On Open Nesting in Distributed Transactional Memory

Alexandru Turcu
Virginia Tech
talex@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Abstract

Distributed Transactional Memory (DTM) is a recent but promising model for programming distributed systems. It aims to present programmers with a simple to use distributed concurrency control abstraction (transactions), while maintaining performance and scalability similar to distributed fine-grained locks. Any complications usually associated with such locks (e.g., distributed deadlocks) are avoided. Building upon the previously proposed Transactional Forwarding Algorithm (TFA), we add support for open nested transactions. We discuss the mechanisms and performance implications of such nesting, and identify the cases where using open nesting is warranted and the relevant parameters for such a decision. To the best of our knowledge, our work contributes the first ever implementation of a DTM system with support for open nested transactions.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming–distributed programming; D.1.3 [*Programming Techniques*]: Concurrent Programming–parallel programming; D.3.3 [*Programming Languages*]: Language Constructs and Features–concurrent programming structures

General Terms Algorithms, Experimentation, Languages, Performance.

Keywords transactional memory, distributed systems, nested transactions, open nesting

1. Introduction

Transactional Memory (TM) is a promising model for programming concurrency control that is aiming to replace locks. Distributed locks, the traditional solution for concurrency control in distributed systems, can often lead to problems that are much harder to debug than their multiprocessor counterparts. Issues such as distributed deadlocks and livelocks can significantly impact programmer productivity, as finding and resolving the problem is not a trivial task. Moreover, it is easy to accidentally introduce such errors. Additional difficulties arise when code composability is desired, because locks would need to be exposed across composition layers, contrary to the practice of encapsulation. This makes building enterprise software with support for concurrency especially difficult, as such software is usually built using proprietary third-party libraries, often without access to the libraries' source code.

To address these problems, Distributed Transactional Memory (DTM) was proposed as an alternative concurrency control mechanism [9]. DTM systems can be classified by the mobility of the transactions or data. In the data-flow model [9, 15, 22], objects are migrated between nodes to be operated upon by immobile transactions. Alternatively, in the control-flow model [7], objects are immobile and are accessed by transactions using Remote Procedure Calls (RPCs).

In TM, nesting is used to make **code composability** easy. A transaction is called *nested* when it is enclosed within another transaction. Three types of nesting models have been previously studied [7, 13]: flat, closed and open. They differ based on whether the parent and children transactions can independently abort:

Flat nesting
is the simplest type of nesting, and simply ignores the existence of transactions in inner code. All operations are executed in the context of the outermost enclosing transaction, leading to large monolithic transactions. Aborting the inner transaction causes the parent to abort as well (i.e., partial rollback is not possible), and in case of an abort, potentially a lot of work needs to be rerun.

Closed nesting
In closed nesting, each transaction attempts to commit individually, but inner transactions do not write to the shared memory. Inner transactions can abort independently of their parent (i.e., partial rollback), thus reducing the work that needs to be retried, increasing performance.

Open nesting
In open nesting, operations are considered at a higher

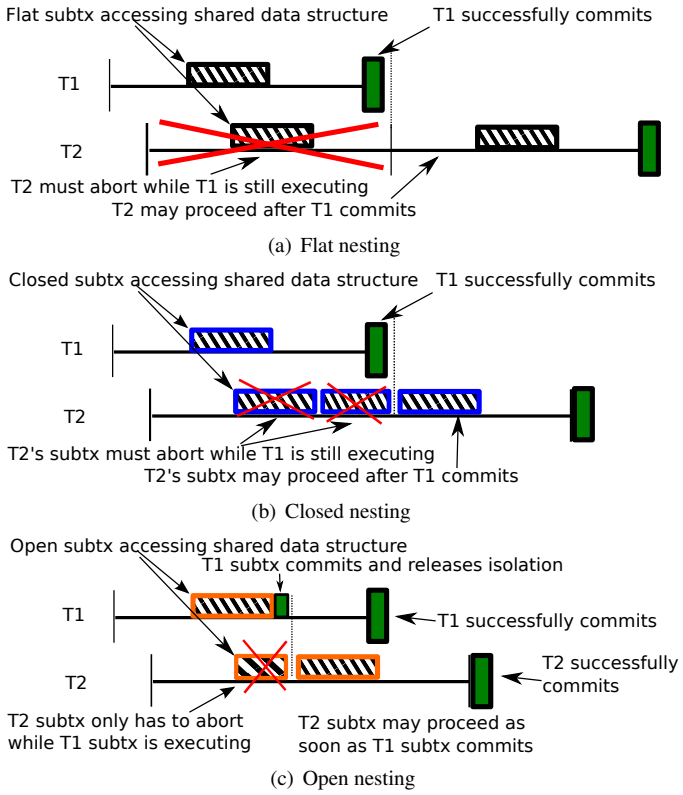


Figure 1. Simple example showing how the execution timeline for two transactions may differ under flat, closed and open nesting.

level of abstraction. Open-nested transactions are allowed to commit to the shared memory independently of their parent transactions, optimistically assuming that the parent will commit. If however the parent aborts, the open-nested transaction needs to run compensating actions to undo its effect. The compensating action does not simply revert the memory to its original state, but runs at the higher level of abstraction. For example, to compensate for adding a value to a set, the system would remove that value from the set. Although open-nested transactions breach the isolation property, this potentially enables significant increases in concurrency and performance.

We illustrate the differences between the three nesting models in Figure 1. Here we consider two transactions, which access some shared data-structure using a sub-transaction. The data-structure accesses conflict at the memory level, but the conflict is not fundamental (we will explain fundamental conflicts later, in Section 3.3), and there are no further conflicts in either T_1 or T_2. With flat nesting, transaction T_2 can not 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 sub-transaction needs to abort and be restarted while T_1 is still executing.

```
@Atomic T popFront() {
  if (this.head == null) retry;
  T result = this.head.value;
  this.head = this.head.next;
  return result;
}
```

Figure 2. Example usage for *retry* construct. Transactions are marked using the @Atomic annotation.

```
@Atomic T chooseFirstAvailable() {
  try { return queue1.popFront(); }
  orElse { return queue2.popFront(); }
}
```

Figure 3. Example usage for *try...orElse* construct.

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 sub-transaction commits independently of its parent, releasing memory isolation over the shared data-structure. T_2's sub-transaction can proceed immediately after that, thus enabling T_2 to commit earlier than in both closed and flat nesting.

Besides providing support for code composability, nested transactions are attractive when transaction aborts are actively used for implementing specific behaviors. For example, **conditional synchronization** can be supported by aborting the current transaction if a pre-condition is not met, and only scheduling the transaction to be retried when the pre-condition is met (for example, a dequeue operation would wait until there is at least one element in the queue, as shown in Figure 2). Aborts can also be used for **fault management**: a program may try to perform an action, and in the case of failure, change to a different strategy (try...orElse, example in Figure 3). In both these scenarios, performance can be improved with nesting by aborting and retrying only the inner-most sub-transaction.

Previous DTM works have largely ignored the subject of partial aborts and nesting [3, 4, 15, 17]. We extend the TFA algorithm [17], which provides atomicity, isolation, and consistency properties for flat-nested DTM transactions, to support open nesting. The resulting algorithm is named Transactional Forwarding Algorithm with Open Nesting(TFA-ON). We also extend the HyFlow Java DTM framework [17] with mechanisms to support open nesting. The transactional operations from TFA (most importantly *commit* and *forward*) are updated for open nesting. Abstract locks, and commit and compensating actions are introduced in HyFlow.

We test our implementation through a series of benchmarks and observe throughput improvements of up to 167% in specific cases. We identify the kinds of workloads that are a good match for open nesting, and we explain how the various parameters influence the gain (or loss) in throughput.

To the best of our knowledge, this work contributes the first ever DTM implementation with support for open nesting.

The remainder of the paper is organized as follows: Section 2 presents related work on nested transactions. The section also overviews the TFA algorithm for completeness. In Section 3, we describe our system model and multi-level transactions. TFA-ON and its mechanisms and implementation are presented as an extension to TFA in Section 4. We report on experimental studies in Section 5. Finally, we conclude the paper in Sections 6.

2. Related work

2.1 Nested Transactions

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [11]. His work focused on the popular two-phase locking protocol and extended it to support nesting. In addition to that, he also proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection.

Open nesting also originates in the database community [6], and was extensively analyzed in the context of undo-log transactions and the two-phase locking protocol [21]. In these works, open nesting is used to decompose transactions into multiple levels of abstraction, and maintain serializability on a level-by-level basis.

One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [13]. They describe the semantics of transactional operations in terms of *system states*, which are tuples that group together a transaction ID, a memory location, a read/write flag, and the value read or written. They also provide sketches for several possible HTM implementations, which work by extending existing cache coherence protocols. Moss further focuses on open nested transactions in [12], explaining how using multiple levels of abstractions can help in differentiating between fundamental and false conflicts and therefore improve concurrency.

Moravan et al. [10] implement closed and open nesting in their previously proposed LogTM HTM. They implement the nesting models by maintaining a stack of log frames, similar to the run-time activation stack, with one frame for each nesting level. Hardware support is limited to four nesting levels, with any excess nested transactions flattened into the inner-most sub-transaction. In this work, open nesting was only applicable to a few benchmarks, but it enabled speedups of up to 100%.

Agrawal et al. combine closed and open nesting by introducing the concept of transaction ownership [2]. They propose the separation of TM systems into transactional modules (or Xmodules), which *own* data. Thus, a sub-transaction would commit data owned by its own Xmodule directly to memory using an open-nested model. However, for data owned by foreign Xmodules, it would employ the closed-nesting model and would not directly write to the memory.

From a different perspective, Herlihy and Koskinen propose transactional boosting [8] as a methodology for implementing highly concurrent transactional data structures. Boosted transactions act as an abstraction above the physical memory layer, being similar to open nesting. They also employ abstract locks for concurrency control in commutative objects. However, boosting works with an existing concurrent data structure (which it treats as a black box), while open nesting is used to implement transactional objects from scratch.

2.2 Transactional Forwarding Algorithm

TFA [16, 18] was proposed as an extension of the Transactional Locking 2 (TL2) algorithm [5] for DTM. It is a data-flow based, distributed transaction management algorithm that provides atomicity, consistency, and isolation properties for distributed transactions. TFA replaces the central clock of TL2 with independent clocks for each node and provides a means to reliably establish the "happens before" relationships between significant events. TFA uses optimistic concurrency control, buffering all operations in per-transaction read and write sets, and acquiring the object-level locks lazily at commit time. Objects are updated once all locks have been successfully acquired. Failure to acquire a lock aborts the transaction, releasing all previously acquired locks.

Each node maintains a local clock, which is incremented upon local transactions' successful commits. An object's lock also contains the object's version, which is based on the value of the local clock at the time of the last modification of that object. When a local object is accessed as part of a transaction, the object's version is compared to the starting time of the current transaction. If the object's version is newer, the transaction must be aborted.

Transactional Forwarding is used to validate remote objects and to guarantee that a transaction observes a consistent view of the memory. This is achieved by attaching the local clock value to all messages sent by a node. If a remote node's clock value is less than the received value, the remote node would advance its clock to the received value. Upon receiving the remote node's reply, the transaction's starting time is compared to the remote clock value. If the remote clock is newer, the transaction must undergo a *transactional forwarding* operation: first, we must ensure that none of the objects in the transaction's read-set have been updated to a version newer than the transaction's starting time (early-validation). If this has occurred, the transaction must be aborted. Otherwise, the transactional forwarding operation may proceed and advance the transaction's starting time.

We illustrate TFA with an example. In Figure 4, a transaction T_k on node N_1 starts at a local clock value $LC_1 = 19$. It requests object O_1 from node N_2 at $LC_1 = 24$, and updates N_2's clock in the process (from $LC_2 = 16$ to $LC_2 = 24$).

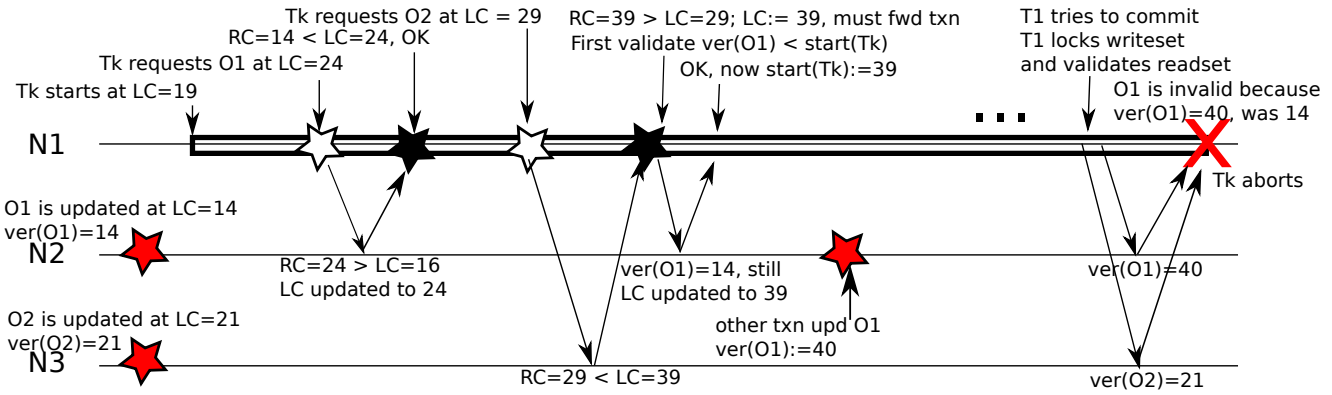


Figure 4. Transactional Forwarding Algorithm Example, from [20]

Later, at time LC_1=29, T_k requests object O_2 from node N_3. Upon receiving N_3's reply, since $RC_3 = 39$ is greater than $LC_1 = 29$, N_1's local clock is updated to $LC_1 = 39$ and T_k is forwarded to $start(T_k) = 39$ (but not before validating object O_1 at node N_2). We next assume that object O_1 gets updated on node N_2 at some later time ($ver(O_1) = 40$), while transaction T_k keeps executing. When T_k is ready to commit, it first attempts to lock the objects in its write-set. If that is successful, T_k proceeds to validate its read-set one last time. This validation fails, because $ver(O_1) > start(T_k)$, and the transaction is aborted (it will retry later — not shown in the figure).

We extended TFA with support for closed nesting in [20]. Our current work continues upon [20].

3. System model

3.1 Base model

As in [9], we consider a distributed system with a set of nodes $\{N_1, N_2, \cdots\}$ that communicate via message-passing links. Let $O = \{O_1, O_2, ...\}$ be the set of objects accessed using transactions. Each object O_j has an unique identifier, id_j. For simplicity, we treat them as shared registers which are accessed solely through read and write methods, but such treatment does not preclude generality. Each object has an owner node, denoted by $owner(O_j)$. Additionally, they may have cached copies at other nodes and they can change owners. A change in ownership occurs upon the successful commit of a transaction which modified the object.

Let $T = \{T_1, T_2, ...\}$ be the set of all transactions. Each transaction has an unique identifier. A transaction contains a sequence of operations, each of which is a read or write operation on an object. An execution of a transaction ends by either a commit (success) or an abort (failure). Thus, transactions have three possible states: active, committed, and aborted. Any aborted transaction is later retried using a new identifier.

3.2 Nesting Model

Our nesting model is based on Moss and Hoskin [13]. While their description uses the abstract notion of system states, we describe our model in terms of concrete read and write-sets, as used in our implementation.

As mentioned before, with non-nested TFA, each transaction maintains a redo-log of the operations it performs in the form of a read-set and a write-set. When an object is read from the globally committed memory, its value is stored in the read-set. Similarly, when an object is written, the actual value written is temporarily buffered in the write-set. Subsequent reads and writes are serviced by these sets in order to maintain consistency: two reads of the same object (not separated by a write within the same transaction) must return the same value. On abort, the sets are discarded and the transaction is retried from the beginning. On commit, the changes buffered in the write-set are saved to the globally committed memory.

With transactional nestings, let $parent(T_k)$ denote the parent (enclosing) transaction of a transaction T_k. A root transaction has $parent(T_k) = \emptyset$. A parent transaction may execute sub-transactions using any of the three nesting models: flat, closed, or open. We denote this by defining the *nesting model* of any sub-transaction T_k:

$$nestingModel(T_k) \in \{CLOSED, OPEN\}$$

Furthermore, root transactions can be considered as a special case of the *OPEN* nesting model.

Let's briefly examine how transaction nesting affects the four important transactional operations. Reading an object O_k first looks at the current transaction's (T_k) read and write-sets. If a value is found, it is immediately returned. Otherwise, depending on the transaction's nesting model, two possibilities arise:

- For $nestingModel(T_k) = OPEN$, the object is fetched from the globally committed memory. This case includes the root transaction.

- For $nestingModel(T_k) = CLOSED$, the read is attempted again from the context of $parent(T_k)$.

Read operations are thus recursive, going up T_k's ancestor chain until either a value is found or an open-nested ancestor is encountered. Write operations simply store the newly written value to the current transaction's write-set.

The read and write-sets of a transaction T_k are denoted by $readset(T_k)$ and $writeset(T_k)$, respectively. The commit of a closed-nested transaction T_k merges $readset(T_k)$ into $readset(parent(T_k))$ and $writeset(T_k)$ into $writeset(parent(T_k))$ [20]. Open-nested transactions commit to the globally committed memory just like root transactions do. They optionally register abort and commit handlers to be executed when the innermost open ancestor transaction aborts or respectively, commits. These handlers are described in Section 4.3.

3.3 Multi-level transactions

We now introduce the concept of multi-level transactions. Consider a data-structure, such as a set implemented using a skip-list. Each node in the list contains several pointers to other nodes, and is in turn referenced by multiple other nodes. When a (successful) transaction removes a value from the skip-list, a number of nodes will be modified: the node containing the value itself, and all the nodes that hold a reference to the deleted value. As a result, other transactions that access any of these nodes will have to abort. This is correct and acceptable if the transactions exist for the sole purpose, and only for the duration of the data-structure access operations. If however, the transactions only access the skip-list incidentally while performing other operations, aborting one of them just because they accessed neighboring nodes in the skip-list would be in vain. Such conflicts are called *false-conflicts*: transactions do conflict at the memory level, as one of them accesses data that was written by the other. However, looking at the same sequence of events from a higher level of abstraction (the remove operation on a set, etc.), there is no conflict because the transactions accessed different items.

It is therefore desirable to separate transactions into multiple levels of abstraction. By making the operations shorter at the lower memory level, isolation at that level is released earlier, thus enabling increased concurrency. This breaches serializability and must be used with care. Multi-level serializability is sufficient in most cases, and can be achieved by reasoning about the commutativity of operations at the higher level of abstraction. Two such operations are conceptually allowed to commute if the final state of the abstract data-structure does not depend on the relative execution order of the two operations [8]. For example, in deleting two different elements from a set, the final state is the same regardless of which of the deletes executes first. In contrast, inserting and deleting the same item from a set can not commute: which of the two operations executes last will determine the state of the set.

In order to achieve level-by-level serialization, non-commutative higher-level operations, when executed by two concurrent transactions, must conflict. Such a conflict is called *fundamental*, as it is essential for a correct execution. One mechanism for detecting fundamental conflicts is by using an *abstract lock*. Two non-commutative operations would try to acquire the same abstract lock. The first one to execute succeeds at acquiring the abstract lock. The second operation would be forced to wait (or abort) until the lock is released. Abstract locks are acquired by open-nested sub-transactions at some point during their execution. When their parent transaction commits, the lock can be released. In case the parent aborts, however, before the lock can be released, the data-structure must be reverted to its original semantic state, by performing compensating actions that undo the effect of the open-nested sub-transaction. Referring back to the set example, to undo the effect of an insertion, the parent would have to execute a deletion in case it has to abort.

3.4 Open nesting safety

Multi-level transactions become ambiguous when open sub-transactions update data that was also accessed by an ancestor. As described by Moss [12], TM implementations have multiple alternatives for dealing with that situation (such as leaving the parent data-set unchanged, updating it in-place, dropping it altogether, and others), which may be confusing for the programmers using those implementations. We thus decide to disallow this behavior in TFA-ON: open sub-transactions may not update objects which were accessed by any of their ancestors, and thereby, impose a clear separation between the multiple transactional levels.

Furthermore, the open nesting model's correctness depends on the correct usage of abstract locking. Should the programmers misuse this mechanism, race conditions and other hard to trace concurrency problems will arise. For these reasons, previous works have suggested that open-nesting be used only by library developers [14] – regular programmers can then use those libraries to take advantage of open-nesting benefits.

4. TFA-ON, Mechanisms and Implementation

We first describe TFA-ON, the algorithmic changes that open nesting imposes on TFA. We then describe key details of its implementation in the HyFlow DTM framework.

4.1 Transactional Forwarding Algorithm with Open Nesting (TFA-ON)

We describe TFA-ON with respect to the TFA algorithm and N-TFA [20], its closed-nesting extension. The low-level details of TFA were summarized in Section 2.2, and we omit them here. In TFA-ON, just as in TFA, transactions are immobile. They are started and executed to completion on the same node. Furthermore, all sub-transactions of a given

```
class Txn {

 // TFA–ON read−set validation routine
 validate() {
  // validate readsets from self until
  // innermost open ancestor
  Txn t = this;
  do {
   if (! t.ReadSet.validate(
          innerOpenAncestor.startingTime ))
    abort(); //validation failed
   t = t.parent;
  } while (t != innerOpenAncestor);
  // validation successful
 }

 forward(int remoteClk) {
  if(remoteClk>innerOpenAncestor.startingTime))
  {validate(); // aborts txn on failure
   innerOpenAncestor.startingTime = remoteClk;
  }
 }

 // TFA–ON commit procedure
 commit() {
  if (nestingModel == OPEN) {
   if ( checkCommit() ) {
    writeSet.commitAndPublish();
    handlers.onCommit();
    parent.handlers += myCommitAbortHandlers;
   } else handlers.onAbort();
  } else if (nestingModel == CLOSED) {
   // merge readSet, writeSet, lockSet and
   // handlers into parent's
  }
 }

 // Called when aborting a transaction due to
 // early−validation/commit failure, etc
 abort() {
  if (! committing)
   handlers.onAbort();
  throw TxnException;
 }

 // acquires locks, validates read−set
 checkCommit() {
  try {
        writeSet.acqLocks();
        lockSet.acqAbsLocks();
        validate();
        return true;
  } catch(TxnException) {
        lockSet.release();
        writeSet.release();
        return false;
  }
 }
```

Figure 5. Simplified source code for supporting Open Nesting in TFA's main procedures.

transaction T_k are created and executed on the same node as T_k.

Open-nested sub-transactions in TFA-ON are similar to top-level, root transactions, in the sense that they commit their changes directly to the shared memory. This affects the behavior of their closed-nested descendants. Under TFA and N-TFA, only the start and commit of root transactions were globally important events. As a result, the node-local clocks were recorded when root transactions started, and the clocks were incremented when root transactions committed. Also, transactional forwarding was performed upon the root transaction itself.

Under TFA-ON, open-nested sub-transactions are important as well: their starting time must be recorded and the node-local clock incremented upon their commit. Closed-nested descendants treat open-nested sub-transactions as a *local root*: they validate read-sets and perform transactional forwarding with respect to the closest open-nested ancestor. Simplified source code of the important TFA-ON procedures is given in Figure 5.

When transactional forwarding is performed, all the read-sets up to the innermost open-nested boundary must be early-validated. Validating read-sets beyond this boundary is unnecessary, because the transactional forwarding operation that is currently underway poses no risk of erasing information about the validity of such read-sets.

4.2 Abstract locks

Additionally, TFA-ON has to deal with abstract lock management and the execution of commit and compensating actions. Abstract locks are acquired only at commit time, once the open-nested sub-transaction is verified to be free of conflicts at the lower level. Since abstract locks are acquired in no particular order and held for indefinite amounts of time, deadlocks are possible. Thus, we choose not to wait for a lock to become free, and instead abort all transactions until the innermost open ancestor. This releases all locks held at the current abstraction level.

We implemented two variants of abstract locking: read-/write locks and mutual exclusion locks. Locks are associated with objects, and each object can have multiple locks. Our data-structure designs typically delegate one object as the *higher level* object, which services all locks for the data-structure, and its value is never updated (thus never causing any low-level conflicts).

4.3 Defining transactions and compensating actions

Commit and compensating actions are registered when an open-nested sub-transactions commits. They are to be ex-

```
new Atomic<Boolean>(NestingModel.OPEN) {
 private boolean inserted = false;
 @Override boolean atomically(Txn t) {
  BST bst = (BST) t.open("tree-1");
  inserted = bst.insert(7, t);
  t.acquireAbsLock(bst, 7);
  return inserted;
 }
 @Override onAbort(Txn t) {
  BST bst = (BST) t.open("tree-1");
  if (inserted) bst.delete(7, t);
  t.releaseAbsLock(bst, 7);
 }
 @Override onCommit(Txn t) {
  BST bst = (BST) t.open("tree-1");
  t.releaseAbsLock(bst, 7);
 }
}.execute();
```

Figure 6. Simplified transaction example for a BST insert operation. Code performing the actual insertion is not shown.

ecuted as open-nested transactions by the innermost open-nested ancestor, when it commits, or respectively, aborts. Closed-nested ancestors simply pass these handlers to their own parents when they commit, but they have to execute the compensating actions in case they abort.

We chose to use anonymous inner classes for defining transactions and their optional commit and compensating actions. Compared to automatic or manual instrumentation, our approach enables rapid prototyping as the code for driving transactions is simple and resides in a single file. Thus, for using open-nested transactions, one only needs to subclass our Atomic<T> helper class and override up to three methods (atomically, onCommit, onAbort). The desired nesting model can be passed to the constructor of the derived class; otherwise a default model will be used. The performance impact of instantiating an object for each executed transaction is insignificant in the distributed environment, where the main factor influencing performance is the network latency.

Figure 6 shows how a transaction would look within our system. Notice how the onAbort and onCommit handlers must request (open) the objects they operate on. They cannot rely on the copy opened by the original transaction, as such a copy may be out-of-date by the time the handler executes (automatic re-open may be a way to address this issue in the future).

4.4 Transaction context stack

Meta-data for each transaction (such as read and write-sets, starting time, etc.) is stored in Transaction Context objects. While originally in HyFlow each thread had its own context object, in order to support nesting, we arrange the context objects in thread-local stacks. Each sub-transaction (closed or open) has a context object on the stack. For convenience, we additionally support flat-nested sub-transactions, which reuse an existing object from the stack instead of creating a new one for the current sub-transaction.

5. Experimental analysis

The goals of our experimental study are finding the important parameters that affect the behavior of open nesting, and based on those, identifying which workloads open nesting performs best in. We evaluate and profile open nesting in our implementation. We quantify any improvements in transactional throughput relative to flat transactions and compare these with the improvements enabled by closed nesting alone. We focus in our study on micro-benchmarks with configurable parameters.

5.1 Experimental settings

The performance of TFA-ON was experimentally evaluated using four distributed micro-benchmarks including three distributed data structures (skip-list, hash-table, binary search tree), and an enhanced counter application.

We ran the benchmarks under flat, closed, and open nesting for a set of parameters. We measured transactional throughput relative to TFA's flat transactions. Each measurement is the average of nine repetitions. Additionally, we quantify how much time is spent under each nesting model executing the various components of a transaction execution:

- Committed/aborted transactions.
- Committed/aborted sub-transactions (closed and open-nesting).
- Committed/aborted compensating/commit actions (open-nesting only).
- Waiting time after aborted (sub-)transactions (for exponential back-off).

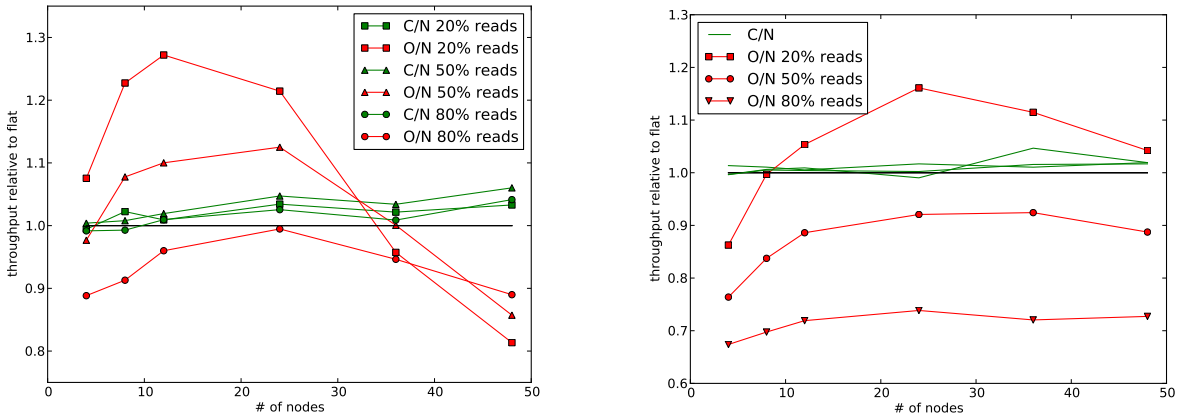
Other data that we recorded includes:

- Number of objects committed per (sub-)transaction.
- Which sub-transaction caused the parent transaction to abort.

Unfortunately, we cannot compare our results with any competitor DTM, as none of the two competitor DTM frameworks that we are aware of support open nesting [3, 4].

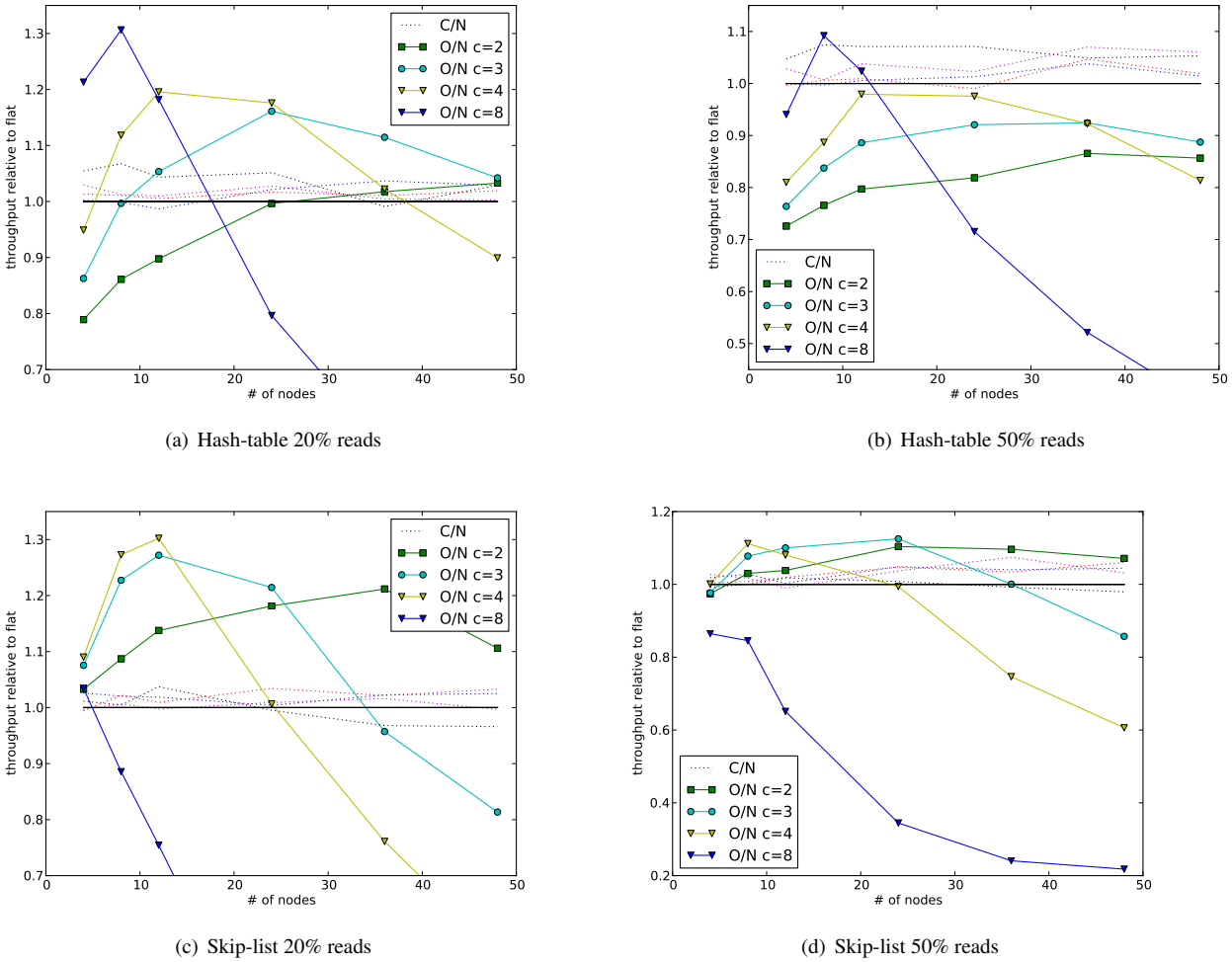
The skip-list, hash-table, and BST benchmarks instantiate three objects each, and then perform a fixed number of random set operations on them using increasing number of nodes. Three important parameters characterize these benchmarks:

- Read-only ratio (r) is the percentage of the total transactions which are read-only. We used $r \in \{20, 50, 80\}$.



(a) Skip-list (b) Hash-table

Figure 7. Performance relative to flat transactions, with $c = 3$ calls per transaction and varying read-only ratio. Both closed-nesting and open-nesting are included.



(a) Hash-table 20% reads (b) Hash-table 50% reads

(c) Skip-list 20% reads (d) Skip-list 50% reads

Figure 8. Performance relative to flat transactions at a fixed read-ratio with varying number of calls. Closed-nesting is depicted, but the individual curves are not identified to reduce clutter.

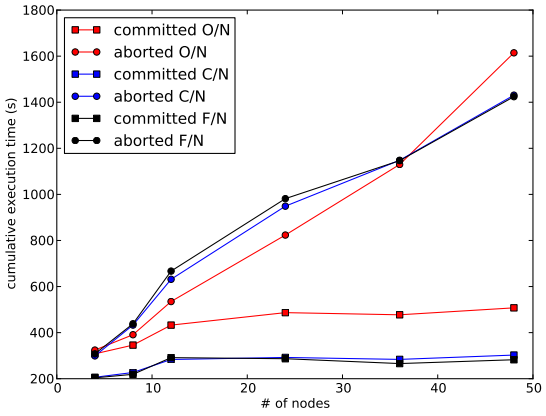


Figure 9. Time spent in committed vs. aborted transactions, on hash-table with $r = 20$ and $c = 4$. Lower lines (circle markers) represent time spent in committed transactions, while the upper lines (square markers) represent the total execution time. The difference between these lines is time spent in aborted transactions.

- Number of calls (c) controls the number of data-structure operations performed per test. Each operation is executed in its own sub-transaction. We used $c \in \{2, 3, 4, 8\}$.
- Key domain size (k) is the maximum number of objects in the set. Lower k values lead to increased fundamental conflicts. Unless otherwise stated, we used $k = 100$.

The fourth benchmark (*enhanced counter*) was designed as a targeted experiment where the access patterns of a transaction are completely configurable. Transactions access counter objects which they read or increment. Transactions are partitioned into three stages: the preliminary stage, the sub-transaction stage, and the final stage. The first and last stages are executed as part of the root transaction, while the middle runs as a sub-transaction. Each stage accesses objects from a separate pool of objects. The number of objects in the pool, the number of accesses, and the read-only ratio are configurable for each stage. We also enable operation without acquiring abstract locks, thus emulating fully commutative objects.

Our experiments were conducted on a 48-node testbed. Each node is an AMD Opteron processor clocked at 1.9GHz. We used the Ubuntu Linux 10.04 server OS and a network with $1ms$ end-to-end link delay.

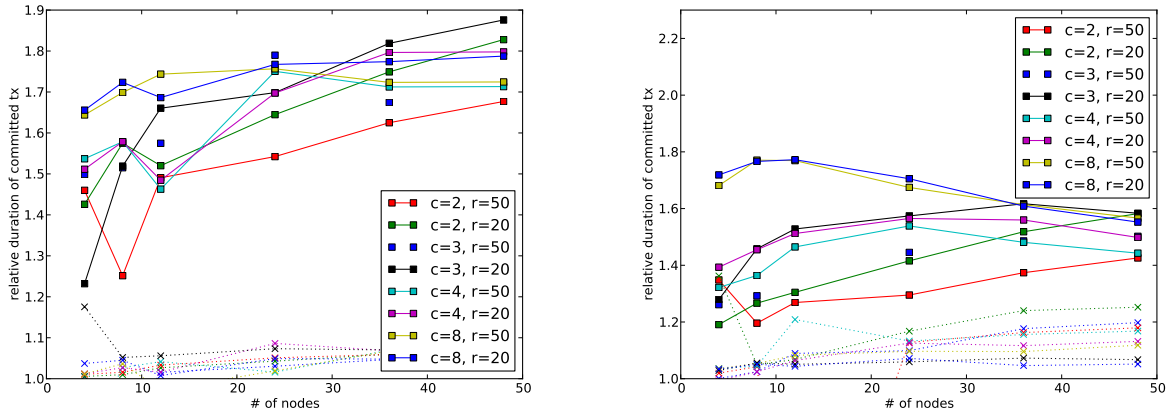
5.2 Experimental results

For all the data-structure micro-benchmarks, we observed that open-nesting's best performance improvements occur at low read-only ratio workloads. For brevity, we only focus on skip-list and hash-table in this paper. The BST plots, and other plots that we considered redundant for inclusion in the paper, can be found in the technical report available at our website [19]. Figure 7 shows how open-nesting throughput climbs up to a maximum and then falls off faster than either flat or closed nesting as contention increases due to more nodes accessing the same objects. Figure 7 also shows the effect that read-only ratio has on the throughput. It is noticeable that on read-dominated workloads, open-nesting actually degraded performance. Closed-nesting constantly stayed in the 0-10% improvement range throughout our experiments (closed nesting behavior is uninteresting and will henceforth be either omitted from the plots or shown without identification markers to reduce clutter).

Focusing on write-dominated workloads ($r = 20$ and $r = 50$), Figure 8 shows how the maximum performance benefit of open nesting generally increases as the number of sub-transactions increases. For more sub-transactions however, the benefit of open nesting occurs at fewer nodes and falls off much faster with increasing number of nodes. The maximum improvements we have observed (with reduced key-domain, $k = 100$) are 30% on skip-list with $r = 20$ and $c = 4$, 31% on hash-table with $r = 20$ and $c = 8$, and 29% on BST with $r = 20$ and $c = 8$ [19]. On skip-list it is noticeable that at high contention ($c = 8$) the region of maximum benefit disappears altogether and the performance decreases monotonously.

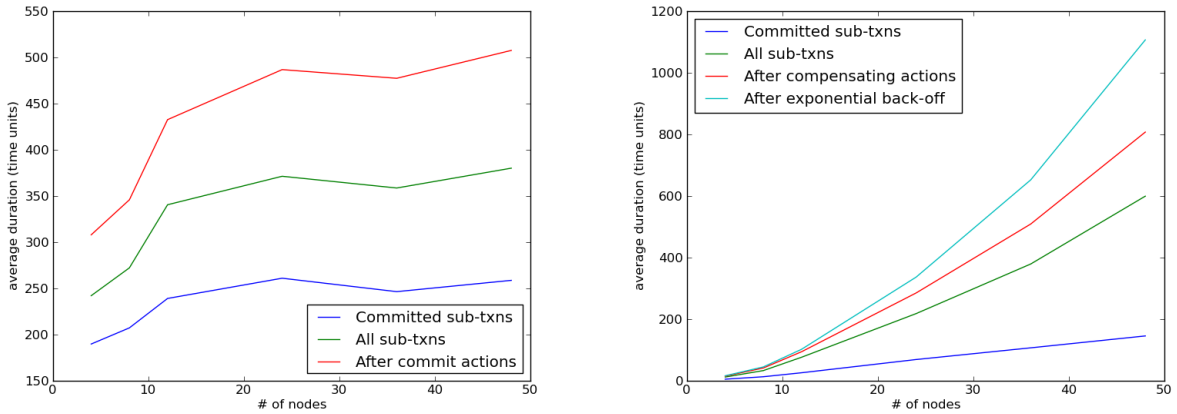
These observations can be explained by examining how is the time spent when using open-nesting. Figure 9 shows how the time taken by successfully committed transactions under open nesting and closed nesting increases at a similar rate. However, open nesting has a significant overhead, caused by the increased rate of commits. This effect is more pronounced in read-dominated workloads, where object updates are rare, and as a result, read-set early-validations under flat-nesting are also rare (early-validations are performed when a commit is detected at another node). In open-nesting however, the read-set must be validated for every sub-transaction commit, thus adding multiple network accesses to the cost of successful transactions. Figure 10 shows that the average overheads of open-nesting relative to flat transactions (50-80% on hash-table and 40-50% on skip-list) are significant and higher than that of closed nesting (3-7% on hash-table and 5-16% on skip-list). We observe the overheads are benchmark dependent, and are lower for workloads which access more objects in every sub-transaction. This is apparent when comparing Figures 10(b) and 10(a), and further experiments we have performed with higher nodal levels on skip-list [19] confirm our observation.

On the other hand, the time taken by aborted transactions in open-nesting (Figure 9) is much lower at low node-counts, but increases rapidly for higher node-counts. Examining the average time taken by the various stages of a single transaction (Figures 11(a) and 11(b)), we see that the duration of a single transaction (committed or aborted) does increase with increasing number of nodes, but this increase is relatively small. Moreover, individual failed transactions consistently take less time than committed ones. Thus, the rapid increase in total time taken by aborted transactions (and therefore



(a) Hash-table (b) Skip-list

Figure 10. Overhead of successful open-nested transactions. Plotted is the relative ratio of the average time taken by successful open-nested transactions to the average time taken by successful flat transactions. Closed-nested transactions are also shown, with dotted markers and without identification.

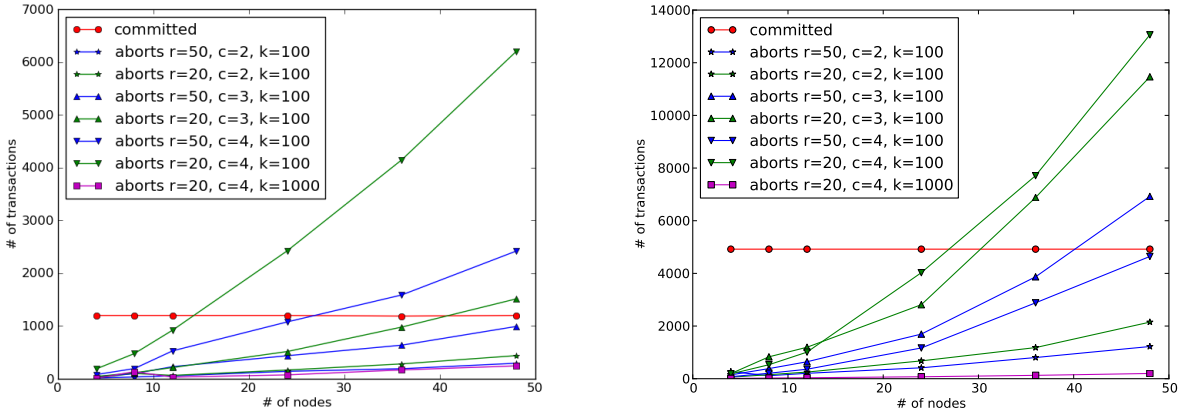


(a) Committed transactions (b) Aborted transactions due to abstract lock acquisition failure

Figure 11. Breakdown of the duration of various components of a transaction under open-nesting, on hash-table with $r = 20$ and $c = 4$.

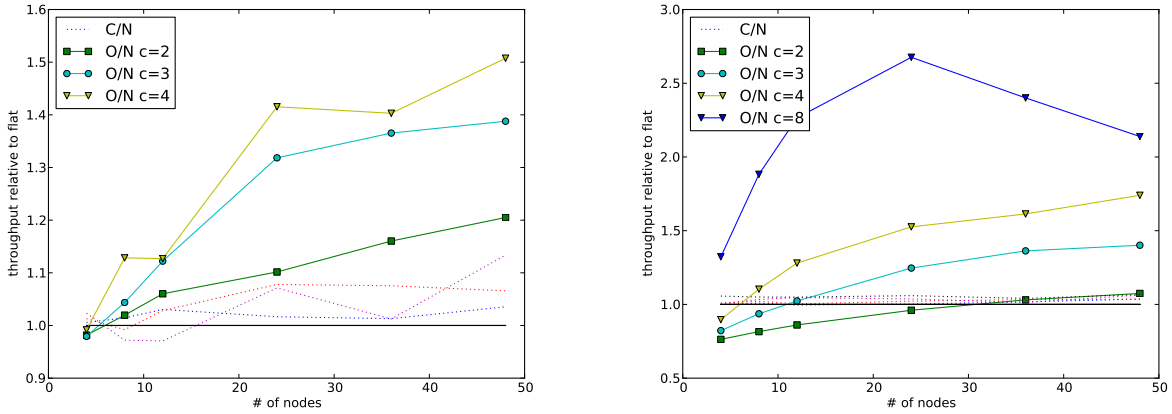
a decrease in overall throughput) can only be explained if there is a significant increase in the number of aborts. The data upholds this hypothesis, as shown in Figure 12. Note that in our data-structure benchmarks under open-nesting, all transaction (full) aborts are caused by abstract lock acquisition failure. Abstract locks are acquired eagerly (on-access) and thus when fundamental conflicts are frequent, will cause more aborts and lower performance compared to TFA's deferred lock acquisition.

Intuitively, the number of aborts is lower when there are fewer sub-transactions competing for the same number of locks, or when the number of available abstract locks is increased. These effects are also illustrated in Figure 12. Increasing the number of calls leads to a rapid increase in the number of aborts. However, the key space k has a more pronounced effect. Setting $k = 1000$ reduced the frequency of fundamental conflicts and abstract lock contention. As a result, the number of aborts as compared to other configurations in Figure 12 became negligible, and thus the performance increase of open nesting is more stable and more significant than for the cases we previously discussed. In Figure 13, we show throughput increase up to 51% on Skip-list (at $c = 4$ and $r = 20$) and up to 167% on Hash-table (at $c = 8$ and $r = 20$). Benefits for open-nesting become possible even in non-write-dominated workloads: with $c = 3$ on skip-list, we have found 12% improvement at $r = 80$ and 21% improvement at $r = 50$ [19].



(a) Hash-table (b) Skip-list

Figure 12. Number of aborted transactions under open-nesting, with various parameters. The figure shows the effect of read-only ratio, number of calls, and key domain size. Note that all aborts depicted in this plot are full aborts due to abstract lock acquisition failure. The number of committed transactions is fixed for each experiment.



(a) Skip-list (b) Hash-table for r=20

Figure 13. Throughput relative to flat nesting with increased key space $k = 1000$ and write-dominated workloads $r = 20$.

In our enhanced counter micro-benchmark we observed improvements consistent with our previous findings (plot in [19]). However, these improvements only manifested if the root transaction does not experience significant contention after the open-nested sub-transaction commits. Any increase in contention at this stage quickly leads to performance degradation. This result is in agreement with the theory, because by releasing isolation early, open-nesting optimistically assumes the parent will commit. Increased contention after the open-nested sub-transaction comes against this assumption.

In the context of this benchmark we also briefly experimented with fully commutative objects, by not acquiring abstract locks at all. For our particular case, this resulted in a further 20-30% performance benefit for open-nesting. Better improvements are however entirely possible if the post-sub-transaction contention is even lower (in our test, a majority of aborts were caused by post-sub-transaction contention).

6. Conclusions

We presented TFA-ON, an extension of the Transactional Forwarding Algorithm that supports open nesting in a Distributed Transactional Memory system. We implemented TFA-ON in the HyFlow DTM framework, thus providing (to the best of our knowledge) the first-ever DTM implementation to support open nesting. Our TFA-ON implementation enabled up to 30% speedup when compared to flat transactions, for write-dominated workloads and increased

fundamental conflicts. Under reduced fundamental conflicts workloads, speedup was as high as 167%.

We determined that open nesting performance is limited by two factors: commit overheads and fundamental conflict rate. Fundamental conflicts limit the scalability of open nesting at higher node-counts, and depend on the available key space for abstract locking. Commit overheads determine the baseline performance of open nesting, at lower node counts, under reduced contention. Commit overheads are significant under read-dominated workloads, and are also influenced by the number of objects accessed in sub-transactions. Furthermore, we confirm that open nesting does not apply well to workloads which incur significant contention after the open-nested sub-transaction commits.

References

[1] *24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings*, 2010. IEEE.

[2] K. Agrawal, I.-T. A. Lee, and J. Sukha. Safe open-nested transactions through ownership. In D. A. Reed and V. Sarkar, editors, *PPOPP*, pages 151–162. ACM, 2009. ISBN 978-1-60558-397-6.

[3] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *IPDPS* DBL [1], pages 1–12.

[4] N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *NCA*, pages 271–274. IEEE Computer Society, 2011. ISBN 978-1-4577-1052-0.

[5] D. Dice, O. Shalev, and N. Shavit. Transactional locking ii. In S. Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2006. ISBN 3-540-44624-9.

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

[7] T. Harris, J. R. Larus, and R. Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[8] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In S. Chatterjee and M. L. Scott, editors, *PPOPP*, pages 207–216. ACM, 2008. ISBN 978-1-59593-795-7.

[9] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In P. Fraigniaud, editor, *DISC*, volume 3724 of *Lecture Notes in Computer Science*, pages 324–338. Springer, 2005. ISBN 3-540-29163-6.

[10] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in logtm. In J. P. Shen and M. Martonosi, editors, *ASPLOS*, pages 359–370. ACM, 2006. ISBN 1-59593-451-0.

[11] J. E. B. Moss. Nested transactions: An approach to reliable distributed computing, 1981.

[12] J. E. B. Moss. Open nested transactions: Semantics and support (poster). In *Workshop on Memory Performance Issues*, Feb 2006.

[13] J. E. B. Moss and A. L. Hosking. Nested transactional memory: Model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, 2006.

[14] Y. Ni, V. Menon, A.-R. Adl-Tabatabai, A. L. Hosking, R. L. Hudson, J. E. B. Moss, B. Saha, and T. Shpeisman. Open nesting in software transactional memory. In K. A. Yelick and J. M. Mellor-Crummey, editors, *PPOPP*, pages 68–78. ACM, 2007. ISBN 978-1-59593-602-8.

[15] P. Romano, L. Rodrigues, N. Carvalho, and J. P. Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *Operating Systems Review*, 44(2):1–6, 2010.

[16] M. M. Saad. Hyflow: A high performance distributed software transactional memory framework. Master's thesis, Virginia Tech, April 2011.

[17] M. M. Saad and B. Ravindran. Supporting stm in distributed systems: Mechanisms and a java framework. In *TRANSACT (ACM SIGPLAN Workshop on Transactional Computing)*, San Jose, California, USA, June 2011.

[18] M. M. Saad and B. Ravindran. Transactional forwarding algorithm. Technical report, Virginia Tech, January 2011.

[19] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. Technical report, Virginia Tech, 2012. URL `http://hyflow.org/hyflow/chrome/site/pub/opennesting-systor12-tech.pdf`.

[20] A. Turcu, B. Ravindran, and M. M. Saad. On closed nesting in distributed transactional memory. In *TRANSACT*, 2012.

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

[22] B. Zhang and B. Ravindran. Dynamic analysis of the relay cache-coherence protocol for distributed transactional memory. In *IPDPS* DBL [1], pages 1–11.