On Open Nesting in Distributed Transactional Memory

Alexandru Turcu, Roberto Palmieri and Binoy Ravindran

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. In this article, we analyze the use of open nesting in the DTM setting. We extend two DTM algorithms, Transactional Forwarding Algorithm (TFA) and SCORe with support for open nested transactions and we implement them into two frameworks for running distributed transactions, such as Hyflow and Infinispan. 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 also contributes the first ever implementations of DTM systems with support for open-nested transactions.

Index Terms—Nesting, Open Nesting, Distributed Transactions, Transactional Memory.

1 Introduction

Transactional Memory (TM) [12] 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 [13]. DTM systems can be classified by the mobility of the transactions or data. In the data-flow model [13], objects are migrated between nodes to be operated upon by immobile transactions. Alternatively, in the control-flow model [21], objects are immobile and are accessed by transactions using Remote Procedure Calls.

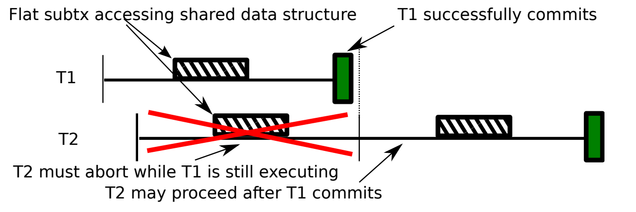
In TM, nesting is used to make **code composability** easy. A transaction is called *nested* when it is enclosed within another transaction. Three types of nesting models have been previously studied [19]: flat, closed and open. They differ based on whether the parent and children transactions can independently 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.

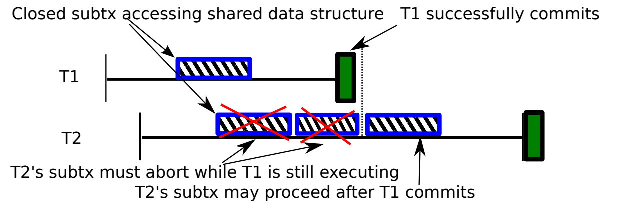
Closed nesting. With closed nesting, each transaction attempts to commit individually, but inner transactions do not publicize their writes to the globally committed memory. Inner transactions can abort independently of their parent (i.e., partial rollback), thus reducing the work that needs to be retried,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

Open nesting. With open nesting, operations are considered at a higher level of abstraction. Open-nested transactions are allowed to commit to the globally committed 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. Open-nested transactions breach the isolation property, thus potentially enabling significant increases in concurrency and performance. However, to be used correctly, logical isolation is still generally required, and the burden for ensuring it now falls on the programmers.

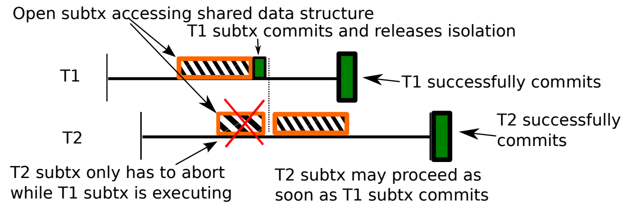
• *Authors are with the Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, VA, 24061. Alexandru Turcu is now at Google.*


Flat subtx accessing shared data structure
T1 successfully commits
T1
T2
T2 must abort while T1 is still executing
T2 may proceed after T1 commits

(a) Flat nesting


Closed subtx accessing shared data structure
T1 successfully commits
T1
T2
T2's subtx must abort while T1 is still executing
T2's subtx may proceed after T1 commits

(b) Closed nesting


Open subtx accessing shared data structure
T1 subtx commits and releases isolation
T1
T1 successfully commits
T2
T2 successfully commits
T2 subtx only has to abort while T1 subtx is executing
T2 subtx may proceed as soon as T1 subtx commits

(c) Open nesting

Fig. 1. Example showing the execution time-line for two transactions under flat, closed and open nesting.

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 it is not a semantic conflict (a semantic conflict arises when two operations on a data-structure lead to different outcomes when commuted, see 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. 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. This example assumes the TM implementation aborts the minimum amount of work required to resolve the conflict, thus leading to the maximum performance for each nesting model (in practice, this is accomplished by validating the read operations and determining the minimal set of transactions that should be aborted).

As a practical example of the usefulness of open nesting, we can consider a sorted list of linked elements. The list is a data structure defining commu-

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

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

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

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

tative operations, which means that concurrent invocations of the same operation (e.g., `add`) on different elements are allowed to complete in parallel. Assume now a transaction performing two `add` operations atomically. With open nesting, each operation can be executed in a separate sub-transaction so that all the elements visited for reaching the point in the list where the `add` should be performed will be discarded once the sub-transaction commits. By doing so we save possible aborts due to the invalidation on those elements while the transaction executes the remaining operations (e.g., the subsequent `add`). Adopting the closed nesting model, those aborts cannot be avoided.

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, 22]. While they were studied in non-distributed TM, the cost of network access in a distributed environment significantly affects the behavior of nested transactions. We develop a framework for extending existing DTM algorithms with support for open nesting, and apply it to two different transaction execution protocols. Specifically, we extend the TFA algorithm [22], 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). The second algorithm we extend is SCORe [21], a scalable one-copy serializable partial replication protocol with multi-version concurrency control (MVCC), which provides One-Copy Serializability and executes read-only transactions without requiring a distributed commit protocol. We call the resulting algorithm SCORe-ON. We discuss the implications of open nesting on read-only and read-write transactions.

These two algorithms were chosen as they cover a significant portion of the design space while providing strong consistency. TFA is a non-replicated, dataflow based, single-version protocol which potentially revalidates transactions after every read. SCORe is a partial replication, control-flow based, multi-version protocol that is able to commit read-only transactions without further network communication. Thus we can show how open nesting is applicable to a wide range of transactional algorithms.

We implement our extensions across two frameworks: HyFlow [22, 22] and Infinispan [15]. To this goal, we introduce new mechanisms such as abstract locks, and commit and compensating actions in both HyFlow and Infinispan. Our choice of frameworks was again made strategically. On the one hand, Hyflow is our in-house DTM framework research prototype, was designed with support for nesting in mind, and collects a variety of metrics to help gain insight into the transaction behavior during execution. On the other hand, Infinispan is a popular open-source in-memory data-grid, with support for distributed transactions. Infinispan is highly configurable, extensible but also complex, is supported commercially and is used in production world-wide.

We test our implementation through a series of benchmarks, which includes micro-benchmarks and two commercial inspired 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 DTM implementations with support for open nesting[1]. Source code is publicly available at hyflow.org.

The remainder of the paper is organized as follows: Section 2 presents related work on nested transactions. The section also overviews the TFA and SCORe algorithms for completeness. In Section 3, we describe our system model and multi-level transactions. Our open nesting framework, TFA-ON and SCORe-ON are presented in Section 4. Mechanisms and implementation details are described in Section 5. We report on experimental studies in Section 6. Finally, we conclude the paper in Sections 7.

1. A preliminary version of this paper appeared in [23]

2 Related work

2.1 Nested Transactions

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [17]. His work focused on the popular two-phase locking protocol and extended it to support nesting. In addition, he also proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection.

Open nesting also originates in the database community [8], and was extensively analyzed in the context of undo-log transactions and the two-phase locking protocol [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 was done by Moss and Hosking in [19]. 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 [18], explaining how using multiple levels of abstractions can help differentiate between semantic conflicts and other conflicts, thus improving concurrency.

Moravan et al. [16] 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 [1]. 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 [11] as a methodology for implementing highly concurrent transactional data structures. Boosted transactions act as an abstraction above the physical memory layer, internally employing open nesting and abstract locks.

In [6] the open nesting model has been extended by Dhoke et al. to avoid the blocking nature of sub-transaction's commit by introducing an asynchronous global commit, which proceeds in parallel with a local speculative execution. This way, part of the overhead of open nesting can be alleviated and the saturation of the network does not represent a blocking factor for the local computation.

2.2 Transactional Forwarding Algorithm

TFA [22] was proposed as an extension of the Transactional Locking 2 (TL2) algorithm [7] 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 lazy concurrency control, buffering all operations in per-transaction read and write sets, and acquiring the object-level locks at commit time. Objects are updated once all locks have been successfully acquired. Being a data-flow algorithm, in TFA objects migrate to the node that successfully commits a transaction which updates the respective objects. Failure to acquire a lock aborts the transaction, releasing 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 always observes a consistent view of the memory (i.e., opacity [9]). This is important in STM because operations are not sandboxed, and thus observing an inconsistent snapshot may lead to unrecoverable errors (e.g., division by zero). Opacity 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 $LC_2 = 24$). 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 (but it will retry later).

2.3 SCORe

SCORe [21] is a control-flow based, scalable, one-copy serializable partial replication protocol. It is genuine, as only nodes replicating data touched by a transaction are contacted during the execution and commitment of the transaction. It also allows read-only transactions to commit locally (without any remote communication during the commit phase) by ensuring transactions always read from a consistent snapshot.

SCORe combines a local multi-version concurrency control algorithm with a distributed logical clock synchronization scheme. Each replica holds multiple versions of the objects it maintains, which are tagged with a scalar timestamp. The clock synchronization scheme is used to (a) determine the snapshot visible to transactions, and (b) agree on a final global serialization order for read-write transactions.

All nodes maintain two scalar variables: *commitId* stores the timestamp of the last read-write transaction to commit on that node, and *nextId* holds the timestamp of the node will propose at the next commit request. Each transaction is associated with a snapshot identifier (*sid*). The *sid* is recorded at the first read operation within each transaction. It is the greatest of the *commitId* at the current node, and the *commitId* at the node servicing the read (if different). The first read operation in a transaction returns the latest version of the object being read. All further reads may only observe object versions whose tag number is $\leq sid$, in order to maintain a consistent snapshot.

SCORe commits transactions using an algorithm that can be seen as a combination between a Two-Phase Commitment (2PC) and the Skeen total order multicast algorithm [10]. 2PC is used to validate the optimistic execution of update transactions and to ensure the global state is updated atomically. Skeen's algorithm is responsible for agreeing on a final commit ordering across all nodes replicating a certain object. Given that SCORe is a control-flow algorithm, objects are immobile and do not migrate.

Finally, a node's *nextId* is advanced whenever a transaction with a larger *sid* reads from that node.

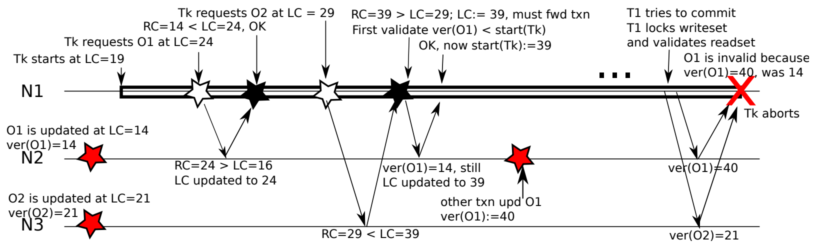


Fig. 4. Transactional Forwarding Algorithm Example (from [24])

This effectively tracks data dependencies between transactions and ensures that a transaction updating object X is serialized after all transactions that have observed a previous version of X.

3 System model

3.1 Base model

As in [13], 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 a 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 a set of owner nodes, denoted by $owners(O_j)$.

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 Hosking [19]. 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.

In the original versions of both TFA and SCORe, 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: inside a transaction, two reads of the same object (not separated by a write) 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 nesting, 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 open nested sub-transactions.

Transactional operations are similar for open nested sub-transactions as they are for a root transaction without any nesting. 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, the object is fetched from the globally committed memory. 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. 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 5.2.

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 semantic 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 [2] and must be used with care. In practice, it is sufficient in most cases to ensure serializability at each abstraction level with respect to other operations at the same level, while preserving conflicts at higher levels (i.e., level-by-level serializability [25]). Level-by-level serializability 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 [11]. 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 *semantic*, and it is essential for a correct execution. One such mechanism for detecting semantic conflicts is by using *abstract locks* (locks that protect an abstract state as opposed to a concrete memory location). 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 [18], 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 them. We thus decide to disallow this behavior in our implementations: open sub-transactions may not update memory which was also accessed by any of their ancestors. We thus impose a clear separation between the memory locations accessed by transactions at the multiple abstraction levels. This separation should make the usage of open nesting less confusing for programmers. Failure to comply to this rule can easily be caught by the runtime system and the programmer notified.

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 [20] – regular programmers can then use those libraries to take advantage of open nesting benefits.

4 Open nesting algorithms: TFA-ON, SCORe-ON

To add open-nesting to a DTM algorithm, one needs to allow for sub-transactions that behave similarly to root transactions, i.e., sub-transactions that commit their changes directly to the globally committed memory. However, such sub-transactions may interact in non-trivial ways with their parents. This section describes the TFA-ON and SCORe-ON algorithms by clarifying these interactions.

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

We describe TFA-ON with respect to the TFA algorithm and N-TFA [24], 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 children of a given 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 globally committed 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. The procedure for accessing objects

```
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;
  }
 }
```

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

is similar to the original TFA. Essentially, in TFA-ON a sub-transaction is treated similarly to a root transaction because it has to commit globally (which is not the case of N-TFA). For this reason, the TFA-ON's logic associated to the commit of a sub-transaction is more complex than the one of N-TFA.

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 SCORe with Open Nesting (SCORe-ON)

For the most part, SCORe-ON transactions (both parents and open-nesting children) behave similarly to normal SCORe transactions, as described in Section 2.3. However, due to snapshot reads (MVCC) and the fact that SCORe commits read-only transactions differently from read-write transactions, special treatment is needed for the various parent/child combinations. We discus how SCORe-ON handles these combinations bellow:

Read-write parent, read-write child. This is the normal behavior where both parent and child undergo the distributed commitment protocol. The child acquires any needed abstract locks, which get passed to the parent upon the sub-transaction's commit.

Read-only parent, read-write child. In this situation, the parent must be treated as a read-write transaction and undergo the distributed commitment protocol. More specifically, the read-set must be validated at commit time. Failure to do so may allow a sub-transaction to make changes based on stale data, thus breaking serializability.

Read-write parent, read-only child. To ensure correctness in this case, SCORe-ON must acquire abstract locks for all read-only sub-transactions. This guarantees a higher-level read operation can not become stale, potentially leading the parent transaction execute an incorrect write operation. A simple way to implement lock acquisition is as normal DTM read-write operations, effectively transforming the child into a read-write sub-transaction that must undergo commit-time validation. Thus, the snapshot reads optimization can not be applied to any sub-transaction that requires abstract locks. This again is needed for maintaining correctness.

Read-only parent, read-only child. This case is essentially a whole read-only transaction. In SCORe, read-only transactions are executed using snapshot reads and never need to abort. Applying open-nesting semantics to this case would negate this optimization. To avoid this, the programmer should instead use normal flat nesting. If this case is not spotted at design time, the system would unnecessarily acquire abstract locks for read-only sub-transactions, slowing transaction execution and reducing concurrency.

5 Mechanisms and Implementation

Beyond the necessary protocol modifications as described in TFA-ON and SCORe-ON, several additional mechanisms are needed in order to support open nesting in an actual implementation. These mechanisms relate to dealing with abstract lock management and the execution of commit and compensating actions.

5.1 Abstract locks

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).

5.2 Defining transactions and compensating actions

Commit and compensating actions are registered when an open-nested sub-transaction commits. They are to be executed as open-nested transactions by the innermost open-nested ancestor, when it commits, or respectively, aborts.

We chose to use anonymous inner classes for defining transactions and their optional commit and compensating actions. Compared to automatic or manual instrumentation, this 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 network latency.

We aimed to make the mechanism for defining open nested transactions consistent across implementations. Specifically, the Atomic<T> acts as a compatibility layer above both Infinispan and Hyflow, and abstracts away the API differences between frameworks — Infinispan uses a map-like interface for accessing data (i.e, get and set), while Hyflow has a directory for keeping track of objects (i.e., open and register). Hyflow's directory implementation was reused in TFA/Infinispan, in order to support object migrations, as required by TFA and the data-flow model. Furthermore, our Atomic<T> layer relieves the user from having to know the actual model currently in use (data-flow or control-flow).

Figure 6 shows how a transaction would look in our implementations. 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 this copy may be out-of-date by the time the handler executes (automatic re-open may be a way to address this issue).

```
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();
```

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

5.3 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 and Infinispan each thread had its own context object, in order to support nesting, we arrange the context objects in thread-local stacks. Each sub-transaction 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.

6 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.

6.1 Experimental settings

The performance of TFA-ON and SCORe-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, and two commercial inspired benchmarks, such as TPC-C [5] and ReTwis [14]. Each protocol was implemented in both

Hyflow and Infinispan, for a total of four implementations.

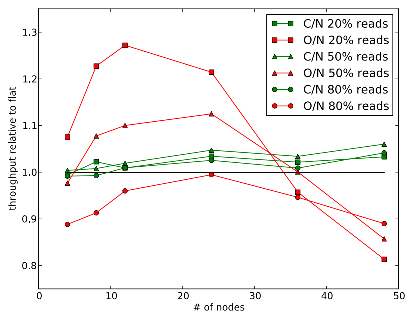
Our evaluation is focused mostly on TFA-ON/Hyflow. Given that Hyflow is our DTM framework research prototype, we were able to easily collect a variety of metrics that allowed us to perform a comprehensive analysis of open nesting behavior. The remaining three implementations (TFA-ON/Infinispan, SCORe-ON/Infinispan and SCORe-ON/Hyflow) were evaluated at a higher level, to confirm that our findings are still valid across different base algorithms and different software frameworks. 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].

For TFA-ON/Hyflow, we ran the micro-benchmarks under flat [22], closed [24], 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).

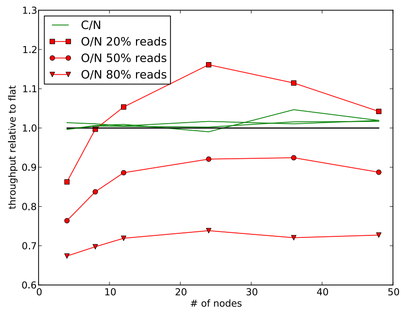
Other data that we recorded includes:
- Number of objects committed per (sub-)transaction.
- Which sub-transaction caused the parent transaction to abort.

The skip-list, hash-table, and BST benchmarks instantiate three objects each, 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 $r \in \{20, 50, 80\}$.
- 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 semantic conflicts. Unless otherwise stated, we used $k = 100$.

The fourth micro-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.



(a) Skip-list



(b) Hash-table

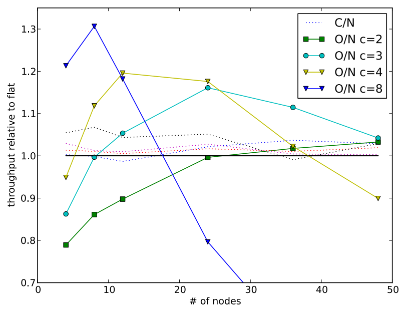
Fig. 7. Performance relative to flat transactions, with $c = 3$ calls per transaction and varying read-only ratio. Both closed nesting and open nesting are included. (TFA-ON/Hyflow)

TPC-C is a popular benchmark modeled after a commercial order-entry environment, and is representative of modern online transactional processing (OLTP) workloads. ReTwis is a clone of the popular website Twitter, and approximates modern social-networking inspired workloads. Both applications were configured with medium/high contention workloads.

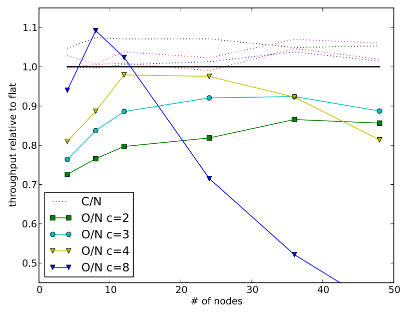
As testbed we used up to 48 nodes. Each node is an AMD Opteron processor clocked at 1.9GHz. We used the Ubuntu Linux 10.04 server OS and a network with $1ms$ end-to-end link delay.

6.2 Experimental results

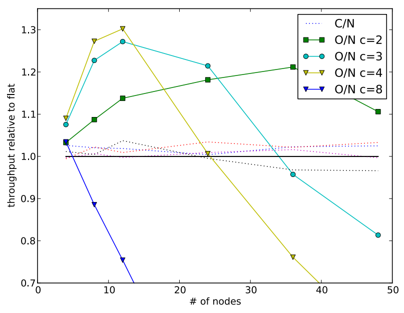
We start with our TFA-ON/Hyflow experimental study before we move on to the other implementations. For all the data-structure micro-benchmarks, we observed that open nesting's best performance improvements occur at low read-only ratio workloads. For brevity, we only focus on skip-list and hash-table in this paper. Figure 7 shows how open



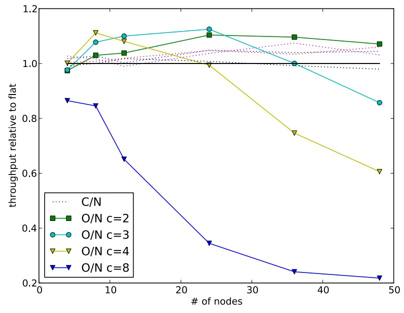
(a) Hash-table 20% reads



(b) Hash-table 50% reads



(c) Skip-list 20% reads



(d) Skip-list 50% reads

Fig. 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. (TFA-ON/Hyflow)

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).

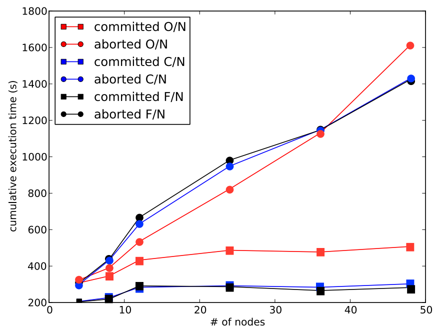
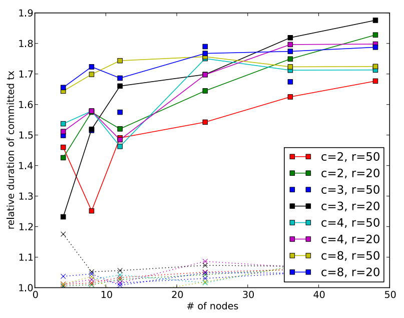


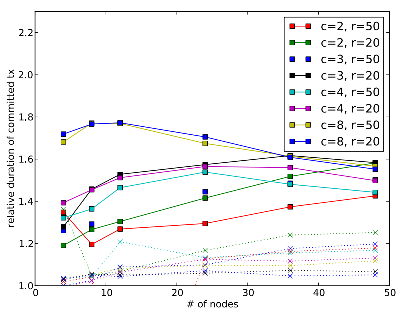
Fig. 9. Time spent in committed vs. aborted transactions, on hash-table with $r = 20$ and $c = 4$. Lower lines (square markers) represent time spent in committed transactions, while the upper lines (circle markers) represent the total execution time. The difference between these lines is time spent in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

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 $c = 8$. On skip-list it is noticeable that at high contention ($c = 8$) the region of maximum benefit disappears 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

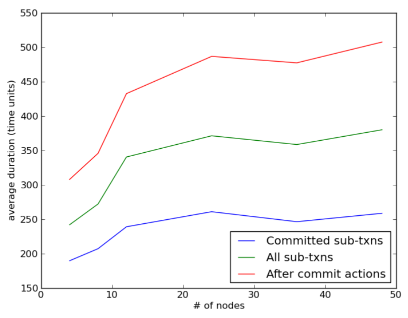


(a) Hash-table

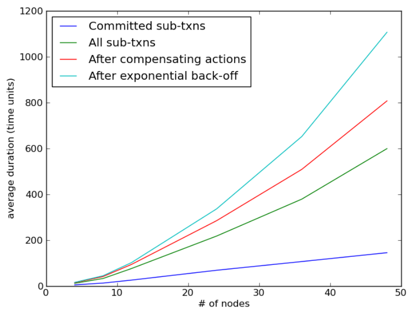


(b) Skip-list

Fig. 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. (TFA-ON/Hyflow)



(a) Committed transactions



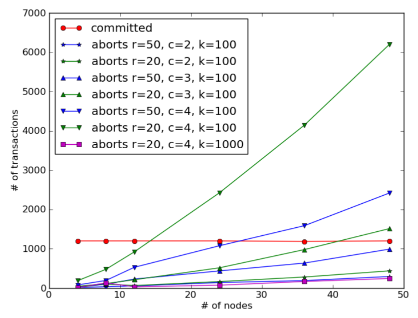
(b) Aborted transactions due to abstract lock acquisition failure

Fig. 11. Breakdown of the duration of various components of a transaction under open nesting, on hash-table with $r = 20$ and $c = 4$. (TFA-ON/Hyflow)

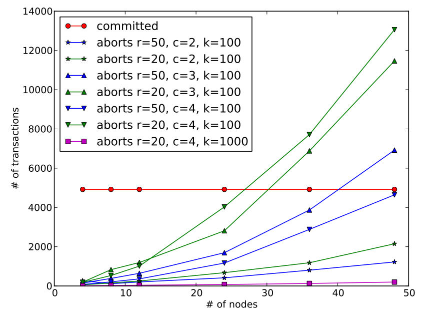
the average overheads of open nesting relative to flat transactions (50-80% on hash-table and 40-50% on skip-list) are significant and higher than that of closed nesting (3-7% on hash-table and 5-16% on skip-list). We observe the overheads are benchmark dependent, and are lower for workloads which access 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 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 transaction (Figures 11(a) and 11(b)), we see that the duration of transactions (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 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. With respect to the top-level transactions, abstract locks are acquired eagerly – when the sub-transaction which performed the access commits. When semantic conflicts are frequent, this strategy will cause more aborts and lower performance compared to TFA's strategy, which defers all lock acquisitions to the end of each top-level transaction.

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 semantic 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

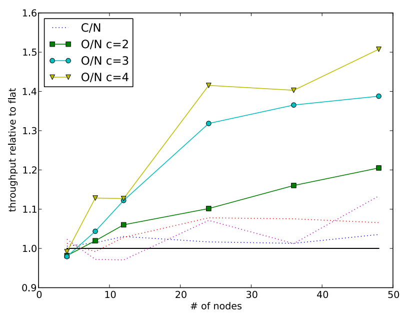


(a) Hash-table

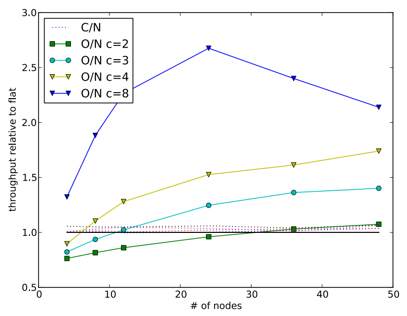


(b) Skip-list

Fig. 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. (TFA-ON/Hyflow)



(a) Skip-list



(b) Hash-table for r=20

Fig. 13. Throughput relative to flat nesting with increased key space $k = 1000$ and write-dominated workloads $r = 20$. (TFA-ON/Hyflow)

21% improvement at $r = 50$.

In our enhanced counter micro-benchmark we observed improvements consistent with our previous findings. 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, as open nesting releases isolation early, optimistically assuming the parent will commit. Increased contention after the open-nested sub-transaction contradicts 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).

The evaluation of our other three implementations are presented in Figures 14-17. The absolute numbers differ due to differences in the underlying architecture and benchmark configurations, but the general trends are consistent to those in our comprehensive evaluation of TFA-ON/Hyflow. It is worth to mention that some scalability bottleneck raises at high node count. This is mainly because, given the uniform distribution of shared objects in the system, the commit phase of a sub-transaction involves more nodes when the system's size is large. As a results, more nodes have to be contacted and thus transaction latency increases.

In TFA-ON/Infinispan (Figure 14) the relative throughput sees an initial increase, followed by a drop. The peak throughput is however wider and the slopes in the graph are much gentler. This test was configured with $c = 3$ and $r = 0$. To further investigate such a performance drop, we measured the average latency (including the aborted trials) and the abort ratio of transactions in Figure 15. Hash-table and BST do not show a significant amount of aborts, which makes open-nesting less effective than skip-list. Given that, at 24 and 48 nodes the cost of committing open-nested transactions overweights the possible benefits, which explains the drop in performance.

For both SCORe-ON implementations (Figures 16, and 17) open nesting performs significantly worse at low contention (fewer nodes), which can be attributed by the inherent differences between TFA and SCORe algorithms — since SCORe orders commit operations using a commit queue, the overhead of extra commits in the case of open nesting is greater. SCORe was configured with the same settings as TFA, to make

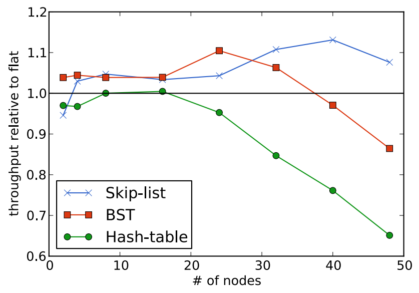


Fig. 14. Relative throughput for TFA-ON implementation in Infinispan.

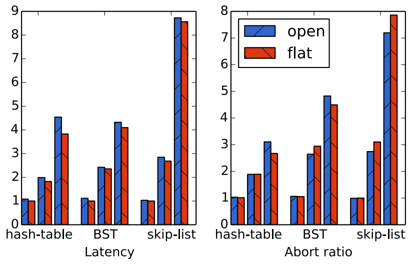


Fig. 15. Latency and abort rate for TFA-ON/Infinispan using 8, 24, 48 nodes. Results are normalized to the lowest measurement per cluster.

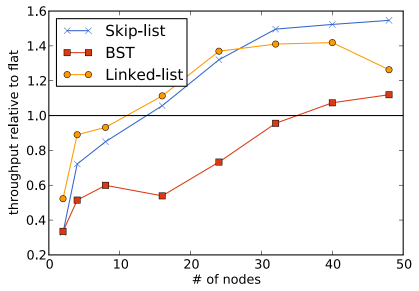


Fig. 16. Relative throughput for SCORe-ON implementation in Infinispan using micro-benchmarks.

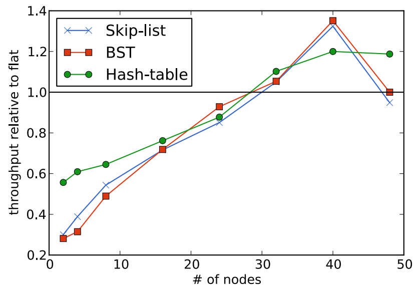


Fig. 17. Relative throughput for SCORe-ON implementation in Hyflow using micro-benchmarks.

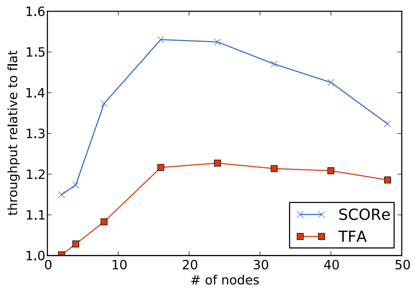


Fig. 18. Relative throughput of SCORe-ON and TFA in Hyflow using TPC-C benchmark.

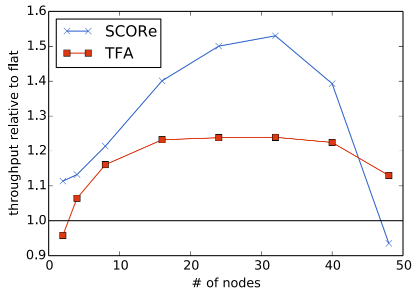


Fig. 19. Relative throughput of SCORe-ON and TFA in Hyflow using ReTwis benchmark.

the comparison fair. Objects were distributed across nodes using a consistent hash function (as is standard in Infinispan). The benchmark parameters were $c = 7$ and $r = 0$. We purposefully avoided read-only transactions in these tests, as SCORe-ON would not need to employ open nesting due to the consistent snapshot reads (See Section 4.2).

To assess the performance of TFA-ON and SCORe-ON when commercial inspired benchmarks are deployed, we implemented the distributed version of two well-known applications, namely TPC-C [5] (Figure 18) and ReTwis [14] (Figure 19). Both the applications have been integrated into the Hyflow framework

The performance of both the open nesting approaches using TPC-C is similar in terms of scalability trend, but with different gains with respect to the original (flat nesting) version of the code. The reason of the showed speed up is mainly related to the possibility of releasing a warehouse (i.e., stop monitoring the modifications on the warehouse object by removing it from the transaction read-set) immediately after its update and without keeping it until the end of the transaction itself. Increasing the node count increases also the contention, which becomes very high given the characteristics of TPC-C. As a result, the parallelism enabled by open nesting diminishes along with the overall performance.

Figure 19 shows the results using ReTwis configured with 500 users and 50 as maximum followers. ReTwis behaves as a distributed hash-table where operations include `post`, which appends a new message to all of a user's followers' timeline, and `get`, which retrieves said timeline for display. The overall performance is determined by the `post` operation because it modifies a significant number of objects. The benefits stem from releasing isolation earlier, thus reducing the probability of a conflict with another concurrent post operation. SCORe-ON and TFA show similar scalability, with a growing trend until the system reaches its best throughput, and then a degradation starting from 40 nodes. This drop in performance is more accentuated on SCORe-ON, which observes a rapid increase in the number of aborts at and above 40 nodes, due to the inherent characteristics of the underlying commit protocol and the nature of the workload (long transactions affecting many objects).

7 Conclusions

We presented TFA-ON and SCORe-ON, extensions to two DTM algorithms with support for open nesting. We determined that open nesting performance is limited by two factors: commit overheads and semantic conflict rate. Semantic 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. We also confirmed that open nesting does not apply well to workloads which incur contention after the open-nested sub-transaction commits.

Acknowledgment

This work is supported in part by US National Science Foundation under grants CNS 1116190, and CNS 1217385, and by US Air Force Office of Scientific Research under grant FA9550-14-1-0187.

References

[1] Kunal Agrawal, I.-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In *PPOPP '09*.

[2] Philip A. Bernstein, Vassco Hadzilacos, and Nathan Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

[3] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *IPDPS '10*.

[4] Nuno Carvalho, Paolo Romano, and Luís Rodrigues. A generic framework for replicated software transactional memories. In *NCA '11*.

[5] TPC Council. TPC-C benchmark. 2010.

[6] Aditya Dhoke, Roberto Palmieri, and Binoy Ravindran. On reducing false conflicts in distributed transactional data structures. In *ICDCN*, pages 8:1–8:10, 2015.

[7] David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In *DISC '06*.

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

[9] Rachid Guerraoui and Michal Kapalka. On the correctness of transactional memory. In *PPoPP '08*.

[10] Rachid Guerraoui and André Schiper. Genuine atomic multicast in asynchronous distributed systems. *Theor. Comput. Sci.*, 254(1-2):297–316, March 2001.

[11] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *PPOPP '08*.

[12] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA '93*.

[13] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *DISC '05*.

[14] Costin Leau. Spring data redis - retwis-j, 2013. http://docs.spring.io/spring-data/data-keyvalue/examples/retwisj/current/.

[15] F. Marchioni and M. Surtani. Infinispan data grid platform. *PACKT Publishing*, 2012.

[16] Michelle J. Moravan, Jayaram Bobba, Kevin E. Moore, Luke Yen, Mark D. Hill, Ben Liblit, Michael M. Swift, and David A. Wood. Supporting nested transactional memory in logtm. In *ASPLOS '06*.

[17] J. Eliot B. Moss. *Nested Transactions: An Approach to Reliable Distributed Computing*. PhD thesis, MIT, 1981.

[18] J. Eliot B. Moss. Open nested transactions: Semantics and support (poster). In *Workshop on Mem Perf Issues*, 2006.

[19] J. Eliot B. Moss and Antony L. Hosking. Nested tm: Model and architecture sketches. *Sci Comp Prog*, 63(2):186–201, 2006.

[20] Yang Ni, Vijay Menon, Ali-Reza Adl-Tabatabai, Antony L. Hosking, Richard L. Hudson, J. Eliot B. Moss, Bratin Saha, and Tatiana Shpeisman. Open nesting in software transactional memory. In *PPOPP '07*.

[21] Sebastiano Peluso, Paolo Romano, and Francesco Quaglia. Score: A scalable one-copy serializable partial replication protocol. In *Middleware '12*.

[22] Mohamed M. Saad and Binoy Ravindran. Hyflow: a high performance distributed software transactional memory framework. In *HPDC '11*.

[23] Alexandru Turcu and Binoy Ravindran. On open nesting in distributed transactional memory. In *SYSTOR '12*.

[24] Alexandru Turcu, Binoy Ravindran, and Mohamed M. Saad. On closed nesting in distributed transactional memory. In *TRANSACT '12*.

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

Alexandru Turcu Alexandru Turcu received the MEng in Digital Electronics in 2010, from The University of Sheffield, UK, and his PhD in Computer Engineering in 2015 from Virginia Tech, Blacksburg, Virginia. His research interests include Distributed Systems, Transactional Memory and Transactional Systems. He is currently a software engineer at Google.

Roberto Palmieri received the BSc in computer engineering, MSc and PhD degree in computer science at Sapienza, University of Rome, Italy. He is a Research Assistant Professor in the ECE Department at Virginia Tech. His research interests include exploring concurrency control protocols for multicore systems, cluster and geographically distributed systems, with high programmability, scalability, and dependability.

Binoy Ravindran is a Professor of Electrical and Computer Engineering at Virginia Tech, where he leads the Systems Software Research Group, which conducts research on operating systems, run-times, middleware, compilers, distributed systems, fault-tolerance, concurrency, and real-time systems. Ravindran and his students have published more than 220 papers in these spaces, and some of his group's results have been transitioned to the DOD. Dr. Ravindran is an Office of Naval Research Faculty Fellow and an ACM Distinguished Scientist.