Scheduling Open-Nested Transactions in Distributed Transactional Memory

Junwhan Kim, Roberto Palmieri and Binoy Ravindran

ECE Department, Virginia Tech, Blacksburg, VA, 24061
{junwhan, robertop, binoy}@vt.edu

Abstract. Distributed transactional memory (DTM) is a powerful concurrency control model for distributed systems sparing the programmer from the complexity of manual implementation of lock-based distributed synchronization. We consider Herlihy and Sun's dataflow DTM model, where objects are migrated to invoking transactions, and the *open nesting* model of managing inner (distributed) transactions. In this paper we present DATS, a dependency-aware transactional scheduler, that is able to boost the throughput of open-nested transactions reducing the overhead of running expensive compensating actions and abstract locks in the case of outer transaction aborts. The contribution of the paper is twofold: *(A)* DATS allows the *commutable* outer transactions to be validated concurrently and (B) allows the *non-commutable* outer transactions, depending on their inner transactions, to commit be committed before others without dependencies.

1 Introduction

Transactional Memory (TM) is an emerging innovative programming paradigm for transactional systems. The main benefit of TM is synchronization transparency in concurrent applications. In fact, leveraging the proven concept of atomic and isolated transactions, TM spares programmers from the pitfalls of conventional manual lock-based synchronization, significantly simplifying the development of parallel and concurrent applications. Moreover lock-based concurrency control suffers from programmability, scalability, and composability challenges [13] and TM promises to alleviate these difficulties. In TM, the developer simply organizes read and write operations on shared objects as transactions and leaves the responsibility of executing those transactions to the TM, ensuring atomicity, consistency and isolation. Two transactions conflict if they access to the same object and at least one access is a write. The contention manager, the component in TM responsible for resolving conflicts among concurrent transactions, typically aborts one and allows the other to commit, yielding (the illusion of) atomicity. Aborted transactions are typically re-started after rollingback the changes in memory.

The Transaction Scheduler (TS) is the component that supports the contention manager in making a decision on how to resolve conflicts (which transaction to abort). The goal of the TS is to order concurrent transactions as to avoid or minimize conflicts (and thereby aborts).

The hazards of manual implementation of lock-based concurrency control increases in distributed settings due to an additional synchronization level among nodes in the system. Distributed STM (DTM) has been motivated as an alternative to distributed lock-based concurrency control. DTM can be classified based on the system architecture: cache-coherent DTM (cc DTM) [14, 22], in which a set of nodes communicate by message-passing links over a communication network, and a cluster model (cluster DTM) [6, 20], in which a group of linked computers work closely together to form a single computer. cc DTM uses a cache-coherence protocol [8, 14] to locate and move objects in the network.

Support for nesting (distributed) transactions is essential for DTM, for the same reasons that they are so for multiprocessor TM – i.e., code composability, performance, and fault-management [19, 23, 24]. Three types of nesting have been studied for multiprocessor TM: *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 [18], 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.

The differences between the nesting models are shown in Figure 1, in which there are two transactions containing a nested-transaction. With flat nesting, transaction T_2 cannot execute until transaction T_1 commits. T_2 incurs full aborts, and thus has to restart from the beginning. Under closed nesting, only T_2's inner-transaction needs to abort and be restarted while T_1 is still executing. The portion of work T_2 executes before the data-structure access does not need to be retried, and T_2 can thus finish earlier. Under open nesting, T_1's inner-transaction commits independently of its outer, releasing memory isolation over the shared data-structure. T_2's inner-transaction can therefore proceed immediately, thus enabling T_2 to commit earlier than in both closed and flat nesting.

The flat and closed nested models have a clear negative impact on large monolithic transactions in terms of concurrency. In fact, when a large transaction is aborted all its flat/closed-nested transactions are also aborted and rolled-back, even if they do not conflict with any other transaction. Closed nesting potentially offers better performance than flat nesting because the aborts of closed-nested inner transactions do not affect their outer transactions. However the open-nesting approach outperforms both in terms of concurrency allowed. When an open-nested transaction commits, its modifications on objects become immediately visible to other transactions, allowing those transactions to start using those objects without a conflict, increasing concurrency [19]. In contrast, if the inner transactions are closed- or flat-nested, then those object changes are not made visible until the outer transaction commits, potentially causing conflicts with other transactions that may want to use those objects.

To achieve high concurrency in open nesting, inner transactions have to implement *abstract serializability* [26]. If concurrent executions of transactions result in the consistency of shared objects at an "abstract level", then the execu-

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

If an outer transaction (with open-nested inner transactions) aborts, all of its (now committed) open-nested inner transactions must rollback and their actions must be undone to ensure transaction serializability. Thus, with the open nesting model, programmers must provide a *compensating* action for each open-nested transaction [2]. In scenarios in which outer transactions increasingly encounter conflicts after that a large number of their open-nested transactions have committed (e.g., long running transactions), the overall performance could collapse and all the benefits of the open-nesting approach vanish due to the processing of compensating actions to undo the modifications provided by the committed open-nested transactions. In closed nesting, since closed-nested transactions are not committed in memory until the outer transaction commits (nested transactions' changes are visible only to the outer), no undo (e.g., compensation) is required. Moreover, in scenarios in which the load of the system is high, the probability of having concurrent conflicting transactions grows and with it the probability to abort outer transactions with a number of open-nested transactions (already committed in memory). Aborting outer transactions with dependencies, instead of alleviating the load of the system, will increase it due to the execution of compensating actions, possibly bringing the system toward dangerous states.

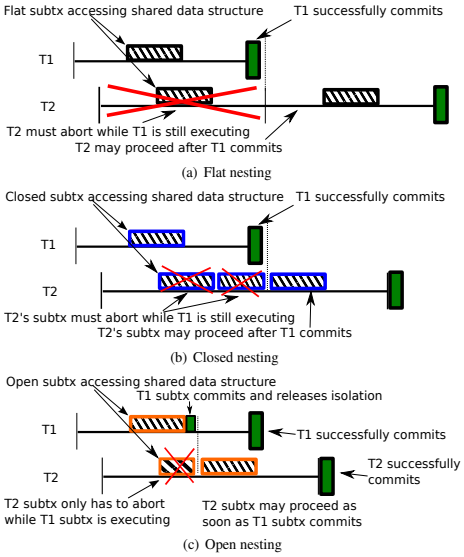


Fig. 1. Two transactions under flat, closed and open nesting (From [23]).

We focus on these problems: the overhead of compensating actions and abstract locks in the open-nested model. Our goal is to boost performance of nested transactions in DTM by increasing concurrency and reducing the aforementioned overhead through transactional scheduling. For these reasons, we designed a scheduler, called the *Dependency-Aware Transactional Scheduler* (or DATS). DATS is responsible for helping the concurrency control minimizing the number of outer-transactions aborted. In order to do that, DATS relies on the notions

of *commutable transactions* and *transaction dependencies*.
Commutable Transactions. Two transactions are defined as commutable if they conflict and they leave the state of the shared data-set consistent even if validated and committed concurrently. A very intuitive example of commutativity is when two operations, $call1(X)$ and $call2(X)$, both access the same object X but different fields of X (See Section 3.2 for discussion about commutativity).
Transaction Dependencies. An outer transaction materializes dependencies with its inner transactions if *(i)* the inner transactions accesses the outer write-set for performing local computation or *(ii)* the results of outer processing are used to decide whether or not to invoke an inner transaction.

DATS is able to detect commutable transactions and validate/commit them, avoiding useless aborts. In the case of non-commutable transactions, DATS identifies how much each outer transaction depends on its inner transactions and schedules the outer transaction with the highest dependency to commit before other outer transactions with lower or no dependencies. Committing this outer transaction prevents its dependent inner transactions from aborting and reduces the number of compensating actions. Moreover, even though the other outer transactions abort, their independent inner transactions will be preserved, resulting in a reduced number of compensating actions and abstract locks without violating the correctness of the object.

We implemented DATS in a Java DTM framework, called HyFlow [21], and conducted an extensive experimental study involving both micro-benchmarks (e.g., Hash Table, Skip-, Linked-List) and a real application benchmark (TPC-C). Our study reveals that throughput is improved by up to 1.7× in micro-benchmarks and up to 2.2× in TPC-C over open-nested DTM without DATS. To the best of our knowledge, DATS is the first ever scheduler that boosts throughput with open-nested transactions in DTM.

The rest of the paper is organized as follows. We present preliminaries of the DTM model and state our assumptions in Section 2. We describe DATS and analyze its properties in Section 3. Section 4 reports our evaluation. We overview past and related efforts in Section 5, and Section 6 concludes the paper.

2 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 network. Similar to [14], we assume that the nodes are scattered in a metric space.
Transaction model. A set of shared objects $O = \{o_1, o_2, \ldots\}$ are distributed in the network among nodes. A transaction is defined as a sequence of requests, each of which is a read or write operation request to an a single object in O. An execution of a transaction is a sequence of timed operations that ends by either a commit (success) or an abort (failure). A transaction is in three possible states: *live*, *aborted*, or *committed*. Each transaction has a unique identifier and is invoked by a node in the system. We consider the data flow DTM model [14]. 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 allowing the local application to interact with the other proxies located on 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.
Atomicity, Consistency, and Isolation. We use the *Transactional Forwarding Algorithm with Open Nesting* (TFA-ON) [23], which extends the TFA algorithm [21] (which originally does not provide any transaction nesting support), to manage flat, closed and open-nested transactions. TFA provides *early validation* of remote objects, guarantees a consistent view of shared objects between distributed transactions, and ensures atomicity for object operations in presence of asynchronous clocks. The early validation of remote objects means that a transaction validated first commits its objects successfully. Validation in distributed systems includes global registration of object ownership. TFA is responsible for caching local copies of remote objects and changing the ownership.

TFA-ON changes the scope of object validations. The behavior of open-nested transactions under TFA-ON is similar to the behavior of regular transactions under TFA. In addition, TFA-ON manages the abstract locks and the execution of commit and compensating actions [23]. To provide conflict detection at the abstract level, an abstract locking mechanism has been 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 [23]. DATS has been integrated in TFA-ON.

3 The DATS Scheduler

3.1 Motivations

Figure 2 shows an example of open-nested transactions with compensating actions and abstract locks. Listings 1.1 and 1.2 in Figure 2 illustrate two outer transactions, T_1 and T_2, and an inner transaction in Listing 1.3. The inner transaction INSERT includes an *insert* operation in a Linked List. T_1 has a *delete* operation with a value. If the operation of T_1 executes successfully, its inner transaction INSERT executes. Conversely, regardless of the success of T_2's *delete* operation, its inner transaction INSERT will execute. *OnCommit* and *OnAbort*, which include a compensating action, are registered when the inner transaction commits. If the outer transaction (i.e., T_1 or T_2) commits, *OnCommit* executes. When the inner transaction commits, its modification becomes immediately visible for other transactions. Thus, if the inner transaction commits, and its outer transaction T_1 or T_2 aborts, a *delete* operation as a compensating action (described in *OnAbort*) executes. Let us assume that T_2 aborts, and *OnAbort* ex-

Listing 1.1. Transaction T_1

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

Listing 1.2. Transaction T_2

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

Listing 1.3. Inner Transaction INSERT

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

Fig. 2. Two open-nested transactions with abstract locks and compensating actions.

ecutes. Even though T_2's inner transaction (INSERT) does not depend on its *delete* operation, unlike T_1, *OnAbort* will execute. Thus, the conflict of object "tree-2" in T_2 causes the execution of compensating action on object "tree-1" in INSERT. The INSERT operation acquires the abstract lock again when it restarts. Finally, whenever an outer transaction aborts, its inner transaction must execute a compensating action, regardless of the operation's dependencies.

This drawback is particularly evident in distributed settings. In fact, distributed transactions typically have an execution time several orders of magnitude bigger than in a centralized STM, due to communication delays that are incurred in requesting and acquiring objects [15]. If an outer transaction aborts, clearly the impact of the time needed for running compensating actions and for acquiring abstract locks for distributed open-nested transactions is exacerbated due to the communication overhead. Moreover it increases the likelihood of conflicts, drastically reducing concurrency and degrading performance.

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

3.2 Abstract and Object Level Dependencies

We consider two types of dependencies among transactions.

Abstract Level Dependency. The first is called *abstract level dependency*

Algorithm 1: Algorithms for checking AOL and OLD

```
1  Procedure Commit
   Input: txid, objects
   Output: commit, abort
2  foreach objects do
3      if txid is open nesting then
4          ▷ Extract <operations,values,DL>
5          Send
           <operations,values,DL,object.id>
6                o object.owner
7          Wait until receive status from
           object.owner
8          if status=noncommute then
9              noncommutativity.put(object);
10 if noncommutativity=∅ then
11     ▷ All objects commute or no conflicts
       detected
12     Retrieve the dependency queue from
       object.owner;
13     Validate objects; ▷ Change the object
       ownership
14     find highest DL from
       dependency.get(object.id);
15     Send object to the node with the highest
       DL;
16     return commit;
17 foreach object ∈ noncommutativity do
18     ▷ Checking abstract level dependency
       (ADL)
19     nestedTxId = CheckALD(object);
20     ▷ Enqueue dependent nested transactions
21     NestedTxs.put(object.id,nestedTxId);
22 Abort(txid, DependentObjects);
23 return abort;
24
25
26 Procedure Retrieve_Object
   Input: operation, value, DL, oid
27 object = findObject(oid);
28 if object=null then
29     ▷ Object just validated, checking object
       level dependency (OLD)
30     if CheckOLD(operation, value) then
31         commutativity.put(object.id, new
           request(operation, values));
32         return commute;
33     ▷ Dependency queue to track updates.
34     dependency.put(oid, DL);
35     return non − commiute;
36 return no − conflict;
37
38 Procedure Abort
   Input: txid, objects
39 if txid is outer-transaction then
40     foreach objects do
41         nestedIds = NestedTxs.get(object.id);
42         if nestedIds ≠ null then
43             foreach nestedIds do
44                 ▷ Execute onAbort() for
                   nestedId
45                 AbortNestedTx(nestedId);
46 AbortOuterTx(txid);
```

(ALD) and it indicates the dependency between an outer transaction and its inner transactions at an abstract level. We define the *dependency level* (DL) as the number of inner transactions that will execute *OnAbort* when the outer transactions abort. For example, T_1 illustrated in Figure 2 depends on its INSERT due to the *deleted* variable. Thus, DATS detects a dependency between T_1 and its INSERT (its inner transaction) because the *delete* operations in T_1 shares the variable *deleted* with the conditional if statement declared for executing INSERT. In this case, the DL=1 for T_1. Conversely, T_2 executes INSERT without checking any pre-condition so its DL=0 because T_2 does not have dependencies with its inner transactions. The purpose of the abstract level dependency is to avoid unnecessary compensating actions and abstract locks. Even though T_2 aborts, *OnAbort* in INSERT will not be executed because its DL=0, and the compensating action will not be processed. Meanwhile, executing *OnAbort* implies running INSERT and acquiring the abstract lock again when T_2 restarts.

Summarizing, aborting outer transactions with smaller DLs leads to a reduced number of compensating actions and abstract lock acquisitions. Such identification can be done automatically at run-time by DATS using byte-code analysis or relying on explicit indication by the programmer. The first scenario is completely transparent from the application point of view but in some cases

could add additional overhead. The second approach, although it requires the collaboration of the developer, is more flexible because it allows the programmer to bias the behavior of the scheduler. In fact, even though the logic of an outer transaction reveals a certain number of dependencies, the programmer may want to force running compensations in case of an abort. This can be done by simply changing the value of DL associated to the outer transaction.

Object Level Dependency. The second is called *object level dependency* (OLD) and it indicates the dependency among two or more concurrent transactions accessing the same shared object. For example, in Figure 2, T_1 depends on T_2 because they share the same object "tree-2". If T_1 and T_2 work concurrently, a conflict between them occurs. However, *delete*(7) of T_1 and *delete*(9) of T_2 commute because they are two operations executing on the same object ("tree-2") but accessing different items (or fields when applicable) of the object (item "7" and item "9"). We recall that, two operations commute if applying them in either order they leave the object in the same state and return the same responses [12]. DATS detects object level dependency at transaction commit phase, splitting the validation phase into two. Say T_a is the transaction that is validating. In the first phase, T_a checks the consistency of the objects requested during the execution. If a concurrent transaction T_b has requested and already committed a new version of some object requested by T_a, then T_a aborts in order to avoid isolation corruption. After the successful completion of the first phase of T_a's validation, DATS detects the object level dependencies among concurrent transactions that are validating with T_a in the second phase. To do that, DATS relies on the notion of commutativity already introduced at the end of Section 1. Suppose T_a and T_b are conflicting transactions but simultaneously validating. If all of T_a's operations commute with all of T_b's operations, they can proceed to commit together avoiding a useless abort. Otherwise one of T_a or T_b must be aborted. This scheduler is in charge of the decision (see next sub-section).

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

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

3.3 Scheduler Design

We designed DATS using abstract level dependencies and object level dependencies. In Figure 1 is presented the pseudo-code with the procedures used by DATS for detecting ALD and OLD at validation/commit time. When outer transactions are invoked, the DL with their inner transactions is checked. When the outer transactions request an object from its owner, the requests with their DLs will be sent to the owner and moved into its scheduling queue. The object owner maintains the scheduling queue holding all the ongoing transactions that have requested the object with their DLs. When T_1 (one of the outer transactions) validates an object, we consider two possible scenarios. First, if another transaction T_2 tries to validate the same object, a conflict between T_1 and T_2 is detected on the object. Thus, DATS checks for the object level dependency. If T_1 and T_2 are independent (according to the object level dependency rules), DATS allows T_1 and T_2 to proceed with the validation. Otherwise, the transaction with lower DL will be aborted. In this way, dependent transactions with the minimal cost of abort and compensating actions are aborted and restarted, permitting transactions with a costly abort operation to commit.

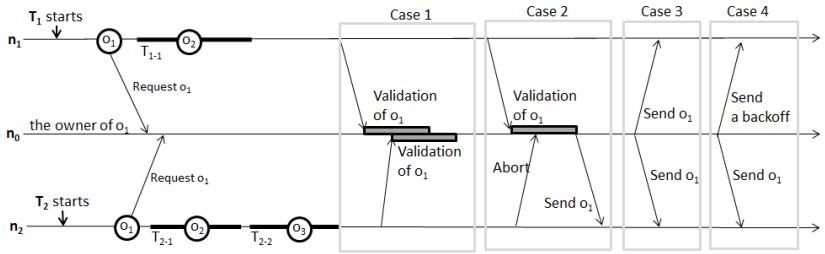


Fig. 3. Four Different Cases for Two Transactions T_1 and T_2 in DATS

Figure 3 illustrates an example of DATS with two transactions T_1 and T_2 invoked on nodes n_1 and n_2, respectively. The transaction T_1 has a single inner-transaction and T_2 has two nested transactions. Let us assume that T_1's DL=1 and T_2's DL=2. The circles indicate written objects. The horizontal line corresponds to the status of each transaction described in the time domain. Figure 3 shows four different cases when T_1 and T_2 terminate. When T_1 and T_2 are invoked, DATS analyzes their DLs, operations, and values. When T_1 requests o_1 from n_0, the meta-data for DLs, operations and values of o_1 will be sent to n_0. These are moved to the scheduling queue of n_0. We consider four different cases regarding the termination of T_1 and T_2.

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

Case 2. T_1 starts to validate and detects it is dependent with T_2 (that is still executing) at the object level on the object o_1. In this case T_2 will abort due to

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

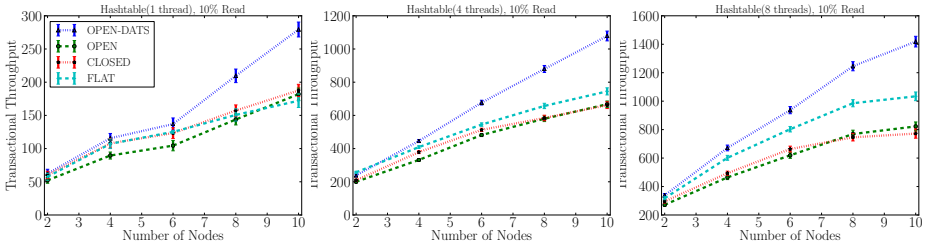
4 Implementation and Experimental Evaluation

Experimental Setup. We implemented DATS in the HyFlow DTM framework [21, 24]. We cannot compare our results with any competitor, as none of the DTMs that we are aware of support open nesting and scheduling. Thus, we compared DATS under TFA-ON (DATS) with only TFA-ON (OPEN) [23], closed nested transaction (CLOSED) [24], and flat nested transaction (FLAT). We contrast with CLOSED and FLAT to show that OPEN does not always perform better than them, while DATS consistently outperforms OPEN.

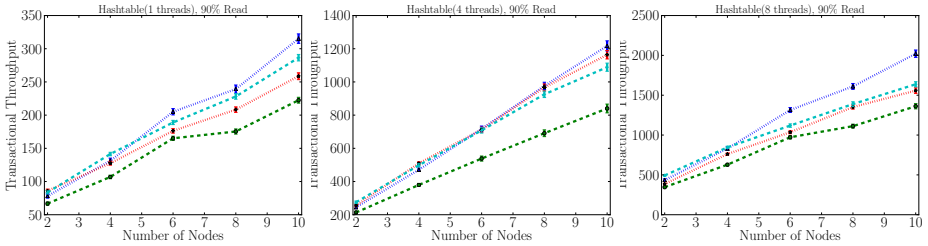
We assess the performance of DATS using Hash Table, Skip List and Linked List as micro-benchmarks, TPC-C [7] as a real-application benchmark. Our testbed is comprised of 10 nodes, each one is an Intel Xeon 1.9GHz processor with 8 CPU cores. We varied the number of application threads performing operations for each node from 1 to 8, considering a spectrum between 2 and 80 concurrent threads in the system. We measured the *throughput* (number of committed transactions per second). All data-points reported are the result of multiple executions, so plots present for each data-point the mean value and the error-bar. In order to assess the goodness of DATS we also present the percentage of aborted transactions and the scheduler overhead.

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

Regarding TPC-C, the write transactions consist of update, insert, and/or delete operations accessing a database of nine tables maintained in memory, where each row has a unique key. Multiple operations commute if they access a row (or object) with the same key and modify different columns. We configured the benchmark with a limited number of warehouses (#3) in order to generate high conflicts. We recall that, in the data flow model, objects are not bound on fixed nodes but move, increasing likelihood of conflicts.



(a) HT (1 thread), 10% Read (b) HT (4 threads), 10% Read (c) HT (8 threads), 10% Read



(d) HT (1 thread), 90% Read (e) HT (4 threads), 90% Read (f) HT (8 threads), 90% Read

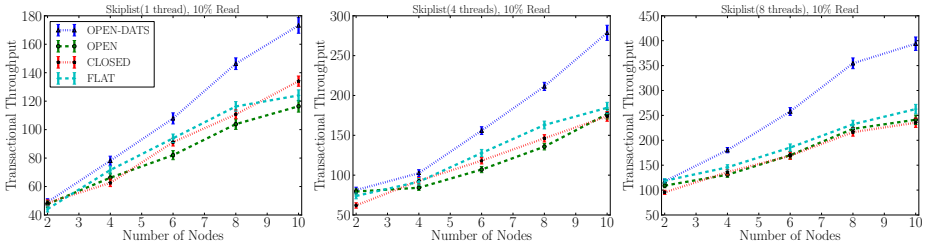
Fig. 4. Performance of DATS Using Hash Table (HT).

Evaluation. Figures 4, 5 and 6(a-f) show the throughput of micro-benchmarks under 10% and 90% of read transactions. The purpose of DATS is to reduce the overheads of compensating actions and abstract locks. In 10% read transactions, the number of aborts increases due to high contention. Outer transactions frequently abort, and corresponding compensating actions are executed; so DATS outperforms OPEN in throughput because it mitigates the abort of outer transactions and the corresponding compensating actions.

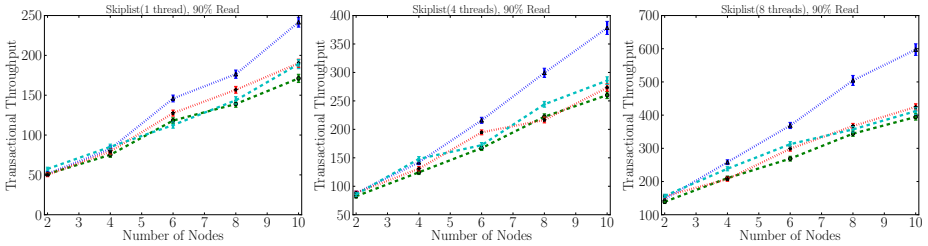
For the experiments with TPC-C in Figure 6(g),6(h),6(i), we used the amount of read and write transactions that its specification recommends. TPC-C benchmark accesses large tables to read and write values. Due to the non-negligible transaction execution time, the number of compensating actions and abstract locks in TPC-C significantly degrades the overall performance. Thus, DATS increases the performance in high contention (a large number of threads and nodes). By these results, it is evident how much unnecessary aborts of inner transactions affects performance and how much performance is improved through minimizing aborts. Even if DATS reduces the number of compensating actions and acquisition of abstract locks, the performance of OPEN is degraded because of the commit overheads of inner transactions [23]; so the throughput of DATS is slightly better than CLOSED and FLAT, but significantly better than OPEN.

Figure 8 shows throughput speed-up relative to OPEN using Hash Table, Skip List, Linked List and TPC-C. Our results show that DATS performs up to 1.7× and 2.2× better than OPEN in micro-benchmarks and TPC-C, respectively.

Figure 7 shows the analysis of scheduling overhead and abort reduction. Checking dependencies occurs when a transaction validates, so we measure the



(a) SL (1 thread), 10% Read (b) SL (4 threads), 10% Read (c) SL (8 threads), 10% Read



(d) SL (1 thread), 90% Read (e) SL (4 threads), 90% Read (f) SL (8 threads), 90% Read

Fig. 5. Performance of DATS Using Skip List (SL).

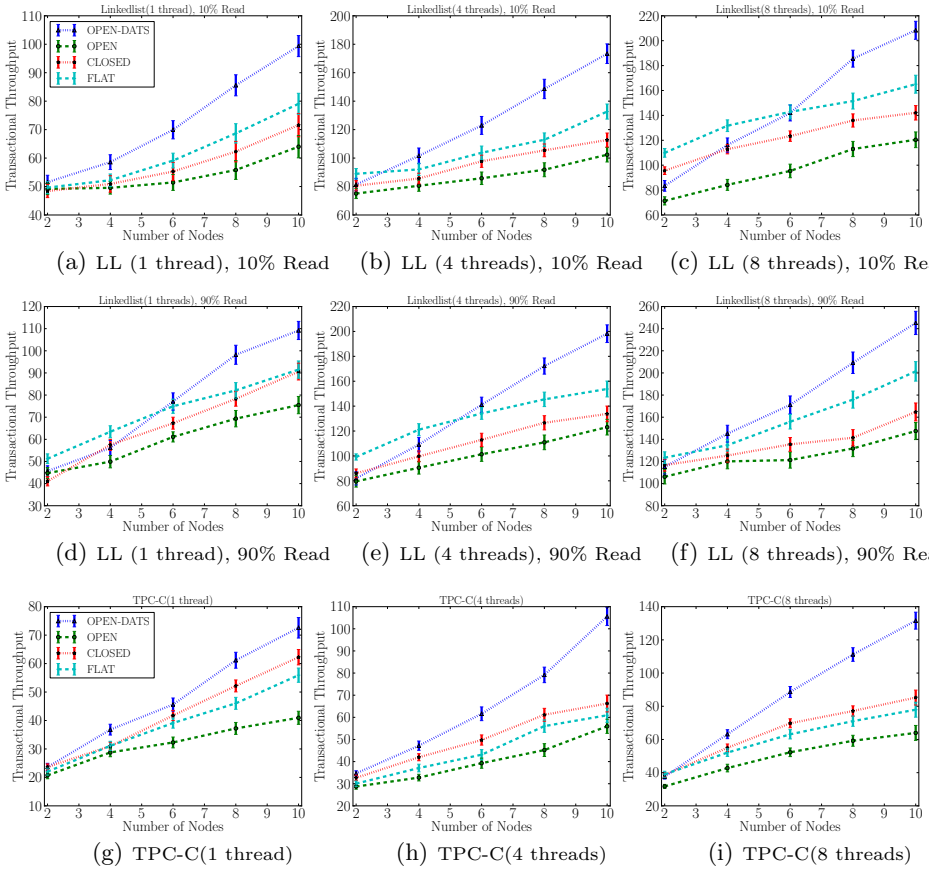
average execution time and the average validation time of committed transactions as illustrated in Figure 7(b). The gap between the two validation times of DATS and OPEN proves the scheduling overhead. Even though the validation time of DATS is up to two times more than OPEN's, a large number of transactions validated simultaneously according to the increment of nodes, results in a shorten transaction response time, reducing the average validation time and aborts. Figure 7(a) the comparison between the percentage of aborted transactions of OPEN and DATS. As long as the number of threads increases, the number of aborts in DATS and OPEN increases too. However, the increasing abort ratio in DATS is less than in OPEN, proving how much DATS reduces the abort rate.

5 Related Work

Nested transactions (using closed nesting) originated in the database community and were thoroughly described in [17]. This work focused on the popular two-phase locking protocol and extended it to support nesting.

Open nesting also originates in the database community [11], and was extensively analyzed in the context of undo-log transactions [25]. 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 has been presented in [19]. They describe the semantics of transactional operations in terms of system states, which are tuples that group together a transaction



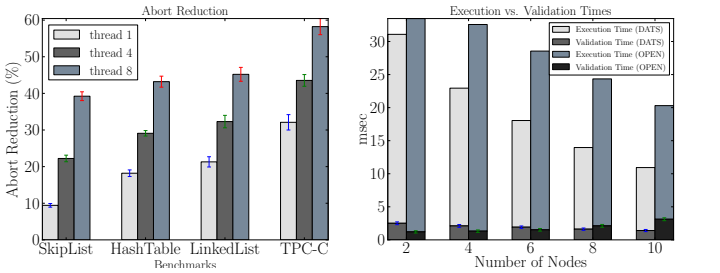
(a) LL (1 thread), 10% Read (b) LL (4 threads), 10% Read (c) LL (8 threads), 10% Read

(d) LL (1 thread), 90% Read (e) LL (4 threads), 90% Read (f) LL (8 threads), 90% Read

(g) TPC-C(1 thread) (h) TPC-C(4 threads) (i) TPC-C(8 threads)

Fig. 6. Performance of DATS Using Linked List (LL) and TPC-C.

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. They further focus on open nested transactions in [18], explaining how using multiple levels of abstractions can help in differentiating between fundamental and false conflicts and therefore improve concurrency. The authors of [16] implemented closed and open nesting in LogTM HTM. They implement nesting models by maintaining a stack of log frames, similar to the run-time activation stack, with one frame for each nesting level. In [1] the authors combined closed and open nesting by introducing the concept of transaction ownership. They propose the separation of TM systems into transactional modules (or Xmodules), which own data. Thus, a sub-transaction commits data owned by its own Xmodule directly to memory using an open-nested model. However, for data owned by foreign Xmodules, it employs the closed nesting model and does not directly write to the memory. The



(a) % Aborted transactions (b) Execution vs. Validation Times

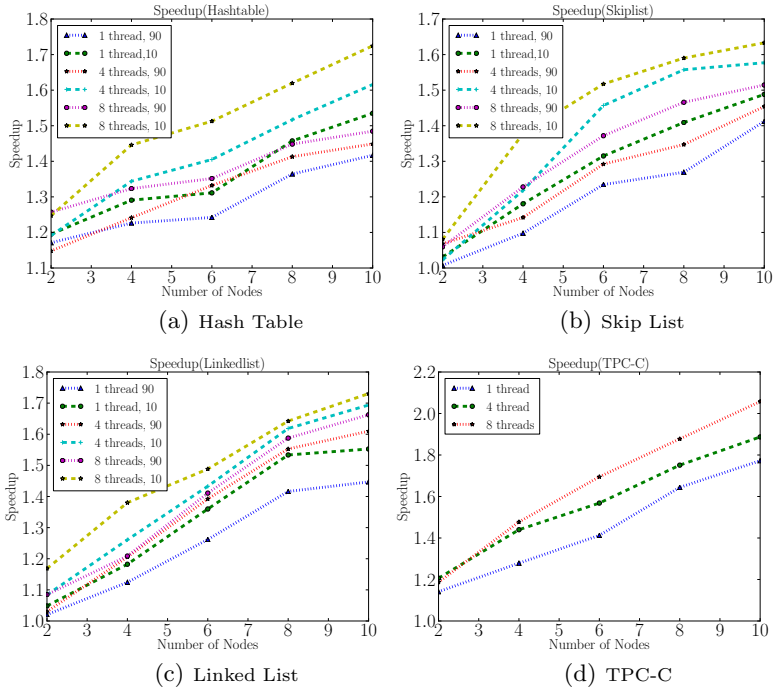
Fig. 7. Analysis of Scheduling Overhead and Abort Reduction.

past closed nesting models [19, 16, 1] have been studied for multiprocessor STM. N-TFA [24] and TFA-ON [23] are the first ever DTM implementation with support for closed and open-nesting, respectively, but do not consider transactional scheduling.

Transactional scheduling has been explored in a number of multiprocessor STM efforts [10, 3, 27, 9, 4]. In [10], the authors describe an approach that schedules transactions based on their predicted read/write access sets. In [3], they 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. The Adaptive Transaction Scheduler (ATS) is present in [27], 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. The CAR-STM scheduling approach is presented in [9], which uses per-core transaction queues and serializes conflicting transactions by aborting one and en-queuing it on another queue, preventing future conflicts. CAR-STM pre-assigns transactions with high collision probability (application-described) to the same core, thereby minimizing conflicts. In [5] they propose the Proactive Transactional Scheduler (PTS). Their scheme detects "hot spots" of contention that can degrade performance, and proactively schedules affected transactions around the hot spots. None of the past transactional schedulers for STM and DTM consider open-nested transactions.

6 Conclusions

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



(a) Hash Table (b) Skip List

(c) Linked List (d) TPC-C

Fig. 8. Speed-up (Throughput Relative to OPEN) in Hash Table, Skip List, Linked List, TPC-C.

shows that DATS enhances transactional throughput for open-nested transactions over no DATS by as much as 1.7× and 2.2× with micro-benchmarks and real-application benchmark, respectively.

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

References

1. Kunal Agrawal, I-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. SPAA, 2008.
2. Kunal Agrawal, Charles E. Leiserson, and Jim Sukha. Memory models for open-nested transactions. MSPC, 2006.
3. Mohammad Ansari, Mikel Lujn, et al. Steal-on-abort: Improving transactional memory performance through dynamic transaction reordering. In *HiPEAC*, 2009.
4. Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. In *OPODIS*, pages 3–17, Berlin, Heidelberg, 2009. Springer-Verlag.
5. G. Blake, R.G. Dreslinski, and T. Mudge. Proactive transaction scheduling for contention management. In *Microarchitecture, 2009.*, pages 156 –167, dec. 2009.

6. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC*, nov 2009.
7. TPC Council. "tpc-c benchmark, revision 5.11". Feb 2010.
8. Demmer and Herlihy. The arrow distributed directory protocol. In *DISC 98*.
9. Shlomi Dolev, Danny Hendler, and Adi Suissa. CAR-STM: scheduling-based collision avoidance and resolution for software transactional memory. In *PODC*, 2008.
10. Aleksandar Dragojević, Rachid Guerraoui, et al. Preventing versus curing: avoiding conflicts in transactional memories. In *PODC '09*, pages 7–16, 2009.
11. Hector Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Trans. Database Syst.*, 8(2):186–213, June 1983.
12. Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. PPoPP '08, pages 207–216. ACM, 2008.
13. Maurice Herlihy, Victor Luchangco, and Mark Moir. A flexible framework for implementing software transactional memory. In *OOPSLA*, pages 253–262, 2006.
14. Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.
15. Junwhan Kim and Binoy Ravindran. Scheduling closed-nested transactions in distributed transactional memory. *In IPDPS*, pages 1–10, 2012.
16. Moravan, Bobba, Moore, Yen, Hill, Liblit, Swift, and Wood. Supporting nested transactional memory in logTM. *SIGPLAN Not.*, 41(11), 2006.
17. E. B. Moss. Nested transactions: An approach to reliable distributed computing. Technical report, Cambridge, MA, USA, 1981.
18. J. Eliot B. Moss. Open-nested transactions: Semantics and support. In *Workshop of Memory Performance Issues*, 2006.
19. J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63:186–201, December 2006.
20. R. Palmieri, F. Quaglia, and P. Romano. Osare: Opportunistic speculation in actively replicated transactional systems. In *SRDS*, 2011.
21. M. Saad and Binoy R. Supporting STM in distributed systems: Mechanisms and a Java framework. In *ACM SIGPLAN Workshop on Transactional Computing '11.*
22. Mohamed M. Saad and Binoy Ravindran. HyFlow: a high performance distributed software transactional memory framework. HPDC '11.
23. Turcu and Ravindran. On open nesting in distributed transactional memory. SYSTOR, 2012.
24. Alex Turcu and Binoy Ravindran. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.
25. Gerhard Weikum. Principles and realization strategies of multilevel transaction management. *ACM Trans. Database Syst.*, 16(1):132–180, March 1991.
26. Yang, Menon, Ali-Reza, Antony, Hudson, Moss, Saha, and Shpeisman. Open nesting in software transactional memory. PPoPP, New York, NY, USA, 2007. ACM.
27. Richard M. Yoo and Hsien-Hsin S. Lee. Adaptive transaction scheduling for transactional memory systems. In *SPAA*, pages 169–178, 2008.