execution time a chance to be committed first, reducing the probability of conflicts.

In the meantime, CTS assigns different backoff times for transactions that will be enqueued. The backoff times indicate when the enqueued aborted transactions receive an object. Due to potential failures of nodes or links, moving objects may be missing or object owners may loose the enqueued aborted transactions. Thus, the enqueued aborted transactions start unless they receive the object until the assigned backoff times. Due to node failure, if a transaction does not receive any response, it requests an object from another object holder in the closest neighbor cluster over a network.

6.3 Illustrative Example

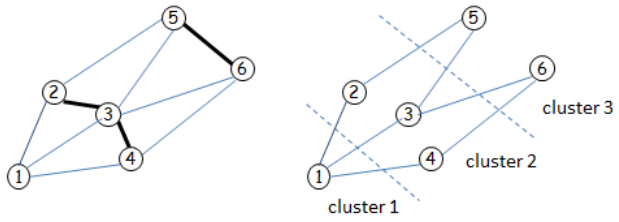


Figure 6.1: An Example of 3-Clustering Algorithm

Figure 6.1 shows an example of 3-clustering algorithm on a six-node network. Each object may be held by different nodes. For example, node 1 holds all objects because cluster 1 consists of only node 1. The length of the link between nodes indicates the communication delay. To construct 3 clusters, the three smallest links indicated by the three thick lines have been removed.

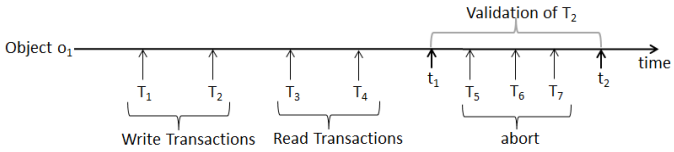
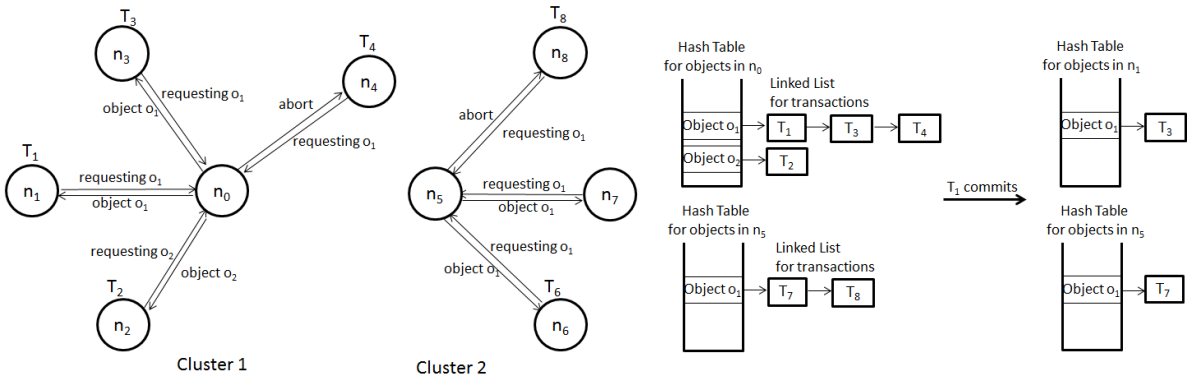


Figure 6.2: An Example of Assigning a Backoff Time

Figure 6.2 illustrates an example of assigning a backoff time over object o_1. Before T_2 starts validation at t_1, write transactions T_1 and T_2 receive o_1 and are enqueued. After T_2 starts validation at t_1, T_1 aborts, and the requesting transactions T_5, T_6, and T_7 between t_1 and t_2 are aborted and enqueued. All aborted transactions T_1, T_5, T_6, and T_7 should start at t_2. Backoff times of $\mid t_2 - t_1 \mid$ is assigned to T_1, and $\mid t_2$ - the aborted time of $T_5 \mid$ is assigned to T_5. A validation in distributed systems includes global registration of object ownership over all clusters. The commit time t_2 can

be predicted using average running times when validation starts at t_1. We discuss how to expect t_2 in Section 6.4.



(a) 2 Clusters with 9 Nodes and 7 Transactions

(b) States of Scheduling Queues at Nodes n_0, n_1, and n_5

Figure 6.3: An Example of CTS

Figure 6.3 illustrates an example of CTS. Figure 6.3(a) shows the architecture of two clusters consisting of 9 nodes. Assume that write transactions T_1, T_2, T_3 and read transaction T_4 request object o_1 from node n_0, and read transactions T_6 and T_8 and write transaction T_7 request o_1 from n_5. While T_1 validates o_1, T_4 and T_8 request o_1 from n_0 and n_5, respectively. So T_4 and T_8 abort. Write transactions T_3 and T_7 abort due to the validation of T_1.

Figure 6.3(b) shows the queue states of nodes n_0, n_1, and n_5. If T_1 commits, n_1 becomes the object holder of o_1 in cluster 1 and n_0 sends the enqueued aborted transactions for o_1 to n_1. Also, n_1 sends o_1 updated by T_1 to n_5 in cluster 2. All enqueued aborted transactions receive o_1 from n_1 and n_5. The enqueued read transactions T_4 and T_8 will be dequeued from the scheduling queues of n_1 and n_5. The enqueued write transactions T_3 and T_7 will remain in the scheduling queues until they commit.

6.4 Algorithms

We now present the algorithms for CTS. There are four algorithms: Algorithm 8 for $Open\_Object$, Algorithm 9 for $Retrieve\_Request$, Algorithm 10 for $Commit$, and Algorithm 11 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 $Open\_Object$. The $Commit$ procedure is invoked whenever a transaction successfully terminates. Finally, $Retrieve\_Response$ is invoked whenever the requester receives a response.

Algorithm 8 describes the procedure of $Open\_Object$. $Open\_ Object$ is invoked when a transaction needs an object. If the transaction restarts due to expired backoff timer, $Open\_Object$ finds the object owner of oid again. If the transaction restarts due to a wake-up signal from the object owner, it simply reads the object from its local cache.

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 assigned backoff time is not 0, the requester saves the backoff time, so xid aborts and wait 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 8: Algorithm of Open_Object

```
1  Procedure Open_Object
2  Input: Transaction_Type type, Transaction_ID txid, Object_ID oid
3  Output: null, object
4  if there is object corresponding to oid then
5      return object;
6  owner = find_Owner(oid);
7  Send type, oid, and txid to owner;
8  Wait until that Retrieve_Response is invoked;
9  Read object and backoff from Retrieve_Response;
10 if object is null and backoff is not 0 then
11     Set backoff; return null;
12 else
13     if txid is already aborted then
14         return null;
15     else
16         return object;
```

The data structures depicted in Algorithm 1 are also used in Algorithms 9 and 11. Algorithm 9 describes $Retrieve\_Request$, which is invoked when an object owner receives a request. $Retrieve\_Request$ enqueues the request and sends the object to the requester. As soon as another transaction starts validation of the object, an abort message with a backoff time is sent to all enqueued transactions. If the object is being validated, the requester is enqueued, and a backoff time is assigned.

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* [8] representation of the most current successful commit times of write transactions. Whenever a transaction starts, an $expected\_commit\_time$ is recorded in the table. If the requested object is being validated, a backoff time is computed as $expected\_com$-mit_time – requesting time (e.g, $current\_time$).

Anticipating an exact commit time is too optimistic. Transactions that have been enqueued may not

receive any object due to node or link failures. Also, transactions may not obtain any objects within their backoff times due to miss-prediction of commit times. In this case, transactions request the objects again. Thus, the $removeDuplicate(address)$ function removes redundant requests from the scheduling queue.

Algorithm 9: Algorithm of Retrieve_Request

```
1  Procedure Retrieve_Request
2  Input: type, oid. txid
3  object = get_Object(oid);
4  address = get_Requester_Address();
5  Integer backoff = 0;
6  if object is not null then
7      Requester_List reqlist = scheduling_List.get(oid);
8      if reqlist is null then
9          reqlist = new Requester_List();
10     else
11         reqlist.removeDuplicate(address);
12     if object is being validated then
13         backoff = expected_commit_time - current_time; object=null;
14     reqlist.addReqeuster(backoff, new Requester(address, txid));
15     scheduling_List.put(oid, reqlist);
16 Send object and backoff to address;
```

Algorithm 10 accepts multiple objects as inputs. If another transaction starts validation first, $status$ of enqueued transactions will be updated to $abort$. Algorithm 10 checks $status$ first and validates each object. Since each cluster has an owner for each object, Algorithm 10 sends the objects to each owner.

Algorithm 10: Algorithm of Commit

```
1  Procedure Commit
2  Input: objects
3  Output: rollback, or commit
4  if status == abort then
5      return rollback;
6  foreach objects do
7      owner = find_Owner(object);
8      Request enqueued aborted transactions from owner;
9      Send an abort message with expected_commit_time to the enqueued aborted transactions;
10     Validate object;
11     Send object to all object owners of other clusters;
12     return commit;
```

In Algorithm 11, $Retrieve\_Response$ is invoked when the object that $Object\_Open$ has requested is received. $Object\_Open$ wakes up and reads $object$ and $backoff$. $Retrieve\_Response$ is also

invoked if another transaction starts or ends the validation of $object$. When the transaction starts validation, $Retrieve\_Response$ receives an abort message with a backoff time. When the transaction ends validation, $Retrieve\_Response$ receives only $object$.

Algorithm 11: Algorithm of Retrieve_Response

```
1 Procedure Retreive_Response
2 Input: object, txid, backoff, scheduling_List
3 if txid is waiting in Object_Open then
4     Send a signal to wake up and give object and backoff;
5 else if txid is ready to restart then
6     Send a signal to restart and give object;
7 else if txid is working then
8     Set backoff and abort txid;
```

Whenever an object is requested, Algorithms 8, 9, and 11 are invoked. We use a hash table for objects and a linked list for transactions. The time complexity is $O(1)$ to enqueue a transaction. To check for duplicated transactions in all enqueued transactions, the time complexity for an object is $O(the\ number\ of\ enqueued\ transactions)$.

6.5 Analysis

We now show that CTS outperforms another scheduler in speed. Recall that CTS uses TFA to guarantee a consistent view of shared objects between distributed transactions, and ensure atomicity for object operations. In [63], TFA is shown to exhibit opacity (i.e., its correctness property) [27] and strong progressiveness (i.e., its progress property [26]). Each cluster maintains the same copy of objects and guarantees TFA's properties. Thus, CTS for each cluster ensures opacity and strong progressiveness. For the purpose of analysis, we consider a symmetric network of N nodes scattered in a metric space. We consider three different models: no replication (NR), partial replication (PR), and full replication (FR) in cc DTM.

Definition 4. *Given a scheduler A and N transactions in D-STM, $makespan_A^N(Model)$ is the time that A needs to complete N transactions on $Model$.*

Definition 5. *The relative competitive ratio (RCR) of schedulers A and B for N transactions on $Model$ in D-STM is $\frac{makespan_A^N(Model)}{makespan_B^N(Model)}$. Also, the relative competitive ratio (RCR) of model 1 and 2 for N transactions on scheduler A in D-STM is $\frac{makespan_A^N(Model1)}{makespan_A^N(Model2)}$.*

Given schedulers A and B for N transactions, if RCR (i.e., $\frac{makespan_A^N(Model)}{makespan_B(Model)}$) < 1, A outperforms B. Thus, RCR of A and B indicates a relative improvement between schedulers A and B

if $makespan_A^N(Model) < makespan_B^N(Model)$. 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 lemmas.

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 η_i transactions in the queue. T_i can only commit after η_i transactions commit if η_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. □

Lemma 6.5.2. *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 $2 \times d(n_i, n_j)$. 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 6.5.3. *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 6.5.4. *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 the requesting times for the objects do not occur in FR. Even though the transactions abort, FR does not incur communication delays. The basic idea of transactional schedulers is to minimize conflicts, reducing object requesting times. In FR, the transactional schedulers do not affect $makespan$. Thus, when a transaction commits, FR takes $\sum_{i=1}^{N-1}\sum_{j=1}^{N-1} d(n_i, n_j)$ for atomic broadcasting to support one-copy serializability. □

Theorem 6.5.5. *Given scheduler B and N transactions, $makespan_B^N(FR) \leq makespan_B^N(PR) \leq makspan_B^N\ (NR)$.*

Proof. Given k PR, $\lim_{k \to 1} makespan_B^N(PR) \leq 2(N-1)\sum_{i=1}^{N-1} d(n_i, n_j) + \Gamma_N$, 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 6.5.6. *Given N transactions and M objects, the RCR of schedulers CTS on PR and 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 schedulers CTS on PR and FR is less than 1, $makespan^N_{CTS}(PR) < makspan^N_B(FR)$. $makespan^N_{CTS}(PR) \leq 2\delta_{N-k} + (N-1)\delta_{k-1} + \Gamma_N$, because CTS does not abort the aborted transactions again. $2\delta_{N-k} + (N-1)\delta_{k-1} \leq (N-1)\delta_{N-1}$, so that $2\delta_{N-k} \leq (N-1)\delta_{N-k}$. Only when $N \geq 3$, PR is feasible. Hence, $makespan^N_{CTS}(PR) < makspan^N_B(FR)$, where $N > 3$. The theorem follows. □

Theorem 6.5.7. *CTS ensures one-copy serializability.*

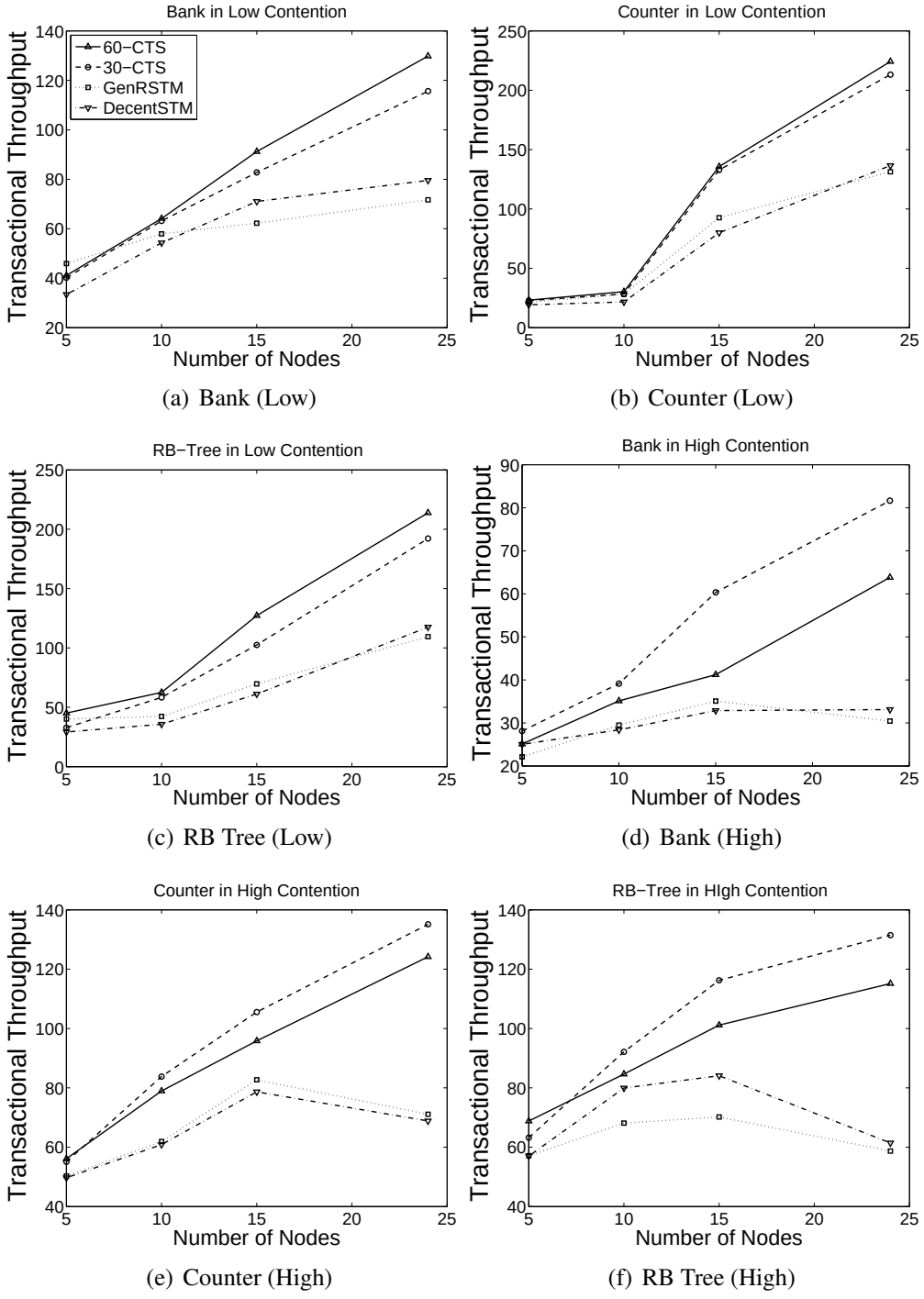
Proof. Consider the set of transactions T=$\{T_1, T_2, \cdots T_N\}$ requesting the object o from the object owners in each cluster at time t. We consider three cases. First, if any transaction $T_i \notin T$ validates o before t, all transactions in T obtain the same copy of o updated by T_i. Second, if all transactions in T except T_j start validation of o first after t, all transactions in T except T_j abort. Note that if any transaction starts validation of o, the status of the object owners of o located in all clusters will be updated, and all transactions that have requested o will abort. All transactions in T except T_j restart and obtain the same copy of o after T_j commits. Finally, if $T_i \notin T$ validates o at t, all transactions in T abort. All transactions in T restart and obtain the same copy of o updated by T_i. The theorem follows. □

6.6 Evaluation

We implemented CTS in the HyFlow DTM framework [63], and developed four benchmarks for experimental studies. The benchmarks include two monetary applications (Bank and Loan) and two distributed data structures including Counter and Red/Black Tree (RB-Tree) [28] as microbenchmarks. In the Bank benchmark, accounts are distributed over nodes (which represent bank branches), and every node invokes concurrent transactions on either balance-checking or money transfer operations. Loan is a simple money transfer application, in which a set of monetary asset holders is distributed over nodes.

We considered two competitor (cluster) DTM implementations: GenRSTM [14] and DecentSTM [6]. 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. We considered 30-CTS and 60-CTS, meaning CTS over 30% and 60% object owners of the total nodes, respectively. For instance, 30-CTS under 10 nodes means CTS over 3-clustering algorithm.

Figure 6.4 shows the throughput of four benchmarks for 30-CTS, 60-CTS, GenRSTM, and DecentSTM with 20% node failure under low and high contention, respectively. In these experiments, 20% of nodes randomly fail. GenRSTM and DecentSTM maintain replicated data for each node,



(a) Bank (Low) (b) Counter (Low)

(c) RB Tree (Low) (d) Bank (High)

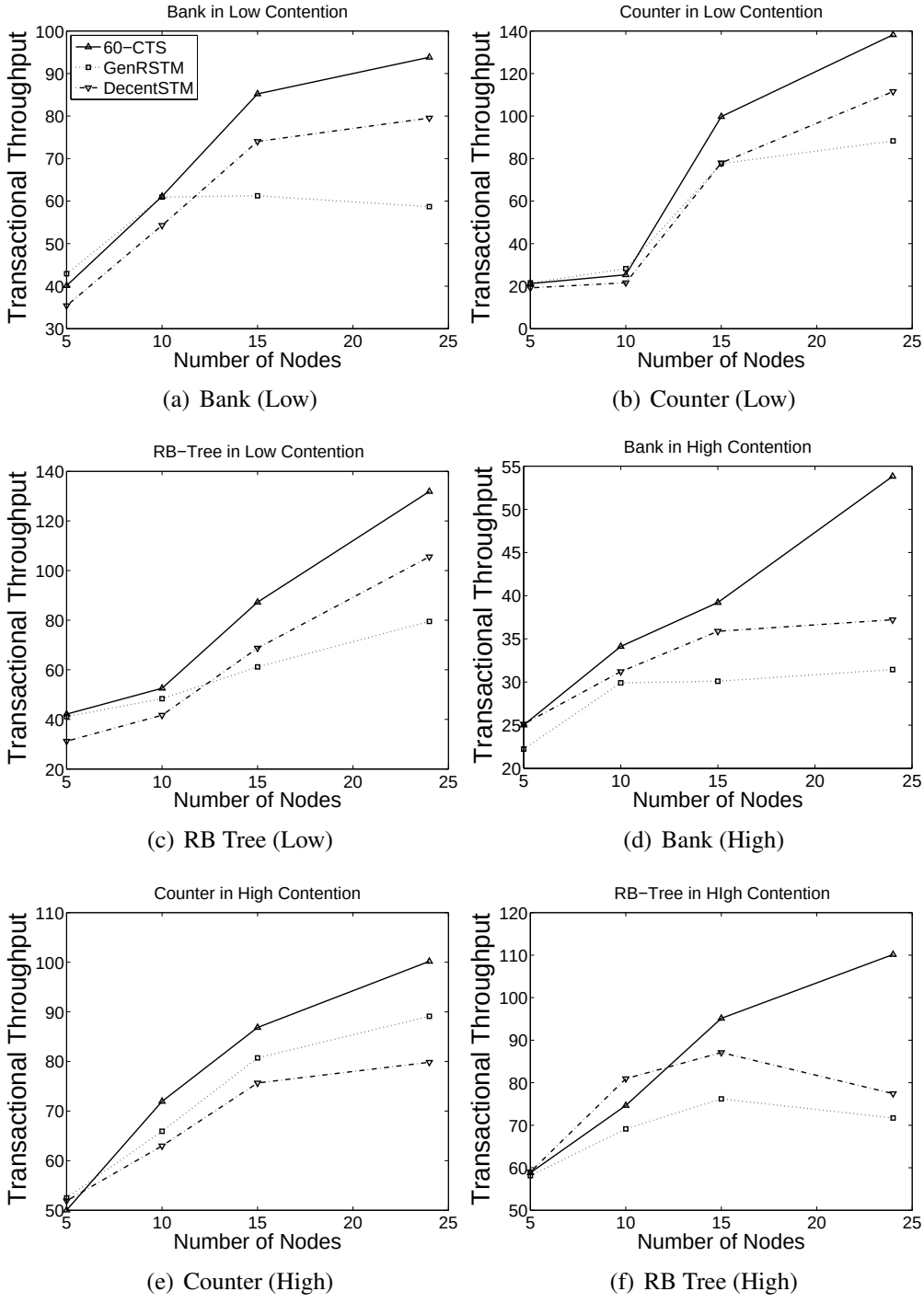
(e) Counter (High) (f) RB Tree (High)

Figure 6.4: Throughput of 4 Benchmarks with 20% Node Failure under Low and High Contention (5 to 24 nodes).

so their throughput decreases as the number of nodes increases. Due to the large number of messages, their performance degrades for more than 24 requesting nodes. We observe that CTS yields higher throughput than GenRSTM and DecentSTM. In particular, 60% of nodes are entitled to the ownership of an object based on 60-CTS. 60-CTS maintains smaller clusters than 30-CTS, so the communication delays to request and retrieve objects decrease. Thus, even though the number of messages increases in 60-CTS, 60-CTS yields higher throughput than 30-CTS under low contention. Under high contention, 60-CTS suffers from large number of messages, degrading its throughput.

Figure 6.5 shows the throughput of four benchmarks for 60-CTS, 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 60-CTS over GenR STM and DecentSTM – i.e., the ratio of CTS's throughput to the throughput of the respective competitor. Tables 6.6(a) and 6.6(b) summarize the throughput speedup under 20% and 50% node failure, respectively. Our evaluations reveal that 60-CTS 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.

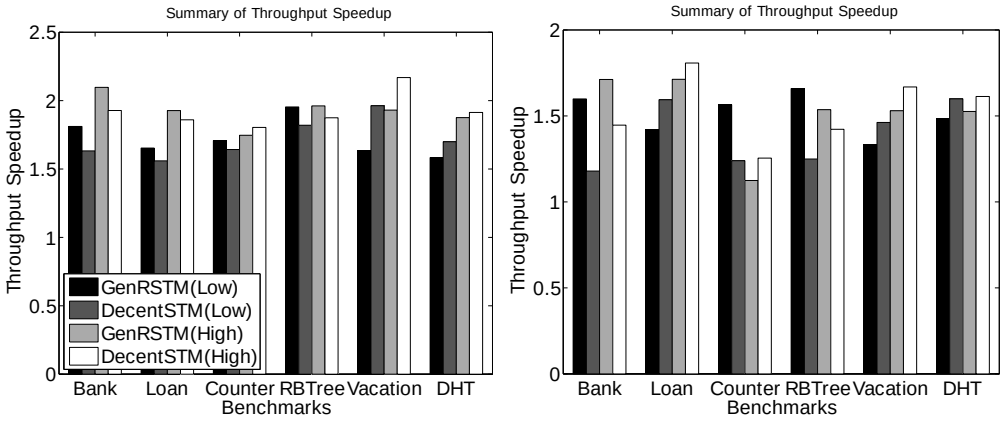


(a) Bank (Low) (b) Counter (Low)

(c) RB Tree (Low) (d) Bank (High)

(e) Counter (High) (f) RB Tree (High)

Figure 6.5: Throughput of 4 Benchmarks with 50% Node Failure under Low and High Contention (5 to 24 nodes).



(a) Summary of Throughput Speedup with 20% Node Failure

(b) Summary of Throughput Speedup with 50% Node Failure

Figure 6.6: Summary of Throughput Speedup

Chapter 7

The Reactive Transactional Scheduler

7.1 Motivation

Past transactional scheduler often causes only small number of aborts and reduces the total communication delay in DTM [38]. 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 [77, 7]. 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.

7.2 Scheduler Design

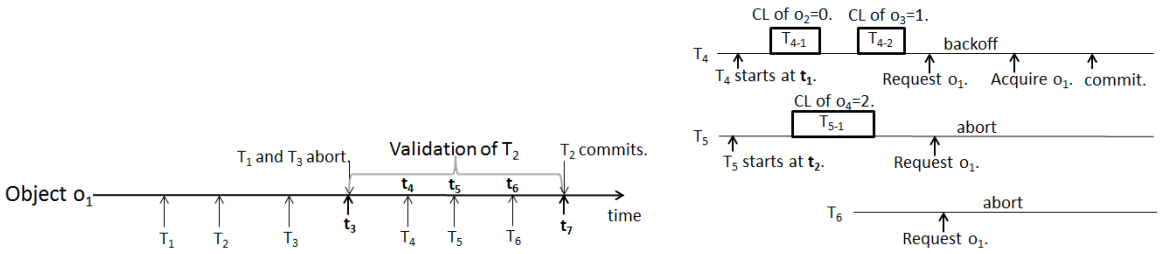
We consider two kinds of aborts that can occur in closed-nested transactions when a conflict occurs: aborts of nested transactions and aborts of parent transactions. Closed nesting allows a nested transaction to abort without aborting its parent transaction. If a parent transaction aborts however, all 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 7.3.

7.3 Illustrative Example

RTS assigns different backoff times for each enqueued transaction. A backoff time is computed as a percentage of estimated execution time. Figure 7.1 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 7.1: 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 7.1 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* [8] 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 7.1, 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.

7.4 Algorithms

We now present the algorithms for RTS. There are three algorithms: Algorithm 12 for $Open\_Object$, Algorithm 13 for $Retrieve\_Request$, and Algorithm 14 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 12 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 12: 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 13 and 14. Algorithm 13

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 13: Algorithm of Retrieve_Request

```
1  Procedure Retrieve_Request
2  Input: oid. txid, Contention_Level, ETS
3  object = get_Object(oid);
4  address = get_Requester_Address();
5  Integer backoff = 0;
6  if object is not null and in use then
7      Requester_List reqlist = scheduling_List.get(oid);
8      if reqlist is null then
9          reqlist = new Requester_List();
10     else
11         reqlist.removeDuplicate(address);
12     if bk < | ETS.r - ETS.s | then
13         Integer contention = reqlist.getContention()+Contention_Level;
14         if contention < CL_Threshold then
15             bk += | ETS.c - ETS.r |; backoff = bk;
16             reqlist.addReqeuster(contention, new Requester(address, txid));
17             scheduling_List.put(oid, reqlist);
18 Send object and backoff to address;
```

In Algorithm 14, $Retrieve\_Response$ 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 12, 13, and 14. 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 14: 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;
```

7.5 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 [63], TFA is shown to exhibit opacity (i.e., its correctness property) [27] and strong progressiveness (i.e., its progress property [26]). 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 i and j. Transactions T_i and T_j are invoked at nodes n_i and n_j, respectively. The local execution time of T_i is defined as γ_i.

Definition 6. *Given a scheduler A and N transactions in D-STM, $makespan_A(N)$ is the time that A needs to complete N transactions.*

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

Definition 7. *The relative competitive ratio (RCR) of schedulers A and B for N transactions in D-STM is $\frac{makespan_A(N)}{makespan_B(N)}$.*

Given schedulers A and B for N transactions, if RCR (i.e., $\frac{makespan_A(N)}{makespan_B(N)}$) < 1, A outperforms B. Thus, RCR of A and B indicates a relative improvement between schedulers A and B if $makespan_A(N) < makespan_B(N)$. 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 7.5.1. *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 7.5.2. *Given scheduler B and N transactions, $makespan_B(N) \leq 2(N-1)\sum_{i=1}^{N} d(n_0, n_i)+ \sum_{i=1}^{N} \gamma_i$.*

Proof. Lemma 7.5.1 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 7.5.3. *Given scheduler RTS and N transactions, $makespan_{RTS}(N) \leq \sum_{i=1}^{N} d(n_0, n_i) + \sum_{i=1}^{N} d(n_{i-1}, n_i) + \sum_{i=1}^{N} \gamma_i$.*

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 7.5.4. *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. $makespan_{RTS}(N) < makespan_B(N)$ 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, $\frac{\sum_{i=1}^{N} d(n_{i-1}, n_i)}{\sum_{i=1}^{N} d(n_0, n_i)} < \log N$ [62]. Hence, $M \times \log N < M \times (2N-3)$, when $N \geq 2$. The theorem follows. □

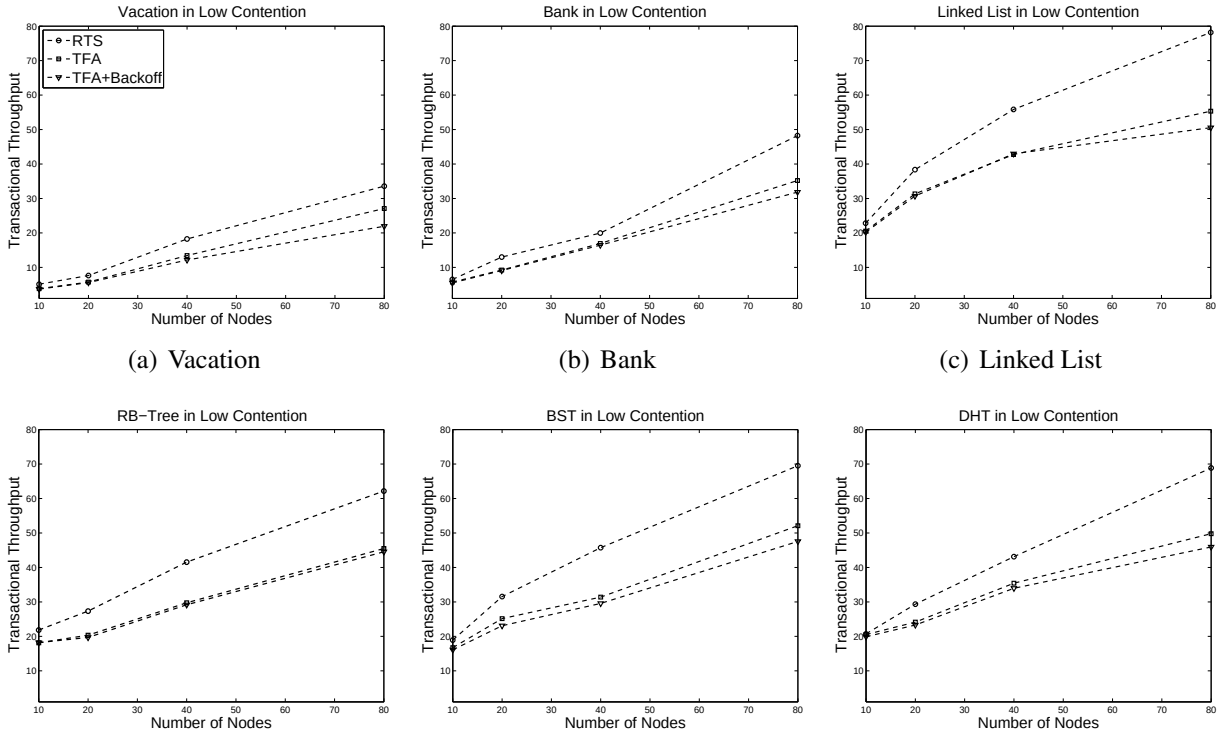
7.6 Evaluation

We implemented RTS in the HyFlow DTM framework [63] 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 [12], Bank as a monetary application [63], and four distributed data structures including Linked-List (LL), Binary-Search Tree (BST), Red/Black Tree (RB-Tree), and Distributed Hash Table (DHT) [28] 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 [20]. 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 [73]. 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.



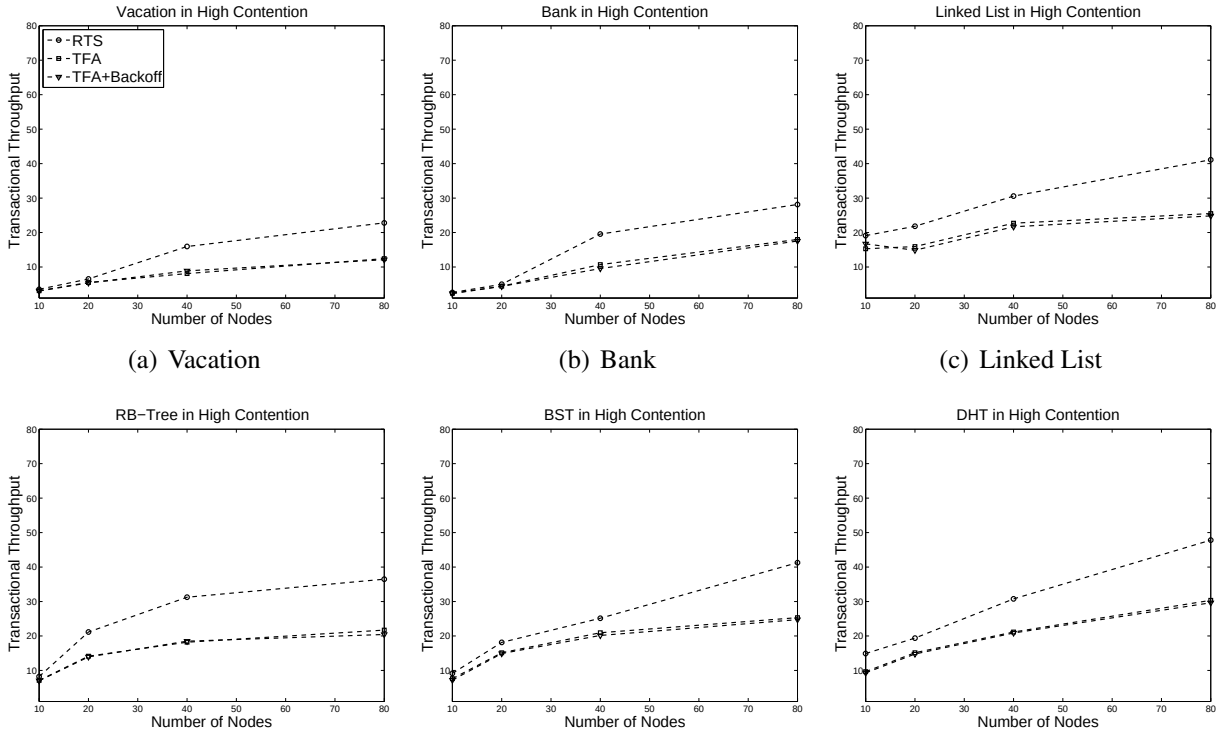
(a) Vacation (b) Bank (c) Linked List

(d) RB-Tree (e) BST (f) DHT

Figure 7.2: Transactional Throughput on Low Contention

Figure 7.2 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 7.2, 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 7.2(a) and 7.2(b), we observe that Vacation and Bank benchmarks take longer execution time than others. The improvement of their throughput is less pronounced.



(a) Vacation (b) Bank (c) Linked List

(d) RB-Tree (e) BST (f) DHT

Figure 7.3: Transactional Throughput on High Contention

Figure 7.3 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 7.3(c), 7.3(d), 7.3(e), and 7.3(f), the throughput is better than that of Bank and Vacation, because LL, RB Tree, BST, and DHT have relatively short local 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 7.4 summarizes the speedup. Our experimental evaluations reveal that RTS improves throughput over DTM without RTS by as much

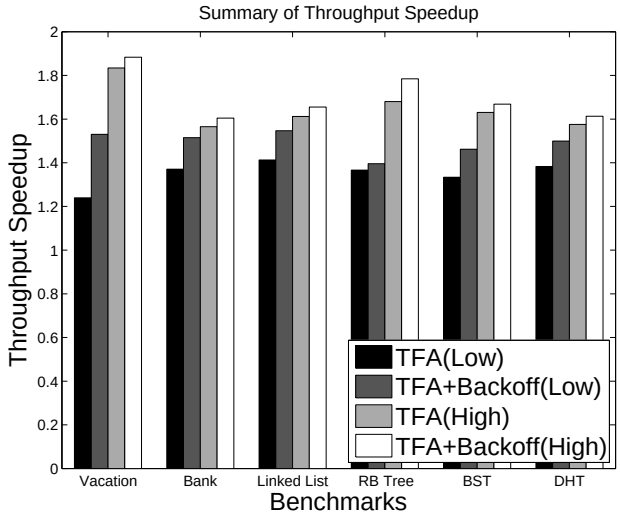


Figure 7.4: Summary of Throughput Speedup

as 1.53 (53%) $\sim$ 1.88 (88%) $\times$ speedup in low and high contention, respectively.

Chapter 8

The Dependency-Aware Transactional Scheduler

8.1 Motivation

```
new Atomic<Boolean >(NestingModel.OPEN){
   @Override boolean atomically ( Txn t ) {
      BST bst = (BST) t.open( "tree-2" );
      int value = bst .remove(key , t);
      boolean inserted = insert(Txn t, vlaue);
      return inserted;
   }
}.execute( );
```

Transaction T_1

```
new Atomic<Boolean >(NestingModel.OPEN){
   @Override boolean atomically ( Txn t ) {
      BST bst = (BST) t.open( "tree-2" );
      bst .delete(7 , t);
      boolean inserted=insert(Txn t, 7);
      return inserted;
   }
}.execute( );
```

Transaction T_2

(a) Two Outer Transactions

```
public boolean insert( Txn t, int value){
private boolean inserted = false;
   @Override boolean atomically ( Txn t ) {
      BST bst = (BST) t.open( "tree-1" );
      inserted = bst .insert(vlaue , t);
      t.acquireAbstractLock (bst , value);
      return inserted ;
   }
   @Override onAbort( Txn t ) {
      BST bst = (BST) t.open ( "tree-1" );
      if (inserted ) bst.delete (value, t);
      t.releaseAbstractLock (bst, value);
   }
   @Override onCommit ( Txn t ) {
      BST bst = (BST) t.open ( "tree-1" );
      t.releaseAbsractLock (bst , value);
   }
}
```

(b) Inner Transaction

Figure 8.1: Two scenarios with abstract locks and compensating actions.

Figure 8.1 shows two examples of open-nested transactions with compensating actions and abstract locks (Codes performing the actual $insert$, $remove$, and $delete$ functions are not shown). Figure 8.1(a) illustrates two outer transactions, T_1 and T_2 including an inner transaction indicated in Figure 8.1(b). The inner transaction includes an *insert* operation. T_1 has a *remove* operation, which returns a removed value indicated by the *key*. The value is used as an input of the inner transaction. Commit and compensating actions are registered when the inner transaction commits. To acquire or release the lock, a message is sent to the owner of $tree$-1. When the inner transaction commits, its modification becomes immediately visible for other transactions. T_2 consists of a *delete* operation and the inner transaction includes an *insert* operation.

Unlike closed or flat nesting, open nesting executes multiple commit operations. A commit of an inner transaction involves acquiring and releasing the abstract lock, as illustrated in Figure 8.1(b). In distributed systems, this involves communication overheads due to remotely acquiring and releasing the lock. [74].

Meanwhile, when T_1 aborts, its inner transaction must execute the abort procedure *onAbort*, because the $value$ in T_1 depends on its inner transaction. Although T_2 does not depend on its inner transaction, the inner transaction executes *onAbort* and acquires the abstract lock again when it restarts. Whenever an outer transaction aborts, its inner transaction must execute a compensating action (e.g., $bst.delet(value, t)$), regardless of object dependencies.

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 [39]. Compensating actions and attempts for acquiring abstract locks for distributed open-nested transactions will take longer time due to the communication overhead, which increases the likelihood for conflicts, degrading performance.

Motivated by these observations, we propose the DATS scheduler for open-nested DTM. We identify which object is subject to a conflict when an outer transaction validates. DATS determines whether the conflicted object has dependency on nested transactions. If there is no dependency between nested and outer transactions, only the conflicting outer transaction is restarted to preserve the inner transactions' abstract lock. Otherwise, the conflicting transaction aborts and other transactions execute compensating actions only when the nested transactions have dependency on the conflicted object.

8.2 Scheduler Design

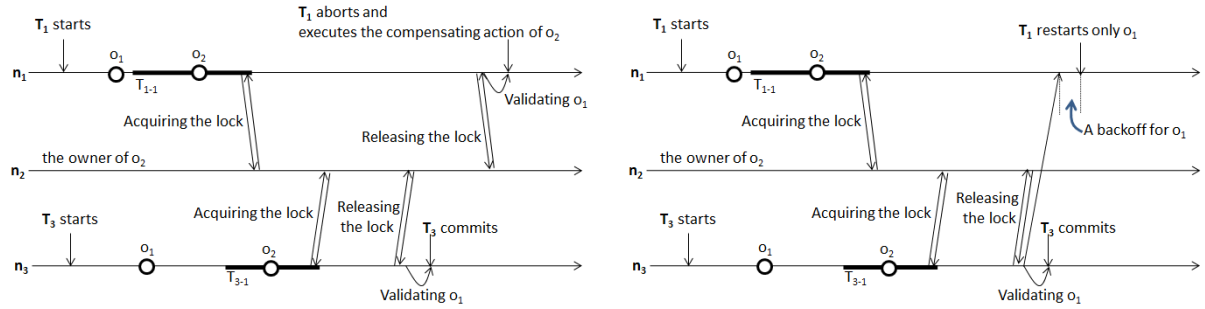
When an outer transaction validates an object, the object owner lets the transactions that have requested the object know that the object has been validated. Those transactions will then abort and DATS determines whether compensating actions must be executed. When an outer transaction aborts, DATS performs two actions: First, DATS detects which object is subject to the conflict. While a transaction T_i validates an object o_j, suppose that transaction T_k is designated to abort

due to the early validation of T_i. DATS lets T_k know that o_j is being validated by T_i. Before T_k validates o_j, T_k rolls-back only the operations that have been executed after requesting o_j if only o_j has been conflicted.

Second, DATS detects which inner transactions depend on the conflicted object o_j. If a nested transaction T_{k-1}, which already has committed, has used o_j, then T_{k-1} must abort and its compensating action must be executed. If T_{k-1} does not have any dependency on o_j, then T_k restarts, except for T_{k-1}. If T_k has conflicted due to only o_j, then only the operations involving o_j in T_k restart, increasing performance. If T_{k-1} has acquired the abstract lock of an object, the abstract lock will be preserved, reducing the number of acquiring the lock.

Additionally, DATS performs two actions to reduce the delays for open nested transactions. First, when an outer transaction aborts due to a conflict, its committed inner transactions will be protected from the abort if they do not depend on the conflicting outer transaction for any object. Second, when an inner transaction has acquired the lock of an object that does not depend on its outer transaction, the acquisition will be reserved. Thus, DATS increases performance by minimizing the number of requests to acquire a lock and unnecessary compensating actions.

8.3 Illustrative Example



(a) TFA-ON (b) DATS on TFA-ON

Figure 8.2: A Scenario of DATS and TFA-ON

Figure 8.2 illustrates an example of TFA-ON and DATS with two transactions T_1 and T_3 invoked on nodes n_1 and n_3, respectively. The circles indicate objects for write operations. Objects are represented on horizontal lines with the circles. The horizontal line corresponding to the status of each transaction describes the time domain.

Figure 8.2(a) shows an example of open nested transactions under TFA-ON. T_1 does a write operation on o_1 and its inner transaction T_{1-1} requests o_2. T_{1-1} acquires o_2's lock from the owner of o_2 (e.g., n_2). T_3 does a write transaction on o_1 and its inner transaction T_{3-1} requests o_2. T_{3-1}

acquires o_2's lock. When T_3 validates o_1, T_{3-1} releases the lock. When T_1 validates o_1, T_1 aborts and executes the compensating action of T_{1-1} due to the early validation of T_3.

Figure 8.2(b) shows an example of DATS with TFA-ON. When T_3 validates o_1, the owner of o_1 lets T_3 know the address of n_1 invoking T_1. T_3 sends an abort message with a backoff time to n_1. T_1 restarts after the backoff time. DATS checks the dependency between T_{1-1} and o_1. If there is no dependency between them, T_1 restarts only the operation on o_1 and the abstract lock of o_2 acquired by T_{1-1} is preserved.

We now describe how to determine the backoff time and determine object dependencies in Section 8.4.

8.4 Algorithms

We now present the algorithms for DATS. There are four algorithms: $Request\_Object$ and $Retrieve\_Request$ (Algorithm 15), and $Commit$ and $Abort$ (Algorithm 16).

The procedure, $Request\_Object$, is invoked whenever a new object needs to be requested. $Request\_Object$ 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 $Request\_Object$.

After finding the owner of the object in $Request\_Object$, 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 assigned backoff time is not 0, the requester waits for the backoff time to elapse. This implies that the requester requests an object, which is being validated by another transaction and therefore does not need to roll back. If the backoff time expires, $Open\_Object$ returns null. Otherwise, the requester wakes up and receives the object. $TransactionQueue$ is used to check the status of live transactions.

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* [8] representation of the most current successful commit times of write transactions. Whenever a transaction starts, an expected commit time is recorded in the table. A backoff time is computed as $expected\_commit\_time$ - $current\_time$ (i.e., the time for validating an object).

$Retrieve\_Request$ is invoked when an object owner receives a request. get_Object returns the object corresponding to oid. If get_Object returns null, it is not the owner of oid. Thus, 0 is assigned as the backoff time, and the requester must retry to find a new owner. If the corresponding object is being validated, $Retrieve\_Request$ has to calculate a backoff time.

Algorithm 16 shows the $Commit$ and $Abort$ procedures. Whenever a transaction commits, the procedure, $Commit$ is invoked and receives $txid$ and $objects$ to be committed as inputs. If the transaction is an open-nested (or inner) transaction, $LoadClass$ reads the application's bytecodes and $CheckDependency$ analyzes the dependency of the nested transaction [41]. The list

Algorithm 15: Algorithms of $Request\_Object$ and $Retrieve\_Request$

```
1  Procedure Request_Object
2  Input: Transaction_Type type, Transaction_ID txid, Object_ID oid
3  Output: null, object
4  owner = find_Owner(oid);
5  Send type, oid, and txid to owner;
6  Wait until object is received;
7  Receive object and backoff from Retrieve_Request;
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_Request;
13         if object is not null then
14             return object;
15         else
16             TransactionQueue.remove(txid);
17     return null;
18 return object;
19 Procedure Retrieve_Request
20 Input: type, oid. txid
21 object = get_Object(type, oid);
22 address = get_Requester_Address();
23 Integer backoff = 0;
24 if object is not null and type is write then
25     if object is being validated then
26         backoff = expected_commit_time - current_time;
27         object=null;
28 Send object and backoff to address;
```

$DependentObjects$ that $CheckDependency$ returns includes all objects that the inner transactions have used. The $object$ to be committed by the inner transaction is removed from $DependentObjects$. If $DependentObjects$ is not empty, $txid$ is added to $NestedTxs$ with the oids of the dependent objects $DependentObjects.oid$. This means that the outer transaction has requested oids. $CommitNestedTx()$ invokes $onComit$, illustrated in Figure 8.1(b). If an outer transaction invokes $Commit$, a backoff time is sent to the outer transactions requesting $objects$. $Request\_Object$ will be invoked after the backoff time elapses to minimize conflicts.

In the meanwhile, if a transaction aborts, the $Abort$ procedure is invoked. If the transaction is an outer transaction and its committed objects exist in $NestedTxs$, then the inner transactions that use those objects abort. Only the aborts of the nested transactions lead to the execution of compensating actions.

The time complexity is $O(1)$ to enqueue a transaction. In Algorithm 16, a backoff message is sent to all outer enqueued transactions for each object, so the time complexity is $O(the\ number\ of$

Algorithm 16: Algorithms of $Commit$ and $Abort$

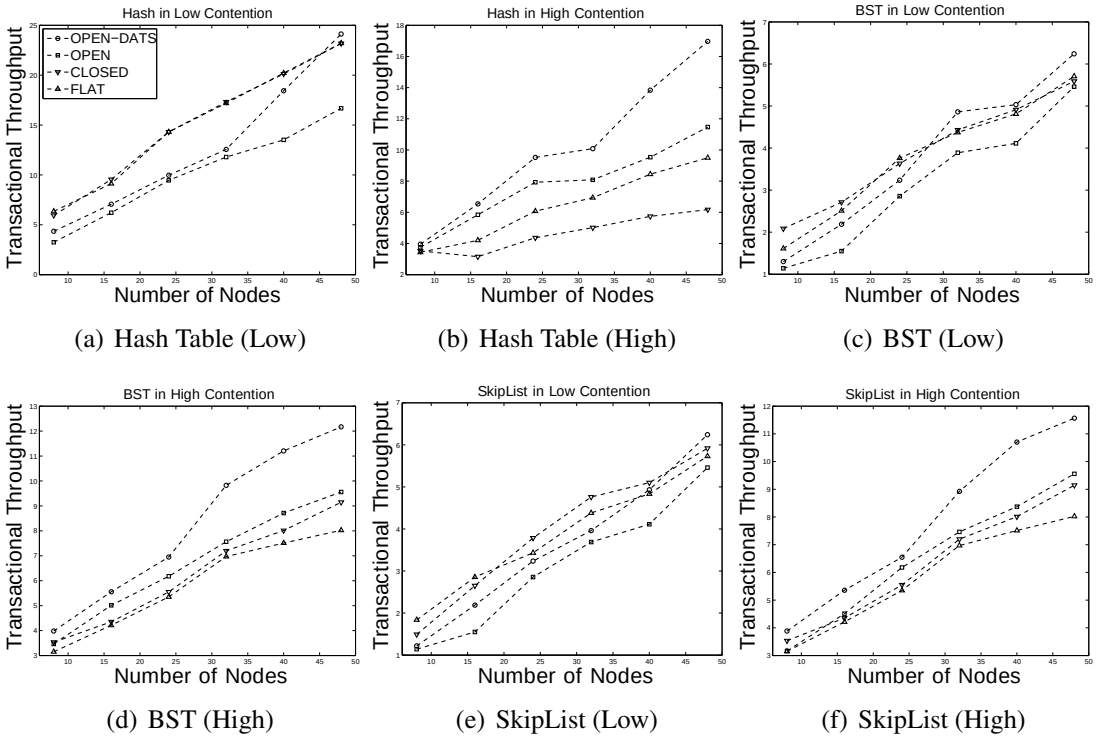
```
1  Procedure Commit
2  Input: txid, objects
3  foreach objects do
4      if txid is open nesting then
5          LoadClass(application.java);
6          DependentObjects = CheckDependency(object);
7          Remove object form DependentObjects;
8          if DependentObjects is not null then
9              NestedTxs.put(DependentObjects.oid,txid); CommitNestedTx(); return;
10     owner = find_Owner(object);
11     Send owner a message to obtain the addresses of the nodes requesting object;
12     if received addresses is not null then
13         Send expected_commit_time - current_time to addresses for a backoff;
14     Validate object;
15     if received addresses is not null then
16         Send object to addresses;
17 Procedure Abort
18 Input: txid, objects
19 if txid is outer then
20     foreach objects do
21         nestedIds = NestedTxs.remove(object.id);
22         if nestedIds is not null then
23             foreach nestedIds do
24                 AbortNestedTx(nestedId);
25 AbortOuterTx(txid);
```

$enqueued\ outer\ transactions$).

8.5 Evaluation

We implemented DATS in the HyFlow DTM framework [63, 73]. We developed four microbenchmarks for experimental studies. The benchmarks include distributed data structures including Hash Table, Binary Search Tree (BST), and SkipList [28]. We compared DATS under TFA-ON (OPEN-DATS) with only TFA-ON (OPEN) [74], closed nested transaction (CLOSED) [73], and flat nested transaction (FLAT). We compare with CLOSED and FLAT to show that OPEN does not always perform better than them, while OPEN-DATS consistently outperforms OPEN.

Figures 8.3 and 8.4 show the throughput of the three benchmarks with a different number of inner transactions, under low and high contention. In high contention, the number of aborts increases. Outer transactions frequently abort, and corresponding compensating actions are executed. DATS



(a) Hash Table (Low) (b) Hash Table (High) (c) BST (Low)

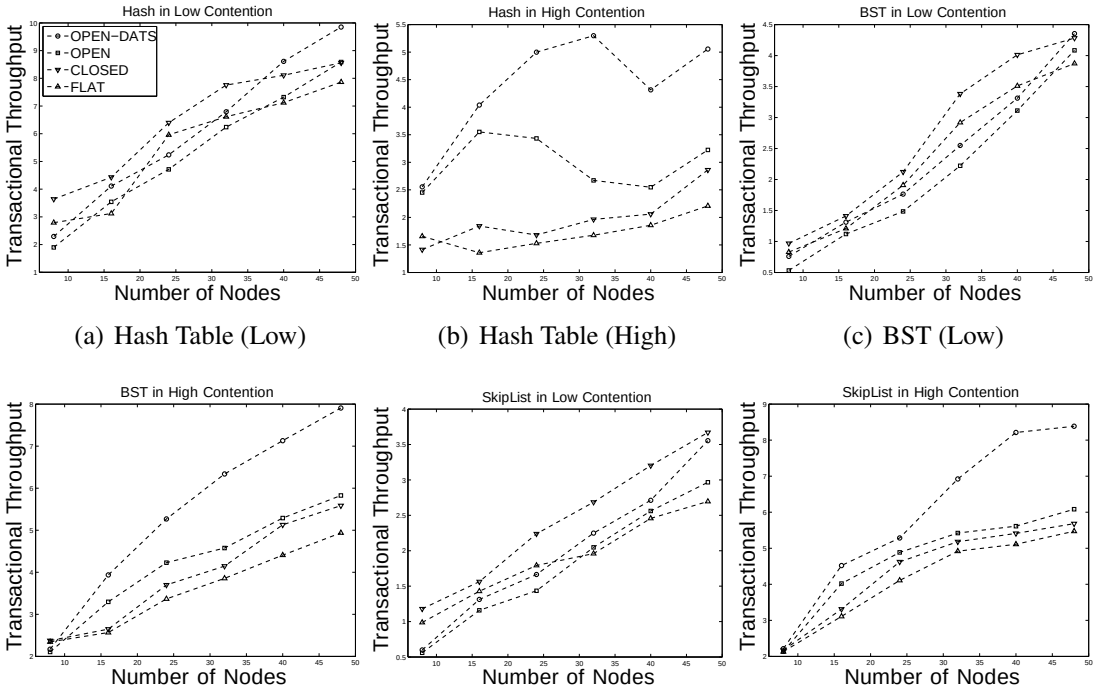
(d) BST (High) (e) SkipList (Low) (f) SkipList (High)

Figure 8.3: Throughput of 3 benchmarks with 4 inner transactions per outer transaction.

reduces the number of the compensating action executions. Also, the commit overheads of the outer transaction with four inner transactions are less than that of the outer transaction with eight inner transactions. Thus, the throughput of OPEN-DATS with eight inner transactions is improved more than that with four inner transactions.

OPEN performs worse than others in low contention (e.g., read-dominated workloads). Also, CLOSED and FLAT outperforms OPEN at small number of nodes. Under these conditions, object updates are rare. FLAT and OPEN take advantage of concurrent read operations, but OPEN has to validate the objects used for read operations. In this condition, OPEN-DATS less reduces the number of abstract locks and compensating action executions. However, in high contention and with large number of inner transactions, OPEN-DATS outperforms others.

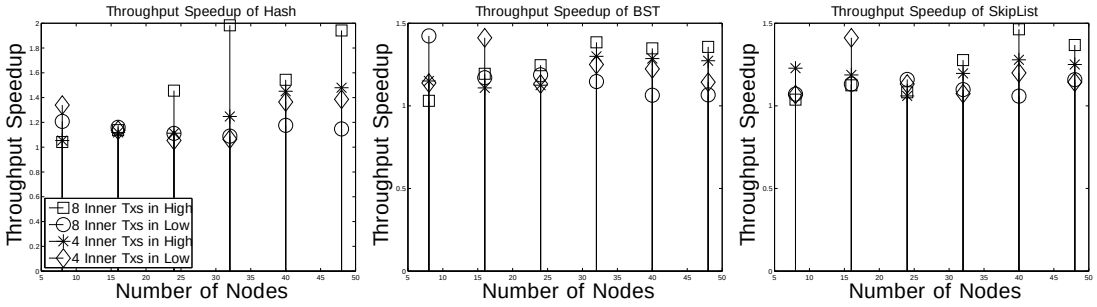
We measured the throughput speedup of OPEN-DATS over OPEN – i.e., the ratio of OPEN-DATS's throughput to the throughput of OPEN. Figure 8.5 shows the throughput speedup of the three benchmarks. We observe that OPEN-DATS improves throughput over OPEN by as much as 1.41× and 1.98× under low and high contention, respectively. We also observe that OPEN-DATS's throughput speedup under four inner transactions is higher than that under eight inner transactions, but its throughput speedup under eight inner transactions is higher than that under four inner transactions. The speedup under eight inner transactions in high contention (e.g., 90% write operations) is higher because of the large number of commits and compensating actions.



(a) Hash Table (Low) (b) Hash Table (High) (c) BST (Low)

(d) BST (High) (e) SkipList (Low) (f) SkipList (High)

Figure 8.4: Throughput of 3 benchmarks with 8 inner transactions per outer transaction.



(a) Hash Table (b) BST (c) SkipList

Figure 8.5: Throughput Speedup of Three Benchmarks against TFA-ON

Chapter 9

Summary, Conclusions, and Proposed Post Preliminary-Exam Work

9.1 Summary

In this dissertation proposal, we studied five 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, PTS has been designed for multi-version DTM. The key idea is to detect which object version is subject to the conflict in the event-based cc DTM and assign backoff times to conflicting transactions to minimize communication overhead. Our evaluation shows that PTS enhances throughput over two state-of-the-art replicated DTM solutions, GenRSTM and DecentSTM, by as much as (average) 3.4$\times$ and 3.3$\times$ under low and high contention, respectively.

Third, 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$\times$ and 1.73$\times$ under low and high contention, respectively.

Fourth, 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

(53%) $\sim$ 1.88 (88%) $\times$ speedup, respectively.

Finally, 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.

All five proposed transactional schedulers focus on data-flow cc DTM, but consider different aspects of the DTM problem space. Bi-interval focuses on single-version DTM. PTS focuses on large monotonic write transactions in multi-version DTM. CTS considers replicated DTM, which increases concurrency and availability. RTS and DATS have been proposed for scheduling closed and open-nested transactions, respectively.

9.2 Conclusions

Bi-interval 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.

PTS focuses on how to reduce the aborts of only large write transactions, because MV-STM inherently guarantees commits of all read transactions, and large write transactions are exposed to a high probability of conflicts. PTS breaks down large write transactions to find the restart point at which they have conflicted, preventing entire transactions from aborting. This results in throughput being improved two times more than other schedulers.

CTS uses multiple clusters to support partial replication for fault-tolerance. 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.

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. Our implementation and evaluation shows that, this strategy is effective and increases throughput.

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. At its core, our work shows that, identifying the underlying causes (e.g., repeated abort of large write transactions, repeated acquisition of remote objects/abstract locks) and eliminating them can yield significant throughput improvement.

9.3 Proposed Post Preliminary-Exam Work

9.3.1 Satisfying Update Serializability

The update serializability (US) [30] consistency criterion, originally studied for databases, is weaker than the opacity consistency criterion satisfied by the schedulers that we have developed so far. With US, read-only transactions are guaranteed to never abort. This is achieved by ensuring that a) they observe snapshots consistent with serializable executions, and b) concurrent read-only transactions may observe different, but compatible snapshots consistent with serializable executions. This means that, read-only transactions may observe different ordering of logically independent operations, which does not violate read-dominated workloads' correctness.

US is achieved by multiversion concurrency control: read-only transactions concurrent to write transactions always read the immediately previous update, ensuring guaranteed commit. The Genuine Multi-version Update serializability protocol (GMU) in [54] is one of the first genuine partial replication protocol guaranteeing that read-only transactions are never aborted, or forced to undergo any additional remote validation phase. GMU determines which object versions have to be returned by read operations of transactions to support US. For asynchronous distributed systems, GMU ensures agreement (only) among the nodes replicating the object updated by a transaction. However, GMU does not consider scheduling transactions to improve throughput.

CTS considers a partial replication model in an asynchronous distributed system, but uses non-genuine multicasting and a single object version model. Our proposed approach is to enhance CTS design to satisfy US. One way to do this is for object owners of each cluster to maintain object versions. When a read transaction T_1 requests an object o, the proposed scheduler can check the dependency (e.g., read transaction needs o being updated by a write transaction) of all current transactions that request o. If T_1 depends on T_2, then T_1 cannot ignore the effects of T_2 and all write transactions that T_2 depends upon. If there exists a dependency, T_1 should be enqueued until the commit of T_2 and all write transactions that T_2 depends upon, guaranteeing US.

9.3.2 Satisfying Strong Eventual Consistency

Eventual consistency (EC) and strong eventual consistency (SEC) [70] criteria, again, originally studied for databases, are also weaker than the opacity consistency criterion satisfied by our sched-

ulers.

Eventual consistency promises better availability and performance [75]. A write transaction updates some replica without synchronization, and then all write transactions take effect at all replicas. An object satisfies SEC if two properties are satisfied [71]: a) *eventual delivery*, which means that an update delivered to a replica is eventually delivered to all correct replicas, and b) *strong convergence*, which means that, replicas delivered the same updates have equivalent states. SEC can be achieved remarkably simply: an update to a replica is delivered to other replicas infinitely often, resulting in those replicas eventually receiving it, and converging to equivalent states.

Note that the two properties for achieving SEC avoid global synchronization (i.e., system-wide commit), maximizing scalability. We propose a new transactional scheduler to support SEC for better availability and performance. The transactional scheduler maintains transactions to guarantee that objects are SEC. When a write transaction starts, its operation's type is propagated to determine whether it changes the state of the object or not. If the transaction's result will change the state, its update will be delivered for strong convergence. Otherwise, its update does not take effect.

9.3.3 Leveraging Genuine Atomic Multicast

In distributed systems, object replication plays a fundamental role in both performance (e.g., read concurrency) and fault-tolerance (e.g., data availability in spite of node/link failures). In particular, partial replication, which increases scalability, has been developed, leveraging two kinds of multicast protocols: non-genuine and genuine multicast protocols. Non-genuine multicast protocol deliver multiple messages addressed to multiple groups, reducing scalability. Genuineness maximizes scalability, delivering one message to support one-copy serializability. Non-genuine protocols offer better performance than genuine protocols in all considered scenarios, except in large and highly loaded systems [69]. In large-scale distributed systems, many works have studied genuine partial replication to increase both performance and scalability [68, 67, 54].

CTS is based on atomic multicasting to ensure consistency. Atomic multicast allows messages to be addressed to a subset of the members of the system. In the case of CTS, that subset is the set of owners of each cluster. Our proposed approach is to design an enhanced CTS that leverages genuine multicast. One way to do this is as follows. When a write transaction starts validating an object, a message is sent to the owner subset. The object owners involved in the set may update the object's status to "validating". When the write transaction ends validating the object, its local clock (LC) is updated to the up-to-date clock, and the object status is changed to "ready". Like GMU, an agreement procedure is needed because genuine multicast is not reliable. If some of the object owners did not update the object status or LC, another transaction may request the old object, and will abort, because the LC is old. Meanwhile, some of the object owners may not change "validating" to "ready", aborting requested transactions. To avoid this, the proposed scheduler could maintain a scheduling queue of aborted transactions. This queue will be joined to the object owner holding up-to-date objects, ensuring one-copy serializability and reducing object requesting

time.

9.3.4 Evaluation using Industrial-strength Benchmarks

Our current evaluations of the schedulers were done using academic benchmarks including distributed versions of the STAMP benchmark [12] and STMBench7 [28]. This only constitutes a first-order evaluation. A more rigorous evaluation using industrial strength benchmarks such as TPC-B [1], BDB, and YCSB [54] is therefore highly desirable.

The transaction processing performance council benchmark (TPC-B) [1] consists of update transactions for database management systems, in both stand-alone and client-server contexts. Berkeley DB (BDB) is a software library that provides a high-performance embedded database for key and value data. BDB supports thousands of simultaneous threads of control or concurrent processes manipulating a database, which is suitable for DTM. Yahoo cloud serving benchmark (YCSB) [15, 54] is a framework specifically aimed at benchmarking NoSQL key-value data grids and cloud stores for performance comparison.

We propose to evaluate the proposed schedulers using these benchmarks.

Bibliography

[1] Transaction processing performance council (TPC). Available `http://www.tpc.org/tpcb`, 2012.

[2] Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 110–112, New York, NY, USA, 2008. ACM.

[3] Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. In *Proceedings of the 2006 workshop on Memory system performance and correctness*, MSPC '06, pages 70–81, New York, NY, USA, 2006. ACM.

[4] Mohammad Ansari, Mikel Lujn, Christos Kotselidis, Kim Jarvis, Chris Kirkham, and Ian Watson. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In André Seznec, Joel S. Emer, et al., editors, *HiPEAC*, volume 5409 of *LNCS*, pages 4–18. Springer, 2009.

[5] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. In *OPODIS*, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.

[6] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized STM algorithm. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[7] G. Blake, R.G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 156 –167, dec. 2009.

[8] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[9] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP*, pages 247–258, New York, NY, USA, 2008. ACM.

[10] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. Transactional predication: high-performance concurrent sets and maps for stm. In *PODC*, pages 6–15, 2010.

[11] Stefan Büttcher and Charles L. A. Clarke. Indexing time vs. query time: trade-offs in dynamic information retrieval systems. CIKM '05, pages 317–318, New York, NY, USA, 2005. ACM.

[12] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, and Kunle Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.

[13] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. An effective hybrid transactional memory system with strong isolation guarantees. In *ISCA*, Jun 2007.

[14] 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*, pages 271 –274, aug. 2011.

[15] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *SoCC*, pages 143–154, 2010.

[16] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC*, nov 2009.

[17] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Dan Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[18] Michael J. Demmer and Maurice Herlihy. The arrow distributed directory protocol. In *DISC*, pages 119–133, London, UK, 1998. Springer-Verlag.

[19] Michael J. Demmer and Maurice Herlihy. The Arrow distributed directory protocol. In *DISC*, pages 119–133, London, UK, 1998. Springer-Verlag.

[20] Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC*, pages 125–134, New York, NY, USA, 2008. ACM.

[21] Aleksandar Dragojević, Rachid Guerraoui, et al. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09*, pages 7–16, 2009.

[22] Sérgio Miguel Fernandes and João 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, pages 179–188, New York, NY, USA, 2011. ACM.

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

[24] Rachid Guerraoui, Thomas A. Henzinger, and Vasu Singh. Permissiveness in transactional memories. In *DISC*, pages 305–319, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[26] Rachid Guerraoui and Michal Kapalka. Transactional memory: Glimmer of a theory. In Ahmed Bouajjani and Oded Maler, editors, *Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg.

[27] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP*, pages 175–184, New York, NY, USA, 2008. ACM.

[28] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STMBench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[29] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. *SIGARCH Comput. Archit. News*, 32(2):102, 2004.

[30] R. Hansdah and L. Patnaik. Update serializability in locking. *LNCS*, 243:171185, 1986.

[31] R. C. Hansdah and Lalit M. Patnaik. Update serializability in locking. In *Proceedings of the International Conference on Database Theory*, ICDT '86, pages 171–185, London, UK, UK, 1986. Springer-Verlag.

[32] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 388–402, Oct 2003.

[33] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS*, page 522, Los Alamitos, CA, USA, 2003. IEEE Computer Society.

[34] Maurice Herlihy, Victor Luchangco, Mark Moir, and III William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, Jul 2003.

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

[36] T. Kanungo, D.M. Mount, N.S. Netanyahu, C.D. Piatko, R. Silverman, and A.Y. Wu. An efficient k-means clustering algorithm: analysis and implementation. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(7):881 –892, jul 2002.

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

[38] Junwhan Kim and Binoy Ravindran. On transactional scheduling in distributed transactional memory systems. In *SSS*, volume 6366 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin / Heidelberg, 2010.

[39] Junwhan Kim and Binoy Ravindran. Scheduling closed-nested transactions in distributed transactional memory. *In IPDPS*, pages 1–10, 2012.

[40] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[41] Eugene Kuleshov. Using ASM framework to implement common bytecode transformation patterns. In *Sixth International Conference on Aspect-Oriented Software Development*, 2007.

[42] Sanjeev Kumar, Michael Chu, Christopher J. Hughes, Partha Kundu, and Anthony Nguyen. Hybrid transactional memory. In *PPoPP*, Mar 2006.

[43] Walther Maldonado, Patrick Marlier, Pascal Felber, Adi Suissa, Danny Hendler, Alexandra Fedorova, Julia L. Lawall, and Gilles 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*, pages 79–90, New York, NY, USA, 2010. ACM.

[44] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208. ACM Press, Mar 2006.

[45] Virendra J. Marathe, William N. Scherer III, and Michael 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.

[46] José F. Martínez and Josep Torrellas. Speculative synchronization: applying thread-level speculation to explicitly parallel applications. In *ASPLOS-X*, pages 18–29, New York, NY, USA, 2002. ACM.

[47] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logTM. *SIGPLAN Not.*, 41(11):359–370, 2006.

[48] E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981.

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

[50] J. Eliot B. Moss. Open-nested transactions: Semantics and support. In *Workshop of Memory Performance Issues*, 2006.

[51] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006.

[52] Yang Ni, Vijay S. Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '07, pages 68–78, New York, NY, USA, 2007. ACM.

[53] Jeffrey Oplinger and Monica S. Lam. Enhancing software reliability with speculative threads. In *ASPLOS-X*, pages 184–196, New York, NY, USA, 2002. ACM.

[54] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *32nd IEEE International Conference on Distributed Computing Systems*, ICDCS. IEEE Computer Society Press, 2012.

[55] Dmitri Perelman, Anton Bishevsky, Oleg Litmanovich, and Idit Keidar. Smv: Selective multiversioning stm. In *In Fifth ACM SIGPLAN workshop on Transactional Computing*. ACM, 2010.

[56] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *PODC*, pages 16–25, New York, NY, USA, 2010. ACM.

[57] Ravi Rajwar and James R. Goodman. Transactional lock-free execution of lock-based programs. In *ASPLOS-X*, pages 5–17, New York, NY, USA, 2002. ACM.

[58] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. Dependence-aware transactional memory for increased concurrency. In *MICRO*, pages 246–257, Washington, DC, USA, 2008. IEEE Computer Society.

[59] Torval Riegel, Christof Fetzer, Heiko Sturzrehm, and Pascal Felber. From causal to z-linearizable transactional memory. In *PODC*, pages 340–341, New York, NY, USA, 2007. ACM.

[60] Paolo Romano, Nuno Carvalho, Maria Couceiro, Luis Rodrigues, and Joao Cachopo. Towards the integration of distributed transactional memories in application servers clusters. In *Quality of Service in Heterogeneous Networks*, volume 22, pages 755–769. Springer Berlin Heidelberg, 2009. (Invited paper).

[61] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joao Cachopo. Cloud-TM: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44:1–6, April 2010.

[62] Daniel J. Rosenkrantz, Richard Edwin Stearns, and Philip M. Lewis II. An analysis of several heuristics for the traveling salesman problem. *SIAM J. Comput.*, 6(3):563–581, 1977.

[63] Mohamed M. Saad and Binoy Ravindran. Distributed transactional locking II and hyflow: A high performance distributed software transactional memory framework. In *Sixth ACM SIGPLAN workshop on Transactional Computing*. ACM, 2011.

[64] Mohamed M. Saad and Binoy Ravindran. Snake: Control-flow software transactional memory. In *Stabilization, Safety, and Security of Distributed Systems*, Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2011.

[65] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, Mar 2006.

[66] Bratin Saha, Ali-Reza Adl-Tabatabai, and Quinn Jacobson. Architectural support for software transactional memory. In *MICRO*, pages 185–196, Washington, DC, USA, 2006. IEEE Computer Society.

[67] N. Schiper, P. Sutra, and F. Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Reliable Distributed Systems, 2009. SRDS '09. 28th IEEE International Symposium on*, pages 166 –175, sept. 2009.

[68] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *Reliable Distributed Systems, 2010 29th IEEE Symposium on*, pages 214 –224, 31 2010-nov. 3 2010.

[69] Nicolas Schiper, Pierre Sutra, and Fernando Pedone. Genuine versus non-genuine atomic multicast protocols for wide area networks: An empirical study. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 166–175, Washington, DC, USA, 2009. IEEE Computer Society.

[70] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

[71] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *SSS*, pages 386–400, 2011.

[72] Arrvindh Shriraman, Michael F. Spear, Hemayet Hossain, Virendra J. Marathe, Sandhya Dwarkadas, and Michael L. Scott. An integrated hardware-software approach to flexible transactional memory. *SIGARCH Comput. Archit. News*, 35(2):104–115, 2007.

[73] Alex Turcu, Binoy Ravindran, and Mohamed Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

[74] Alex Turcu and Biony Ravindran. On open nesting in distributed transactional memory. Available `http://hyflow.org/hyflow/chrome/site/pub/opennesting-systor12-tech.pdf`, 2012.

[75] W. Vogels. Eventually consistent. ACM Queue, 2008.

[76] Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, March 1991.

[77] Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA*, pages 169–178, 2008.

[78] Bo Zhang and Binoy Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS*, pages 48–53, Berlin, Heidelberg, 2009. Springer-Verlag.

[79] Bo Zhang and Binoy Ravindran. Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *SRDS*, pages 268–277, Washington, DC, USA, 2009. IEEE Computer Society.

[80] Bo Zhang and Binoy Ravindran. Dynamic analysis of the Relay cache-coherence protocol for distributed transactional memory. In *IPDPS*, Washington, DC, USA, 2010. IEEE Computer Society.