Scheduling Closed-Nested Transactions in Distributed Transactional Memory

Junwhan Kim
ECE Dept., Virginia Tech
Blacksburg, VA, 24061
Email: junwhan@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA, 24061
Email: binoy@vt.edu

***Abstract*—Distributed software transactional memory (D-STM) is an emerging, alternative concurrency control model for distributed systems that promises to alleviate the difficulties of lock-based distributed synchronization—e.g., distributed deadlocks, livelocks, and lock convoying. We consider Herlihy and Sun's dataflow D-STM model, where objects are migrated to invoking transactions, and the *closed nesting* model of managing inner (distributed) transactions. We present a transactional scheduler called, reactive transactional scheduler (or RTS) to boost the throughput of closed-nested transactions. RTS determines whether a conflicting parent transaction must be aborted or enqueued according to the level of contention. If a transaction is enqueued, its nested inner transactions do not have to retrieve objects again, resulting in reduced communication delays. Our implementation of RTS in the HyFlow D-STM framework and experimental evaluations reveal that RTS improves throughput over D-STM without RTS, by as much as 88%.**

***Keywords*-Software Transactional Memory, Closed-Nested Transactions, Transactional Scheduling, Distributed Systems**

I. INTRODUCTION

Lock-based concurrency control suffers from scalability, programmability, and composability challenges [14]. These difficulties are exacerbated in distributed systems with nodes, possibly multicore, interconnected using message passing links, due to additional, distributed versions of their centralized problem counterparts [16].

Transactional memory (TM) promises to alleviate the difficulties with lock-based concurrency control. With TM, programmers organize code that read/write shared memory objects as transactions, which appear to execute atomically. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager [15] resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are restarted, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). In addition to a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking, especially during high contention situations [23]. Multiprocessor TM has been proposed in hardware, called HTM (e.g., [13]), in software, called STM (e.g., [8]), and in a combination, called Hybrid TM (e.g., [7]).

Distributed STM (D-STM) has built upon these results, as an alternative to distributed lock-based concurrency control. In Herlihy and Sun's dataflow D-STM model [16], transactions are immobile and objects are dynamically migrated to invoking transactions. The model requires a cache-coherence protocol, which locates an object's latest cached copy, and moves a copy to the requesting transaction, while guaranteeing one writable copy. Contention management is also needed. When an object is attempted to be migrated, it may be in use. Thus, a contention manager mediates object access conflicts, while avoiding deadlocks and livelocks. Similar to multiprocessor STM, D-STM provides a simple distributed programming model (e.g., locks are precluded in the interface), and effective performance (e.g., [18]).

Support for nesting transactions is essential for D-STM, for the same reasons that they are so for multiprocessor TM –i.e., composability, performance, and fault-management [20]. Composability is the ability to group atomic operations into larger atomic operations. 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 and D-STM. 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 multiprocessor

STM: *flat*, *closed*, and *open*. 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 [19], 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.

A complimentary approach for dealing with transactional conflicts is transactional scheduling. Broadly, a transactional scheduler determines the ordering of concurrent transactions so that conflicts are either avoided altogether or minimized. Two kinds of transactional schedulers have been studied in the past: reactive [8], [3], [17] and proactive [25], [4]. These schedulers cannot directly be used to schedule nested distributed transactions.

When a conflict between two transactions occurs, the contention manager determines which transaction wins or loses, and then the losing transaction aborts. Since aborted transactions might abort again in the future, reactive schedulers enqueue aborted transactions, serializing their future execution [8], [3], [17]. Past studies show that, such schedulers often causes only small number of aborts and reduces the total communication delay in D-STM [17]. 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 take a different strategy. Since aborted transactions should not abort again when re-issued, proactive schedulers abort the losing transaction with a backoff time, which determines how long the transaction is stalled before it is re-started [25], [4]. Determining backoff times for aborted transactions is generally difficult in D-STM. 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 D-STM.

We consider closed-nested transactions in D-STM, which is more efficient than flat nesting and guarantees *serialization* [1]. (Open nesting is similar to closed nesting, but may need different semantics for concurrency control [19].) 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 assign the enqueued transaction with a backoff time to boost transactional throughput. We implement RTS in a Java D-STM framework, called HyFlow [22], and conduct experimental studies. Our results reveal that transactional throughput is improved by up to 88% over D-STM without RTS. To the best of our knowledge, RTS is the first ever transactional scheduler for nested transactions in D-STM.

The rest of the paper is organized as follows. We present preliminaries of the D-STM model and state our assumptions in Section II. We describe RTS and analyze its properties in Section III. Section IV describes our experimental studies. We overview past and related efforts in Section V, and conclude in Section VI.

II. Preliminaries

We consider a distributed system that consists of a set of nodes that communicate with each other by message-passing links over a communication network. A set of *distributed transactions* $T = \{T_1, T_2, \cdots\}$ is assumed that share objects $O = \{o_1, o_2, \ldots\}$, which are 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, and is invoked by a node in the system.

We consider Herlihy and Sun's dataflow D-STM model [16], where transactions are immobile, and objects move from node to node to invoking transactions. In this model, 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 in 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.

We consider two properties of the CC protocol. First, when the TM proxy of T_i requests o_j, the CC protocol 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.) Second, at any given time, the CC protocol must locate only one copy of o_j in the network, and only one transaction is allowed to eventually write to o_j.

Figure 1 shows a code example that illustrates a nested transaction. The *tx_begin* and *tx_end* delimiters mark the

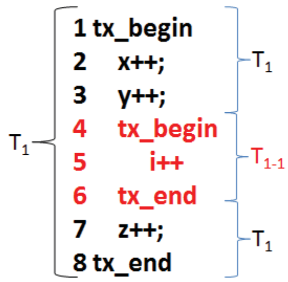


Figure 1. A code example where transaction T_{1-1} is nested inside T_1.

beginning and end of a transaction, respectively. T_1 is a parent transaction of its first nested transaction T_{1-1}. When T_1 starts, the CC protocol locates objects x and y to conduct the $++$ operation. The CC protocol independently locates the object i for T_{1-1}. After T_{1-1} commits, the protocol requests object z for T_1. Objects x and y are still in use unless T_1 commits or aborts.

We use the *Transactional Forwarding Algorithm* (or TFA) [22] 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.

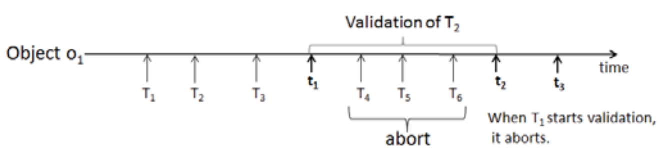


Figure 2. An Example of TFA

For completeness, we illustrate TFA with an example (See [22], [24] for detailed examples). In Figure 2, suppose that six transactions (i.e., T_1, T_2, T_3, T_4, T_5, and T_6) request object o_1 from the object holder. Assume that T_2 validates o_1 from t_1 to t_2. Write transaction T_1 that has requested o_1 before T_2 starts validation will abort when T_1 validates at t_3. T_2 creates a new version of o_1, which is different from the version that T_1 has requested, so T_1 aborts. In the meantime, while T_2 validates, all other requesting transactions (i.e., T_4, T_5, and T_6) abort.

There are two kinds of aborts on TFA in Figure 2. First, T_1 aborts due to the early validation of T_2. In this case, the transactions may have requested multiple objects and may have already conducted some operations on those objects. So they must abort. Second, losing transactions T_4, T_5, and T_6 abort while T_2 validates. A validation in distributed systems includes global registration of object ownership, which take a relatively long time due to communication overhead. Transactions that request an object being validated must abort. Many existing transactional schedulers optimize the order for re-executing aborted transactions to avoid their repeated aborts (e.g., [8], [25], [2]). If a nested transaction commits, its modifications become visible to its parent transaction. These changes only become visible for other transactions when the parent transaction commits [19]. Thus, RTS determines that parent transactions, which are designated to abort due to the second case of aborting in TFA, are aborted or enqueued to minimize aborts of its nested transactions.

III. Reactive Transactional Scheduling

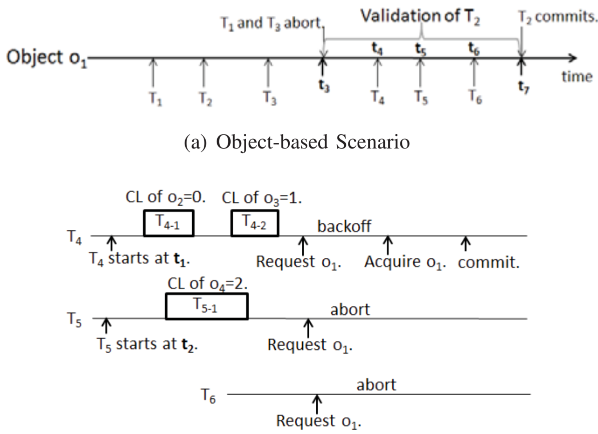
A. Overview

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 III-B.

B. Illustrative Example



(a) Object-based Scenario

(b) Transaction-based Scenario

Figure 3. A Reactive Transactional Scheduling Scenario

RTS assigns different backoff times for each enqueued transaction. A backoff time is computed as a percentage of estimated execution time. Figure 3 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 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 3 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* [5] 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 3, 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.

C. Algorithm Description

We now present the algorithms for RTS. There are three algorithms: Algorithm 2 for $Open\_Object$, Algorithm 3 for $Retrieve\_Request$, and Algorithm 4 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.

The data structures depicted in Algorithm 1 is used in Algorithms 3 and 4. The data structure of $Requester$ consists of the address of the transaction identifier of a requester. $Requester\_List$ maintains a linked list for $Requester$ and a contention level. $getContention()$ gives the total contention level of objects representing how many transactions have requested. $scheduling\_List$ is a hash table to hold a $Requester\_List$ including requesters for an object with $Object\_ID$.

Algorithm 2 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.

Algorithm 1: Structure of Scheduling Table

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

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 2: Algorithm of Open_Object

```
Procedure Open_Object
Input: Transaction_ID txid, Object_ID oid
Output: null, object
owner = Find_owner(oid);
Send oid, txid, myCL, and ETS to owner;
Wait until that Retrieve_Response is invoked;
Read object, backoff, and remoteCL from Retrieve_Response;
if object is null then
    if backoff is not 0 then
        TransactionQueue.put(txid);
        Wait for backoff;
        Read object and backoff from Retrieve_Response;
        if object is not null then
            return object;
        else
            TransactionQueue.remove(txid);
    return null;
else
    return object;
```

Algorithm 3 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 3: Algorithm of Retrieve_Request

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

In Algorithm 4, $Retrieve\_Response$ 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 $scheduling\_List$.

Algorithm 4: Algorithm of Retrieve_Response

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

Whenever an object is requested, RTS performs Algorithms 2, 3, and 4. 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 $O(CL\ threshold)$.

D. 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 [22], TFA is shown to exhibit opacity (i.e., its correctness property) [11] and strong progressiveness (i.e., its progress property [10]). 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 1: 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 2: 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 3.1: 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 3.2: 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 3.1 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 3.3: 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 3.4: 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, $\frac{\sum_{i=1}^{N} d(n_{i-1}, n_i)}{\sum_{i=1}^{N} d(n_0, n_i)} < \log N$ [21]. Hence, $M \times \log N < M \times (2N - 3)$, when $N \geq 2$. The theorem follows. ■

IV. Experimental Evaluation

A. Experimental Setup

We implemented RTS in the HyFlow D-STM framework [22] 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 [6], Bank as a monetary application [22], and four distributed data structures including Linked-List (LL), Binary-Search Tree (BST), Red/Black Tree (RB-Tree), and Distributed Hash Table (DHT) [12] 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 [8]. 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 50$msec$ 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 transactional throughput. Thus, in this experiment, the CL's threshold corresponding to the peak point is determined.

We conducted our experiments in a distributed system testbed comprised of 80 nodes, each of which is an Intel Xeon 1.9GHz processor, running Linux, and interconnected by message passing links.

B. Abort Rate of Nested Transactions

RTS minimizes the number of aborts of parent transactions, preventing committed nested transactions from aborting. However, some parent transactions holding committed nested transactions may abort due to early validation. Also, anticipating an exact execution time is too optimistic. An assigned backoff time may expire before the transaction can obtain an object, so parent transaction may lose all committed nested transactions. Thus, there are two causes to abort nested transactions. First, a nested transaction aborts due to the early validations or inconsistency of objects. Second, a nested transaction aborts due to its parent transactions' aborts.

Table I
ABORT RATE OF NESTED TRANSACTIONS

	Low Contention		High Contention	
	RTS	TFA	RTS	TFA
Vacation	25.6%	55.5%	29.1%	67.5%
Bank	21.5%	46.4%	23.3%	63.7%
Linked List	14.4%	37.6%	17.9%	43.2%
RB Tree	13.7%	32.2%	22.4%	45.1%
BST	11.1%	29.4%	17.5%	37.4%
DHT	12.8%	31.3%	19.9%	39.2%

We measure the number of nested transaction aborts caused by the two aforementioned cases. Table I shows the abort rate of nested transactions (i.e., nested transaction aborts due to parent transaction's abort / total nested transaction aborts) under ten thousand transactions and 80 nodes. The number of nested transactions per transaction are randomly decided. Vacation and Bank benchmarks take longer execution time than other benchmarks, so the abort rate of their nested transactions increases. In high contention, the number of write transactions frequently validate, so the abort rate increases. Under RTS, the abort rate of nested transactions decreases approximately 60%.

C. Transactional Throughput

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 [24]. The purpose of measuring the throughput of TFA is to understand the overall performance improvement of RTS. TFA+Backoff means TFA utilizing a transactional scheduler. With the scheduler, a transaction aborts with a backoff time if a conflict occurs. The purpose of measuring TFA+Backoff's throughput is to understand the effectiveness of enqueuing live transactions to prevent the abort of nested transactions.

Figure 4 shows the transactional throughput at low contention (i.e., 90% read transactions) for each of the six benchmarks, running on 10 to 80 nodes. From Figure 4, 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 5(a) and 5(b), we observe that Vacation and Bank benchmarks take longer execution time than others. The improvement of their transactional throughput is less pronounced.

Figure 5 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 5(c), 5(d), 5(e), and 5(f), the throughput is better than that of Bank and Vacation, because LL, RB Tree, BST, and DHT have relatively short local execution times.

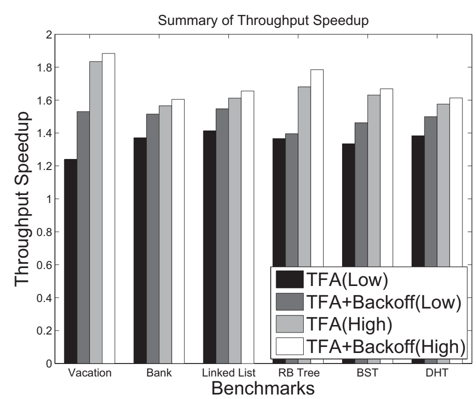
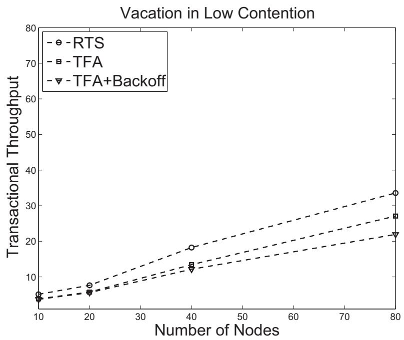
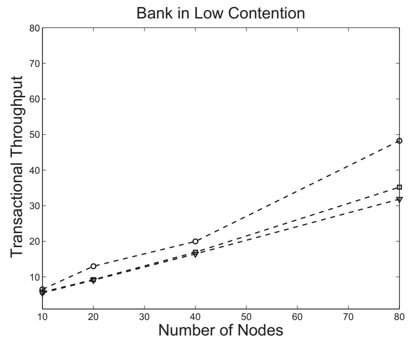


Figure 6. Summary of Throughput Speedup

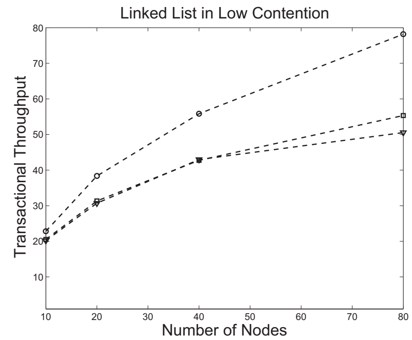
We computed the throughput speedup of RTS over TFA and TFA+Backoff – i.e., the ratio of RTS's throughput to that of the respective competitors. Figure 6 summarizes the speedup. Our experimental evaluations reveal that RTS improves throughput over D-STM without RTS by as much as 1.53 (53%) $\sim$ 1.88 (88%) $\times$ speedup in low and high contention, respectively.



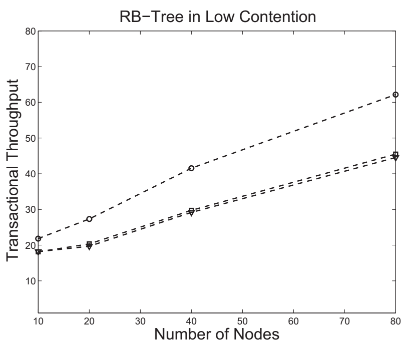
(a) Vacation



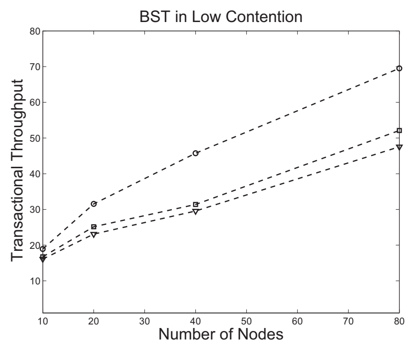
(b) Bank



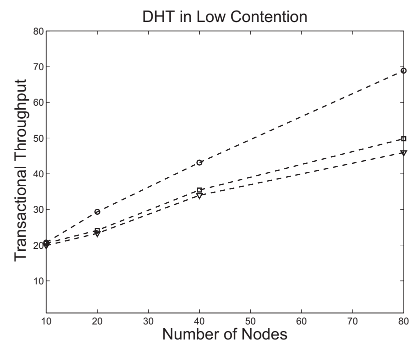
(c) Linked List



(d) RB-Tree

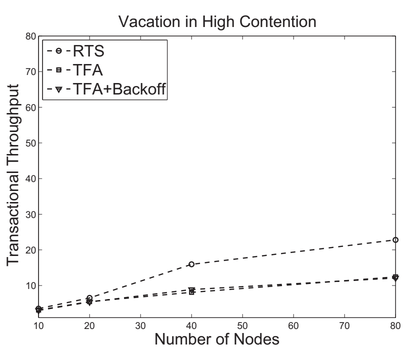


(e) BST

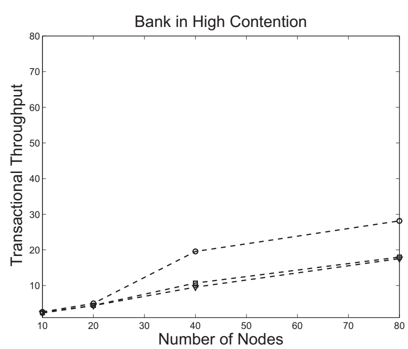


(f) DHT

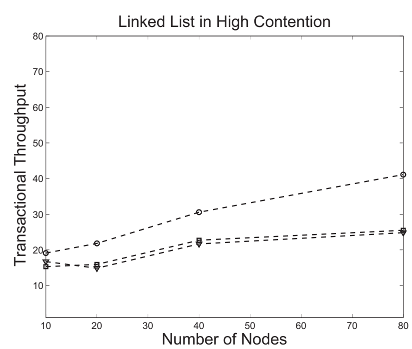
Figure 4. Transactional Throughput on Low Contention



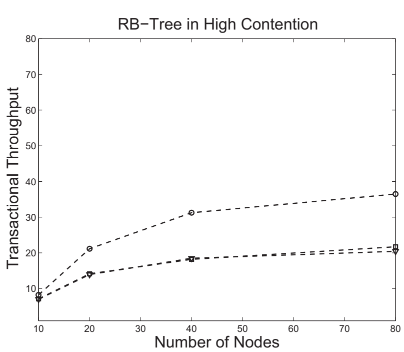
(a) Vacation



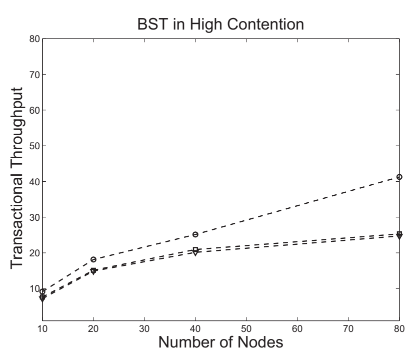
(b) Bank



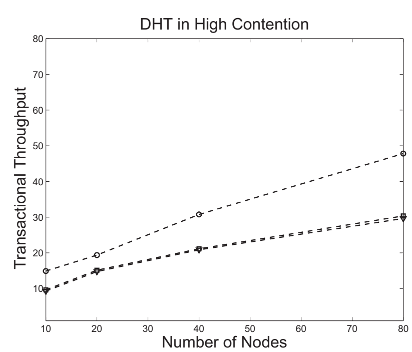
(c) Linked List



(d) RB-Tree



(e) BST



(f) DHT

Figure 5. Transactional Throughput on High Contention

V. Related Work

Transactional scheduling has been explored in a number of multiprocessor STM efforts [9], [2], [25], [8], [3]. In [9], Dragojević *et. al.* describe an approach that schedules transactions based on their predicted read/write access sets. In [2], 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) [25] 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 [8], which uses per-core transaction queues and serializes conflicting transactions by aborting one and queueing it on the other's queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, and thereby minimizes conflicts.

Blake, Dreslinski, and Mudge propose the Proactive Transactional Scheduler (PTS) in [4]. 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 [6] shows PTS outperforming a backoff-based policy by an average of 85%.

Attiya and Milani present the BIMODAL scheduler [3], targeting 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 reading transactions. Kim and Ravindran extend the BIMODAL scheduler for distributed STM [17]. Their scheduler, called Bi-interval, groups concurrent requests into read and write intervals, and exploits the tradeoff between object moving times (incurred in dataflow distributed STM) and concurrency of reading transactions, yielding high throughput.

Steal-On-Abort, CAR-STM, and BIMODAL enqueue aborted transactions to minimize future conflicts. In contrast, RTS enqueues live transactions which conflict with other transactions. The purpose of enqueuing is to prevent closed-nested transactions from restarting. Of course, enqueuing live transactions may lead to deadlock or livelock. Thus, RTS enqueues those live transactions with a low CL and assigns different backoff times for each.

ATS and PTS determine contention intensity and use it for contention management. Unlike ATS and PTS, which are designed for multiprocessor STM, predicting contention intensity will incur communication delays in D-STM. Thus, RTS collects the CL – a history of how many transactions have requested – to measure the contention intensity. The purpose of the CL is not only to manage contention, but also to reduce the retries of nested transactions. Unlike multiprocessor STM, two communication delays will be incurred for a retry, one for requesting an object and the other for retrieving it.

ATS assigns backoff times to aborted transactions. The backoff time indicates when the aborted transactions restart. If a parent transaction aborts, the backoff times may not be effective without considering nested transactions if they exist. RTS focuses on whether a parent transaction is aborted or enqueued. If it is enqueued, RTS gives the transaction a backoff time indicating when it aborts.

VI. Conclusions

Our work illustrates the idea of enqueuing a live parent transaction to prevent its nested transactions from aborting due to a conflict. Doing so will boost transactional throughput by preserving the commits of nested transactions. However, whenever a conflict occurs, enqueuing all live parent transactions does not always improve throughput, because the probability of conflicts also increases. Our transactional scheduler, RTS, determines transactional contention level (heuristically computed) to decide on whether the live parent transaction aborts or is enqueued, and a backoff time that determines how long the live parent transaction waits.

Our experimental evaluation validates our idea: RTS is shown to enhance transactional throughput at high and low contention, by as much as 1.53 (53%) $\sim$ 1.88 (88%) $\times$ speedup, respectively.

VII. Acknowledgements

This work is supported in part by NSF CNS 0915895, NSF CNS 1116190, NSF CNS 1130180, and US NSWC under Grant N00178-09-D-3017-0011.

References

[1] 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.

[2] 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.

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

[4] 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.

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

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

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

[14] Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, New York, NY, USA, 2006. ACM.

[15] 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.

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

[17] 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.

[18] 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.

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

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

[21] 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.

[22] 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.

[23] 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.

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

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