Scheduling Transactions in Replicated Distributed Software Transactional Memory

Junwhan Kim and Binoy Ravindran
ECE Department, Virginia Tech, Blacksburg, VA, 24061
Email: junwhan, binoy@vt.edu

***Abstract*—Distributed software transactional memory (DTM) is an emerging, alternative concurrency control model for distributed systems that promises to alleviate the difficulties of lock-based distributed synchronization. Object replication can improve concurrency and achieve fault-tolerance in DTM, but may incur high communication overhead (in metric-space networks) to ensure one-copy serializability. We consider metric-space networks and develop a cluster-based object replication model for DTM. In this model, object replicas are distributed to clusters of nodes, where clusters are determined based on distance between nodes, to maximize locality and fault-tolerance and to minimize communication overhead. We develop a transactional scheduler for this model, called CTS. CTS enqueues live transactions and identifies some of the transactions that must be aborted in advance to enhance the concurrency of the other transactions over clusters, reducing a significant number of future conflicts. Our implementation and experimental evaluation reveals that CTS improves transactional throughput over state-of-the-art replicated DTM solutions by as much as (average) 1.55× and 1.73× under low and high contention, respectively.**

***Keywords*-Distributed Systems; Software Transactional Memory; Transactional Scheduling; Replicated Model;**

I. Introduction

Lock-based concurrency control suffers from programmability, scalability, and composability challenges [20]. Transactional memory (TM) promises to alleviate these difficulties. With TM, code that read/write shared objects is organized as transactions, which optimistically 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 resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, 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 [29] and is composable. Multiprocessor TM has been proposed in hardware (HTM), in software (STM), and in hardware/software combination [26].

Distributed STM (or DTM) has been similarly motivated as an alternative to distributed lock-based concurrency control. DTM can be classified based on the system architecture: cache-coherent DTM (cc DTM) [21], [28], in which a set of nodes communicate with each other by message-passing links over a communication network, and a cluster model (cluster DTM) [10], in which a group of linked computers works closely together to form a single computer. The most important difference between the two is communication cost. cc DTM assumes a *metric-space* network (i.e., the communication cost between nodes form a metric), whereas cluster DTM differentiates between local cluster memory and remote memory at other clusters. cc DTM uses a cache-coherence protocol 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 [10], [23], [28].

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) [10], [9], [8], [4]. 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. (See Section V for discussion.) Thus, directly applying cluster DTM replication solutions to cc DTM may not yield similar performance.

We consider a *cluster-based partial object replication model* for 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.[1] 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 (i.e., a full replication model).

This paper focuses on how to schedule memory transactions in the cluster-based partial replication model for high

[1] Note our usage of the word *cluster* in the context of cc DTM to indicate grouping of nodes. This is not to be confused with *cluster DTM*.

performance, called cluster-based transactional scheduler (CTS). Each cluster has an object owner for scheduling transactions. In each object owner, CTS enqueues live transactions and identifies some of the transactions that must be aborted to avoid future conflicts, resulting in the concurrency of the other transactions.

We implemented CTS in a Java DTM framework, called HyFlow [28], and conducted experimental studies. Our studies reveal that transactional throughput is improved by up to 1.73× (on average) over two replicated DTMs, GenRSTM [8] and DecentSTM [4]. To the best of our knowledge, CTS is the first ever transactional scheduler for partially replicated cc DTM, and constitutes the paper's contribution.

The rest of the paper is organized as follows. We outline the preliminaries and the system model in Section II. We describe CTS and analyze its properties in Section III. Experimental studies are reported in Section IV. We overview past and related efforts in Section V and conclude in Section VI.

II. Preliminaries and System Model

We consider a distributed system which consists of a set of nodes $N = \{n_1, n_2, \cdots\}$ that communicate with each other by message-passing links over a communication network. Similar to [21], we assume that the nodes are scattered in a metric space. The metric $d(n_i, n_j)$ is the distance between nodes n_i and n_j, which determines the communication cost of sending a message from n_i to n_j. We assume that the nodes are *fail-stop* [31]. Additionally, communication links may also fail to deliver messages.

A. Distributed Transactions

A set of *distributed transactions* $T = \{T_1, T_2, \cdots\}$ is assumed. The transactions share a set of objects $O = \{o_1, o_2, \ldots\}$, which are assumed to be distributed in the network. A transaction contains a sequence of requests, each of which is a read or write operation request to an individual object. An execution of a transaction is a sequence of timed operations. An execution ends by either a commit (success) or an abort (failure). A transaction is in one of three possible states: *live*, *aborted*, or *committed*. Each transaction has a unique identifier (id), and is invoked by a node in the system.

We consider Herlihy and Sun's data flow DTM model [21]. In this model, transactions are immobile and objects move from node to node to invoking transactions. Each node has a *TM proxy* that provides interfaces to the local 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 from the network. Node n_k holding 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 there are multiple copies (or replicas) of an object in the network, the CC protocol is responsible for locating the *nearest* copy of the object in terms of the distance from the requesting node. Thus, T_i may incur *requesting* and *object retrieving times* to fetch o_j from the network in data flow DTM model [23], [24]. The requesting time of T_i is a communication delay for T_i's request invoked by n_i to travel in the network to n_k. The object retrieving time of T_i is a communication delay of o_j held by n_k to travel in the network to n_i. Our proposed transactional scheduler, CTS, ensures replica consistency in the sense that multiple copies of an object appear as a single logical object to the transactions i.e., the one-copy serializability property [3].

B. Atomicity, Consistency, and Isolation

We use the *Transactional Forwarding Algorithm* (TFA) [28] 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. As an extension of the Transactional Locking 2 (TL2) algorithm [12], TFA replaces the central clock of TL2 with independent clocks for each node and provides a means to reliably establish the "happens-before" relationship between significant events. 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).

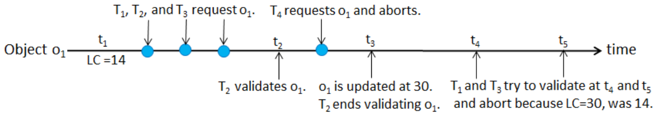

T_1, T_2, and T_3 request o_1.
T_4 requests o_1 and aborts.
Object o_1
t_1
LC =14
t_2
t_3
t_4
t_5
time
T_2 validates o_1.
o_1 is updated at 30.
T_2 ends validating o_1.
T_1 and T_3 try to validate at t_4 and t_5
and abort because LC=30, was 14.

Figure 1. An Example of TFA

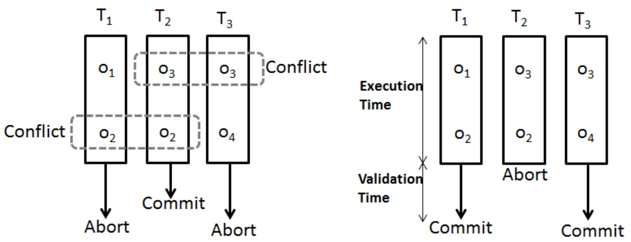
For completeness, we illustrate TFA with an example. In Figure 1, a transaction updates object o_1 at t_1 (i.e., local clock (LC) is 14) and four transactions (i.e., T_1, T_2, T_3, and T_4) request o_1 from the object holder. Assume that T_2 validates o_1 at t_2 and updates o_1 with LC=30 at t_3. Any 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_4 and t_5, respectively, T_1 and T_3 that have acquired o_1 with LC=14 before t_2 will abort, because LC is updated to 30.

III. Cluster-based Scheduling

A. Motivation

Directory-based CC protocols (e.g., Arrow and Ballistic) [11], [21] in the single-copy model often keep track of the single writable copy. In practice, not all transactional requests are routed efficiently; possible locality is often overlooked, resulting in high communication delays.

A distributed transaction consumes more execution time, which include the communication delays that are incurred in requesting and retrieving objects than a transaction on multiprocessors [24]. Thus, the probability for conflicts and aborts is higher. Even though a transaction in a full replication model does not request and retrieve objects, maintaining replicas of all objects at each node is costly. Increasing locality (and availability) by brute-force replication while ensuring one-copy serializability can lead to communication overhead. Motivated by this, we consider a k-cluster-based replication model for cc DTM. In this model, multiple copies of each object are distributed to k selected nodes to maximize locality and availability and to minimize communication overhead.



(a) TFA (b) CTS with TFA

Figure 2. Executing T_1, T_2, and T_3 Concurrently

Moreover, a transaction may execute multiple operations with multiple objects, increasing the possibility of conflicts. Figure 2 shows a scenario two conflicts occurring with three concurrent transactions, T_1, T_2, and T_3 using two objects. Under TFA, a conflict over o_2 between T_1 and T_2 occurs and another conflict over o_3 between T_2 and T_3 occurs. If T_2 commits first, T_1 and T_3 will abort because T_2 will update o_3 and o_2 even though T_1 and T_3 do not contend. If T_2 aborts as shown in Figure 2(b), T_1 and T_3 will commit. Motivated by this, CTS aborts T_2 in advance and allows T_1 and T_2 to commit concurrently. A contention manager resolves a conflict between two transactions, but CTS avoids two conflicts among three transactions and guarantees the concurrency of two transactions of them.

B. Scheduler Design

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

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

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

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

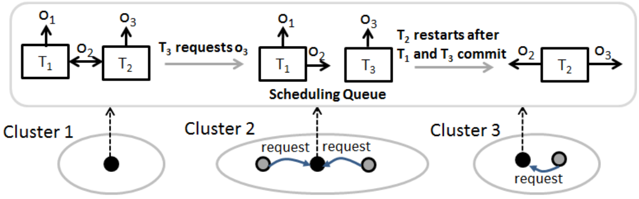


Figure 3. An Example of CTS

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

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

C. 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 [28], TFA is shown to exhibit opacity (i.e., its correctness property) and strong progressiveness (i.e., its progress property) [16]. 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. The local execution time of T_i is defined as γ_i, $\sum_{i=1}^{N} \gamma_i = \Gamma_N$ for N transactions. We consider three different models: no replication (NR), partial replication (PR), and full replication (FR) in cc DTM to show the effectiveness of CTS in PR.

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

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

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

Given schedulers A and B for N transactions, if RCR (i.e., $\frac{makespan_A^N(Model)}{makespan_B^N(Model)}$) < 1, A outperforms B. Thus, RCR of A and B indicates a relative improvement between schedulers A and B if $makespan_A^N(Model) < makespan_B^N(Model)$. 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 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 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 1 gives the total number of aborts on N transactions under scheduler B. If a transaction T_i requests an object, the communication delay will be $2\times d(n_i, n_j)$ for both requesting and object retrieving times. Once T_i aborts, this delay is incurred again. To complete N transactions using scheduler B, the total communication delay will be $2(N-1)\sum_{i=1}^{N-1} d(n_i, n_j)$. The theorem follows. ■

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

Theorem 5: Given scheduler B and N transactions, $makespan_B^N(FR) \leq makespan_B^N(PR) \leq makespan_B^N(NR)$.

Proof: Given k PR, $\lim_{k\to 1} makespan_B^N(PR) \leq 2(N-1)\sum_{i=1}^{N-1} d(n_i, n_j) + \Gamma_N$, 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: Given N transactions and M objects, the

RCR of schedulers CTS on PR and scheduler B on FR is less than 1, where $N > 3$.

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

Theorem 6 shows that CTS in PR performs better than FR. Even though PR incurs requesting and object retrieving times for transactions, CTS minimizes these times, resulting in less overall time than the broadcasting time of FR.

IV. Implementation and Experimental Evaluation

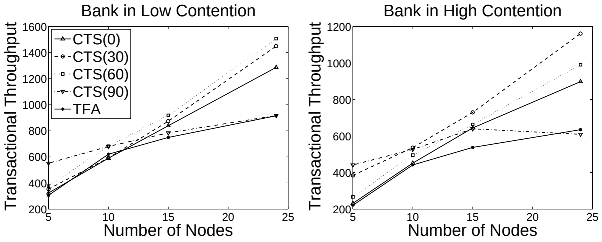
A. Experimental Setup

We implemented CTS in the HyFlow DTM framework [28], and developed six benchmarks for experimental studies. The benchmarks include two monetary applications (Bank and Loan) [28], distributed versions of the Vacation of the STAMP benchmark suite [7], and three distributed data structures including Counter, Red/Black Tree (RB-Tree) [17], and Distributed Hash Table (DHT).

To select k nodes for distributing replicas of each object, we group nodes into clusters, such that nodes in a cluster are closer to each other, while those between clusters are far apart. Recall that the distance between a pair of nodes in a metric-space network determines the communication cost of sending a message between them. We use a k clustering algorithm based on METIS [22], to generate k clusters with small intra-cluster distances i.e., k nodes may hold the same objects. Our partial replication relies on the usage of a *total order multicast* (TOM) primitive to ensure agreement on correctness in a genuine multicast protocol [30]. The object owners for each cluster update objects through a TOM-based protocol.

We use *low* and *high contention*, which are defined as 90% and 10% read transactions of one million active concurrent transactions per node, respectively [17]. A read transaction includes only read operations, and a write transaction consists of only write operations [17]. Our experiments were conducted on 24-node testbed. Each node is an AMD Opteron processor clocked at 1.9GHz. We use Ubuntu Linux 10.04 server OS and a network with a private gigabit ethernet. Each experiment is the average of ten repetitions. The number of objects for a transaction is selected randomly from 2 to 20. We considered CTS(30) and CTS(60), meaning CTS over 30% and 60% object owners of the total nodes, respectively. For instance, CTS(30) under 10 nodes means CTS over 3-clustering algorithm. We measured the *transactional throughput* (number of committed transactions per second) under increasing number of requesting nodes and failed nodes.

B. Evaluation



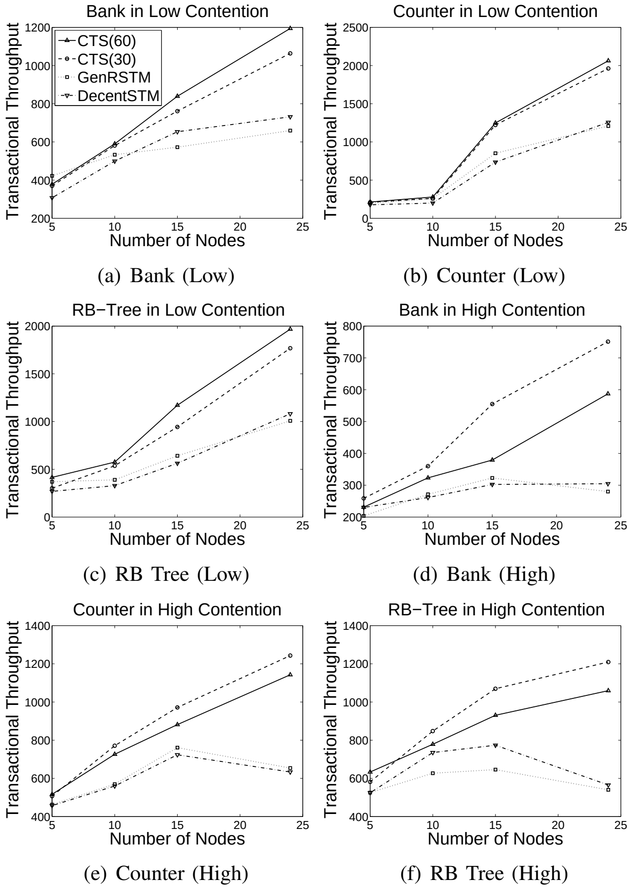
(a) Bank (Low) (b) Bank (High)

Figure 4. Throughput of Bank Benchmark with No Node Failures.

Figure 4 intends to show two effects of scheduling by CTS and the improvement of object availability by increasing object locality. To show the effectiveness of CTS, TFA is compared with CTS(0) – the combination of CTS and TFA with no replication. CTS(0) improves throughput over TFA as much as 1.5× under high contention because the number of conflicts decreases. CTS(0) outperforms CTS(90) in throughput, but it is non-fault-tolerant. The throughput produced by CTS(90) is degraded due to the large number of broadcasting messages needed to update all replicas. Due to high object availability on CTS(90), the requesting times of aborted transactions are less reduced. Meanwhile, due to low object availability on CTS(0), the requesting times are more reduced but object retrieving times increase. Thus, CTS(30) and CTS(60) achieve decreased object requesting and retrieving times, resulting in a better throughput than CTS(0) and CTS(90).

We considered two competitor DTM implementations: GenRSTM [8] and DecentSTM [4]. GenRSTM is a generic framework for replicated STMs and uses broadcasting to achieve transactional properties. DecentSTM implements a fully decentralized snapshot algorithm, minimizing aborts. We compared CTS with GenRSTM and DecentSTM.

Figure 5 shows the throughput of three benchmarks for CTS(30), CTS(60), GenRSTM, and DecentSTM with 20% node failure under low and high contention, respectively. In these experiments, 20% of nodes randomly fail. GenRSTM broadcasts updates to all other replicas, which incurs a overhead. DecentSTM is based on a snapshot isolation algorithm, which requires searching the history of objects to find a valid snapshot. This algorithm also incurs a significant overhead. Due to those overheads, their performance degrades for more than 24 requesting nodes. Thus we observe that CTS yields higher throughput than GenRSTM and DecentSTM. In particular, 60% of nodes are entitled to the ownership of an object based on CTS(60). CTS(60) maintains smaller



(a) Bank (Low)

(b) Counter (Low)

(c) RB Tree (Low)

(d) Bank (High)

(e) Counter (High)

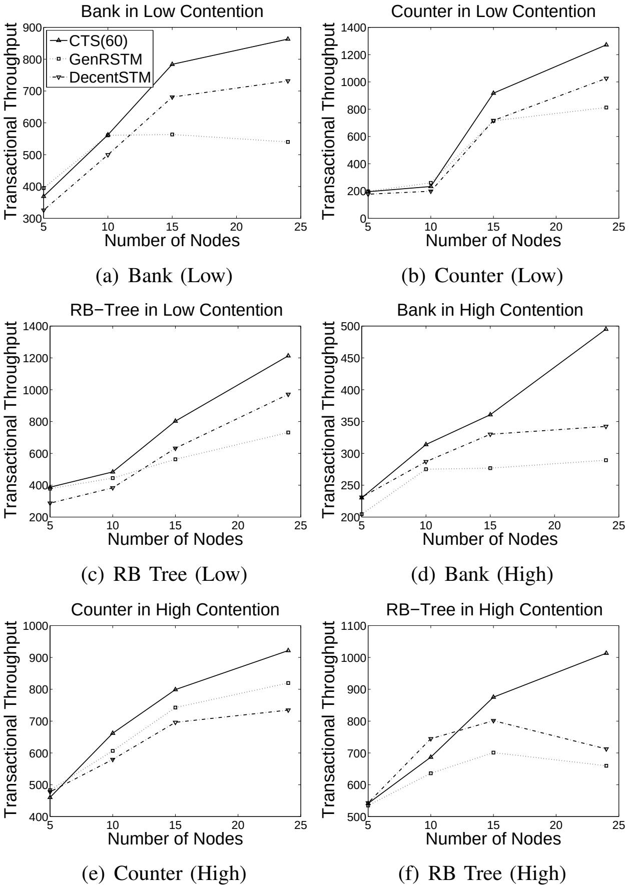
(f) RB Tree (High)

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

clusters than CTS(30), so the communication delays to request and retrieve objects decrease, but the number of messages increases. Under high contention, CTS avoids the large number of conflicts, so CTS yields much higher throughput than GenRSTM and DecentSTM.

Figure 6 shows the throughput of three benchmarks for CTS(60), GenRSTM, and DecentSTM with 50% node failure under low and high contention, respectively. GenRSTM's and DecentSTM's throughput do not degrade as the number of failed nodes increases, because every node holds replicated objects. However, in CTS, this causes communication delays to increase, degrading throughput, because object owners may fail or scheduling lists may be lost. Over less than ten nodes with 50% failed nodes, GenRSTM yields higher throughput than CTS, because the number of messages decreases. As the number of nodes increases, CTS outperforms GenRSTM and DecentSTM in throughput.

We computed the throughput speedup of CTS(60) over GenR STM and DecentSTM i.e., the ratio of CTS's throughput to the throughput of the respective competitor. Figure 7 summarizes the throughput speedup under 20% and 50% node failure. Our evaluations reveal that CTS(60) improves throughput over GenRSTM by as much as 1.9533 (95%) $\sim$ 2.0968 (109%) × speedup in low and high contention, respectively, and over DecentSTM by as much as 1.9622 (96%) $\sim$ 2.1683 (116%) × 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× and 1.73× under low and high contention, respectively.



(a) Bank (Low)

(b) Counter (Low)

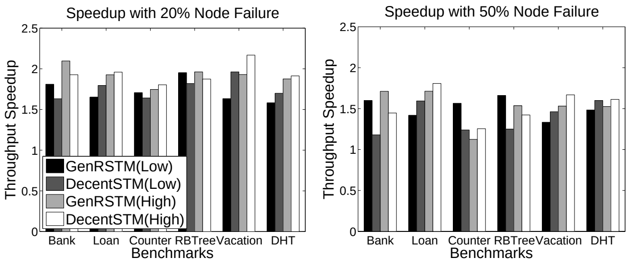
(c) RB Tree (Low)

(d) Bank (High)

(e) Counter (High)

(f) RB Tree (High)

Figure 6. Throughput of 3 Benchmarks with 50% Node Failure under Low and High Contention (5 to 24 nodes).



(a) 20% Node Failure

(b) 50% Node Failure

Figure 7. Summary of Throughput Speedup

V. Related Work

Transactional scheduling has been explored in a number of multiprocessor STM efforts [14], [1], [32], [13], [2].

In [14], Dragojević *et. al.* describe an approach that dynamically schedules transactions based on their predicted read/write access sets. In [1], 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) [32] 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 [13], which uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts.

Blake, Dreslinski, and Mudge propose the Proactive Transactional Scheduler (PTS) [5]. PTS detects "hot spots" of contention that can degrade performance, and proactively schedules affected transactions around those hot spots. Evaluation on the STAMP benchmark suite [7] shows PTS outperforming a backoff-based policy by an average of 85%.

Attiya and Milani present the BIMODAL scheduler [2], 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 writing and reading transactions are given priority, respectively, ensuring greater concurrency for read transactions. Kim and Ravindran extend the BIMODAL scheduler for DTM in [23]. Their scheduler, called Bi-interval, groups concurrent requests into read and write intervals, and exploits the tradeoff between object moving times (incurred in dataflow DTM) and concurrency of reading transactions, yielding high throughput.

All past transactional schedulers have been studied for the single-copy STM/DTM model. Replicated object models for DTM have been studied in [33], [10], [9], [27], [25], [6], but these efforts do not consider or support transactional scheduling. While [33] consider cc DTM, all other efforts focus on cluster DTM.

Zhang and Ravindran [33] propose a quorum-based replication (QR) framework for DTM to enhance availability of objects without incurring high communication overhead. All nodes based on QR have to hold all objects, and one-copy serializability is ensured using a flooding algorithm.

D^{2}STM [10] relies on a commit-time atomic broadcast-based distributed validation to ensure global consistency. Motivated by database replication schemes, distributed certification based on atomic broadcast [18] avoids the costs of replica coordination during the execution phase and runs transactions locally in an optimistic fashion.

Carvalho *et. al.* present Asynchronous Lease Certification (ALC) DTM replication scheme in [9], which overcomes some drawbacks of atomic broadcast-based replication [10]. ALC reduces the replica coordination overhead and avoids unnecessary aborts due to conflicts at remote nodes using asynchronous leases. ALC relies on uniform reliable broadcast [18] to exclusively disseminate the *writeset*s, which reduces inter-replica synchronization overhead. Manassiev *et al.* present a page-level distributed multiversioning algorithm for cluster DTM [27].

Kotselidis *et al.* present the DiSTM cluster DTM framework in [25]. Under the TCC protocol [19], DiSTM induces large traffic overhead at commit time, as a transaction broadcasts its read/write sets to all other transactions, which compare their read/write sets with those of the committing transaction. Using lease protocols [15], this overheard is eliminated. However, they also show that an extra validation step is added to the master node as well as bottlenecks are created under high contention because of acquiring and releasing the leases.

None of the replication models for cc and cluster DTM consider transactional scheduling. Also, as mentioned before, broadcasting transactional read/write sets or memory differences as done for cluster DTM is inherently non-scalable for cc DTM (which is our focus), as messages transmitted grow quadratically with the number of nodes.

VI. Conclusions

We presented a transactional scheduler for a replicated object model in cc DTM, called CTS. 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), enqueues live transactions, and identifies transactions that must be aborted for enhancing concurrency of other transactions. CTS's design shows how cluster-based transactional scheduling impacts throughput in DTM. Our implementation and experimental evaluation shows that CTS enhances transactional throughput over two state-of-the-art replicated DTM solutions, GenRSTM and DecentSTM, by as much as (average) 1.55× and 1.73× under low and high contention, respectively.

Acknowledgements

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385.

References

[1] Mohammad Ansari, Mikel Lujn, et al. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC*, pages 4–18, 2009.

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

[3] Philip A. Bernstein and Nathan Goodman. Multiversion concurrency controltheory and algorithms. *ACM Trans. Database Syst.*, 8:465–483, December 1983.

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

[5] G. Blake, R.G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *MICRO-42*, pages 156 –167, 2009.

[6] 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.

[7] Chi Cao Minh, JaeWoong Chung, et al. STAMP: Stanford transactional applications for multi-processing. In *IISWC*, September 2008.

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

[9] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Middleware*, pages 376–396, Berlin, Heidelberg, 2010. Springer-Verlag.

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

[11] Michael J. Demmer and Maurice Herlihy. The arrow distributed directory protocol. In *DISC*, pages 119–133, 1998.

[12] David Dice, Ori Shalev, and Nir Shavit. Transactional locking II. In *DISC*, pages 194–208, 2006.

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

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

[15] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *In SOSP*, pages 202–210, NY, USA, 1989. ACM.

[16] Rachid Guerraoui and Michal Kapalka. Transactional memory: Glimmer of a theory. In *CAV*, volume 5643 of *LNCS*, pages 1–15. Springer Berlin, 2009.

[17] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. STM-Bench7: a benchmark for software transactional memory. *SIGOPS Oper. Syst. Rev.*, 41(3):315–324, 2007.

[18] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[19] 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.

[20] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.

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

[22] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20:359–392, 1998.

[23] Junwhan Kim and Binoy Ravindran. On transactional scheduling in distributed transactional memory systems. In *SSS*, pages 347–361, 2010.

[24] Junwhan Kim and Binoy Ravindran. Scheduling closed-nested transactions in distributed transactional memory. *In IPDPS*, 0:179–188, 2012.

[25] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[26] James R. Larus and Ravi Rajwar. *Transactional Memory*. Morgan and Claypool, 2006.

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

[28] Mohamed Saad and Binoy Ravindran. Supporting STM in distributed systems: Mechanisms and a Java framework. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, June 2011.

[29] Bratin Saha, Ali-Reza Adl-Tabatabai, et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, pages 187–197, 2006.

[30] N. Schiper, P. Sutra, and F. Pedone. P-store: Genuine partial replication in wide area networks. In *SRDS*, pages 214 –224, 31 2010-nov. 3 2010.

[31] Richard D. Schlichting and Fred B. Schneider. Fail-stop processors: an approach to designing fault-tolerant computing systems. *ACM Trans. Comput. Syst.*, 1:222–238, August 1983.

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

[33] Bo Zhang and Binoy Ravindran. A quorum-based replication framework for distributed software transactional memory. In *OPODIS*, 2011.