Technical Report

On Closed Nesting in Distributed Transactional Memory

Alexandru Turcu
Virginia Tech
talex@vt.edu

Binoy Ravindran
Virginia Tech
binoy@vt.edu

Mohamed Saad
Virginia Tech
msaad@vt.edu

Abstract

Distributed Software Transactional Memory (D-STM) is a recent but promising model for programming distributed systems. It aims to present programmers with a simple to use abstraction (transactions), while maintaining performance and scalability similar to distributed fine-grained locks. Any complications usually associated with such locks (i.e. distributed deadlock) are avoided. Building upon the previously proposed Transactional Forwarding Algorithm (TFA), we add support for closed nested transactions. We further discuss the performance implications of such nesting, and identify the cases where using 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 D-STM system with support for closed nested transactions and partial aborts.

Keywords distributed systems, software transactional memory, closed nesting

1. Introduction

Transactional Memory (TM) was introduced in 1993 by Herlihy et al. [1] as an alternative to traditional concurrency control based on locks. It attempts to provide performance and scalability that is at least comparable to fine-grain lock systems [2], while having an easy to use programming interface. Transactional Memory exists in three flavors: Hardware Transactional Memory (HTM) employs hardware support and is usually implemented as an extension to existing cache coherence protocols [1]; Software Transactional Memory (STM) turns regular memory accesses into transactional operations but is not limited by hardware restrictions [3]; Hybrid Transactional Memory is a combinations between the two, taking what's best from both HTM and STM [4].

In TM systems, the code that needs to access shared objects is organized as a transaction. When a transaction writes to an object, any other concurrent transaction that accesses the same object is said to *conflict* with the first one. In such a situation, only one may immediately proceed. A *conflict manager* decides whether to delay or *abort* one of the transactions, and which one. When a transaction executes to completion without any conflicts it is said to *commit*, and its execution appears to all other transactions as atomic. A transaction that aborts is *rolled back* such that its effects are not visible and is later restarted, after the conflict has been resolved (see Figure 1 for a Java sketch of how STM transactions work).

```
for (int i=0; i<MAX_RETRIES; i++)
{
  // initialize transaction explicitly
  BeginTransaction();
  // specialized function for reads
  int val = TransactionRead(ADDR_A);
  // specialized function for writes
  TransactionWrite(ADDR_A, val+1);
  // attempt to commit
  if (CommitTransaction())
    // transaction committed
    return true;
  else
    // Need to undo what we wrote
    RollbackTransaction();
}
// transaction failed to commit within
// the given number of retries
return false;
```

Figure 1. Simplified behind-the-scenes Java code for a transaction that increments a variable. Note that variable A's value cannot be directly read nor written.

Concurrency control in distributed systems is traditionally achieved using distributed locks and 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, without access to the libraries' source code.

To address these problems, Distributed Software Transactional Memory (D-STM) was proposed as an alternative concurrency control mechanism [5]. D-STM systems can be classified by the mobility of the transactions or data. In the more popular data-flow model [5–7], objects are migrated between nodes to be operated upon by the immobile transactions. Alternatively, in the control-flow model [8], the objects are immobile and are accessed by transactions using Remote Procedure Calls (RPC).

In TM, nesting is used to make **code composability** easy. A transaction is called *nested* when it is enclosed within another transaction. There are three types of nesting [8, 9]: flat, closed and open. They differ based on whether the parent and children transactions can independently abort:

```
@Atomic void
transfer(String acc1, String acc2, int x){
  withdraw(acc1, x);
  deposit(acc2, x);
}
@Atomic void
withdraw(String acc_id, int amount) {
  Account acc = HyFlow.getLocator()
    .open(acc_id);
  account.amount -= amount;
}
@Atomic void
deposit(String acc_id, int amount) {
  Account acc = HyFlow.getLocator()
    .open(acc_id);
  account.amount += amount;
}
```

Figure 2. Simple nested transactions in Java. This example assumes that methods annotated with @Atomic are automatically instrumented to the transactional form.

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 and increasing performance.

Open nesting
: In open nesting, operations are considered at a higher 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 enables increased concurrency and performance.

A simple Java code example with nested transactions in a distributed setting is shown in Figure 2. Besides providing support for code composability, nested transactions are attractive when transaction aborts are actively used for implementing specific behaviors. **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). 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...erElse). In both these scenarios, performance can be improved with nesting by aborting and retrying only the inner-most sub-transaction.

Previous D-STM work has ignored the subject of partial aborts and nesting. We extend the existing Java D-STM framework named HyFlow[1] and the previously proposed TFA algorithm to support closed nesting. The resulting algorithm is named Nested Transactional Forwarding Algorithm (N-TFA), and is shown to be opaque and strongly progressive. To the best of our knowledge, this work contributes the first ever D-STM implementation with support for closed nesting. We test our implementation through a series of benchmarks and observe throughput improvements of up to 84% in specific cases. However, the average performance was only 2% higher compared to flat transactions. Thus, we identify the kinds of workloads that are a good match for closed nesting and how various parameters influence the gain (or loss) in throughput.

The remainder of the paper is organized as follows: Section 2 presents related work on nested transactions and the original TFA algorithm. In Section 3, we describe our system model. N-TFA is presented as an extension of TFA in Section 4. We analyze N-TFA and sketch correctness and liveness proofs in Section 5. Implementation details and benchmark results are discussed in Section 6. Finally, we suggest future research directions and conclude the paper in Sections 7 and 8, respectively.

2. Related work

2.1 Nested Transactions

Nested transactions originated in the database community and were thoroughly described by Moss in [10]. 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.

One of the early works introducing nesting to Transactional Memory was done by Moss and Hosking in [9]. They describe the semantics of transactional operations in terms of *system states*, which are tuples grouping 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 [13], explaining how using multiple levels of abstractions can help in differentiating between fundamental and false conflicts and therefore improve performance.

Moravan et al. [12] implement closed and open nesting upon their earlier LogTM HTM proposal. They implement it by maintaining a stack of log frames, similar to the 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. Their experiments show that closed nesting can, in certain benchmarks, perform up to 10% better than flat nesting, but in many cases, it brings no benefit. Open nesting was only possible to apply 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 [14]. 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.

As an alternative mechanism for supporting partial roll-backs, Koskinen and Herlihy argue for using a checkpointing mechanism instead of transaction nesting [15]. Checkpoints are a generaliza-

[1] HyFlow is available on-line, as an open-source project, at http://hyflow.org

tion of closed nesting: transactions can be rolled back to any previous checkpoint in order to resolve conflicts. Furthermore, setting up checkpoints and rollback can be more intelligently controlled compared to the rigidness of closed nesting, where rollback may only be performed to an ancestor (enclosing) function call boundary. The drawback for checkpoints is the need to save/restore the processor context and the activation stack. While C and C++ support this via the *setcontext*-family functions, Java by default lacks support for this functionality.

2.2 Transactional Forwarding Algorithm

TFA[16, 17] was proposed as an extension of the Transactional Locking 2 (TL2) algorithm [2] for D-STM. 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 "happens before" relationships between significant events. TFA uses optimistic concurrency control, acquiring the object-level locks lazily at commit time.

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.

For completeness, we illustrate TFA with an example. In Figure 3, 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 (it will retry later — not shown in the figure).

3. System model

3.1 Base model

Since we extend upon Saad [16, 17], our work uses the same system model (which is, in turn, based on Herlihy and Sun [5]). Specifically, we consider a set of n nodes: $\{N_1, N_2, ...N_n\}$. The nodes may communicate via message passing links. Messages may incur communication delays. We can thus represent the network using an undirected graph $G = (N, E, c)$, where N is the set of nodes, E is the set of links and c is the function defining the communication cost. The messages transmitted through the network are denoted by the set M.

Let $O = \{O_1, O_2, ...\}$ be the set of objects accessed using transactions. Every such 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. Transactions have three possible states: active, committed and aborted. Each transaction has an unique identifier. Any aborted transaction is later retried using a new identifier. The read-set and write-set of a transaction T_k are denoted with $readset(T_k)$ and $writeset(T_k)$, respectively.

3.2 Nesting Model

In an approach identical to Moss and Hoskin [9], we define the nesting semantics with respect to the system state. The system state S is the totality of system state entries s_i. Each system state entry $s_i = (T_k, id_j, w, val_{O_j})$ is a tuple aggregating the following information:

- A transaction identifier T_k.
- An object identifier id_j.
- A boolean value *w* denoting whether the current system state entry represents a read or a write.
- A value val_{O_j} representing the contents of the object read or written to memory.

For simplicity, we assume that the granularity of read/write operations is at object level. Finer grained (field-level) access is however possible, and is in fact used throughout our implementation.

Let $parent(T_k)$ denote the parent (enclosing transaction) of a transaction T_k. If T_k is a top-level transaction, then $parent(T_k) = \emptyset$. Each transaction may only have one active child, i.e. parallel nested transactions are outside the scope of this work.

System state entries with $T_k = \emptyset$ represent the globally committed memory. In our distributed system model, each such entry refers to the most recently committed version of an object, O_j. By definition, the system state must contain an entry for each location of the globally committed memory, i.e. $\forall O_j \in O, \exists s_i \in S \mid s_i = (T_k = \emptyset, id_j, w = 1, val_{O_j})$.

A read operation in the context of a transaction T_k is performed using the following procedure:

- If transaction T_k has a system state entry for object O_j, return the value val_{O_j} of that entry. This can be either part of the read-set or of the write-set.
- Otherwise, recursively attempt the read operation in the context of $parent(T_k)$.

This is guaranteed to work, as the system state must specify a value in the globally committed memory for all possible locations. An entry $s_i = (T_k, id_j, w = 0, val_{O_j}^{read})$ is added to the system state upon reading of object O_j by transaction T_k, if it does not already exist. Write operations simply add the appropriate entry to the system state, $s_i = (T_k, id_j, w = 1, val_{O_j}^{written})$.

A transaction T_k is allowed to write an object O_j if for all system state entries $(T_{k_2}, id_j, w, val_{O_j})$ corresponding to object O_j, the transaction T_{k_2} is either T_k or an ancestor of T_k. Reads are

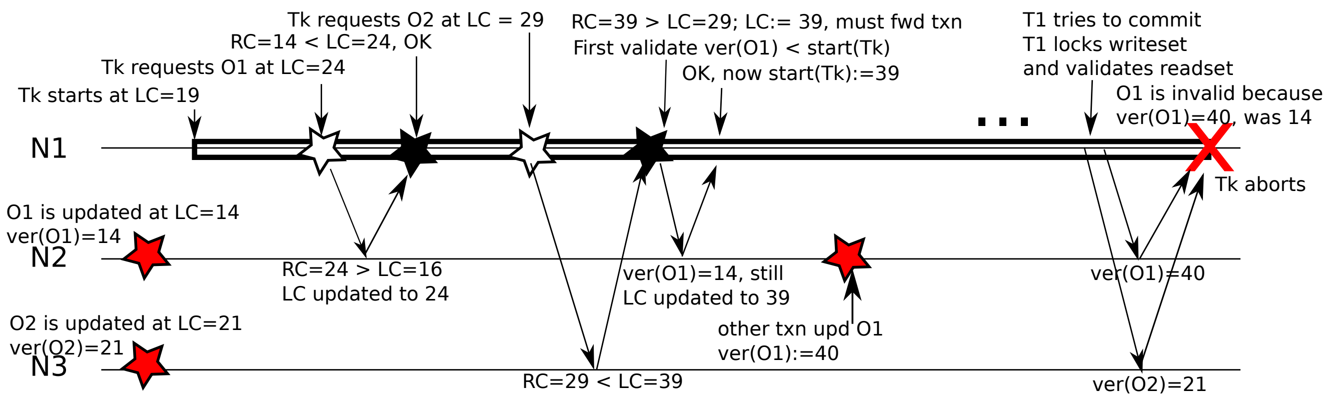

Figure 3. Transactional Forwarding Algorithm Example

also allowed if w=false for all entries for which T_{k_2} is not T_k or an ancestor of T_k. If a read or write operation is not allowed, but a transaction attempts it anyway, a conflict arises. In our implementation, the Transaction Manager will resolve the conflict by aborting all but one of the conflicting transactions.

Upon the successful commit of transaction T_k, the following actions are performed on the system state:

- Drop all entries for transaction T_k, and also drop the entries for $parent(T_k)$ that access memory locations which are part of T_k's read and write sets.
- Add new entries for $parent(T_k)$ corresponding to the old entries of T_k. Care must be taken when setting the write flag: if the location of the new entry was in the write-set of either T_k or $parent(T_k)$, the flag is set to true.

Otherwise, if the transaction is aborted, all entries corresponding to transaction T_k are dropped from the system state.

Read and write operations do not need to be validated at the time they occur, as the TFA algorithm uses optimistic concurrency control. Thus, reads and writes are recorded into a per-transaction replay-log structure that appears to mimic the set of system state entries corresponding to the transaction in question.

4. Nested Transactional Forwarding Algorithm

In TFA, transactions are immobile. Furthermore, we also consider that all sub-transactions of a transaction T_k are created and executed on the same node as T_k. Within these assumptions, it is straightforward to implement the rules described in the previous subsection.

Note that there are two types of commit. The original, *top-level commit model* is used when a top-level transaction commits the changes from its replay-log to the globally committed memory. This commit is only performed after the successful validation of all objects in the transaction's read-set, as defined by the TFA algorithm [17]. If the validation fails, i.e. at least one of the objects' version is newer than the current transaction's starting time, the transaction is aborted. The new *merge commit model* is used when a sub-transaction commits the changes from its replay-log to the replay-log of its parent.

A number of questions about how to apply TFA in the context of nested transactions arise. In TFA, every transaction commit increments the node-local clock and updates the affected objects' lock version. Should these operations also be performed upon the commit of a sub-transaction? Which objects should be processed during the early-validation procedure? What is the meaning of transaction forwarding inside a sub-transaction?

By answering these questions, we design a protocol which we will call Nested Transactional Forwarding Algorithm (N-TFA). Additionally, we must note that two variations of N-TFA can be obtained based on whether merge commits are conditioned by a read-set validation or occur unconditionally. We will call them N-TFA with validation (N-TFA w/V) and N-TFA without validation (N-TFA w/o V), respectively.

Assume that transaction T_k opened and read an object O_1. Let T_{k2} be a sub-transaction of T_k. Assume that T_{k2} also reads object O_1, and moreover, T_{k2} can successfully commit (O_1 was not modified by any other transaction). Intuitively, T_{k2} should not update the object's lock version when it commits, because, the object as seen by other transactions did not change. If the version was updated at this point, other unrelated transactions would be forced to unnecessarily abort due to invalid read-set even if T_k eventually aborts (due to other objects) without changing O_1 in the globally committed memory.

In order to maintain similarity with the original TFA, all objects will be validated against the outer-most transaction's starting time. While we could imagine an algorithm where sub-transaction's start times were used to validate objects, doing so would only add unnecessary complexity and would provide no real benefit. Therefore, all transaction forwarding operations must be operated upon the starting time of the root transaction.

Summarizing the previous two observations, the starting time of sub-transactions is not used for object validity verification and the object versions are not updated upon a sub-transaction's commit. Consequently, merge-commits and the start of new sub-transactions are not globally important events and should not be recorded by incrementing node-local clocks. If the clocks were incremented on such events, remote nodes would need to perform the transaction forwarding operation unnecessarily, only to find that no objects were changed. This is undesirable as the forwarding operation bears the overhead of validating all objects in the transaction's read-set. Additionally, since no global objects are changed at merge-commits, no locks need to be acquired for such commits.

Early validation is the process that checks for the consistency of all objects in a transaction's read-set before advancing the transaction's starting time. If early validation was performed on only the objects in the current sub-transaction (say, T_{k2}), a situation may arise when an object in a previous sub-transaction (say, T_{k1}) becomes inconsistent. In such a case, the parent transaction's clock would be advanced, thereby erasing any evidence that T_{k1}'s object is inconsistent. Thus, early validation must process all objects encountered to date by the outer-most enclosing transaction and all of its children.

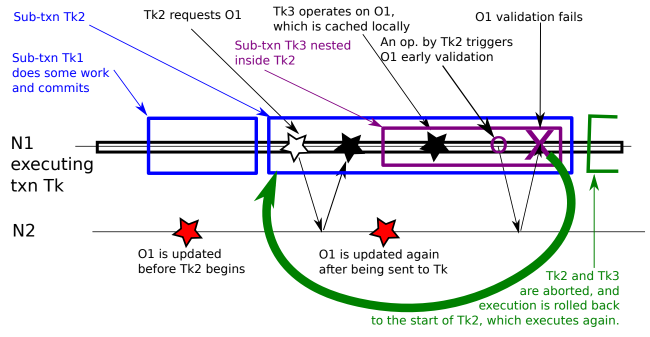
Sub-txn Tk2
Tk2 requests O1
Tk3 operates on O1, which is cached locally
O1 validation fails
Sub-txn Tk3 nested inside Tk2
An op. by Tk2 triggers O1 early validation
Sub-txn Tk1 does some work and commits
N1 executing txn Tk
N2
O1 is updated before Tk2 begins
O1 is updated again after being sent to Tk
Tk2 and Tk3 are aborted, and execution is rolled back to the start of Tk2, which executes again.

Figure 4. Nested Transactional Forwarding Algorithm Example

In case one or more objects are detected as invalid, the uppermost transaction that contains an invalid object and all of its children should be aborted. In TFA, it was sufficient to stop the validation procedure when the first invalid object is observed. However, with N-TFA, all objects within the root transaction must be validated (ideally in parallel) in order to determine the best point to roll back to.

Let's now look at an example of N-TFA (Figure 4). The top-level transaction T_k is executing on node N_1. A sub-transaction T_{k1} executes and commits successfully. Next, another sub-transaction T_{k2} opens an object O_1, which is located on node N_2. T_{k2} spawns a further sub-transaction, T_{k3} which operates on O_1. Assume that at this point sub-transaction T_{k3} performs an operation that attempts to validate O_1 (such as an early validation or a merge-commit) and this validation fails. Under TFA, this would abort the root transaction T_k, including the work done by sub-transaction T_{k1}. N-TFA on the other hand only aborts as many sub-transactions are needed to resolve the conflict. In this case, only T_{k2} and T_{k3} need to abort. The transaction will be rolled back to the beginning of T_{k2}, such that the next operation performed is retrieving a new copy of the previously invalid object, O_1.

5. Analysis

5.1 N-TFA Properties

We show that N-TFA maintains the properties of the original TFA, in particular, opacity and strong progressiveness.

Opacity [18] is a **correctness criterion** proposed for memory transactions. A transactional memory system is opaque if the following conditions are met:

- Committed transactions appear to execute sequentially, in their real-time order.
- Any modifications done by aborted or live transactions to the shared state are never observed by any other transaction.
- All transactions observe a consistent view of the system at all times.

Theorem 5.1. *N-TFA ensures opacity.*

Proof. The proof for opacity in TFA can be trivially extended to cover N-TFA. The real-time ordering condition is satisfied as shown in [17], because changes made to objects by a transaction are not exposed to other unrelated transactions until the outermost transaction's commit phase, when the ordering is ensured through the usage of locks. Within a transaction, sub-transactions execute serially. There is no need to discuss the ordering of sub-transactions of different top-level transactions: they are effectively invisible to each other.

Uncommitted changes within a transaction are isolated from outside transactions through the use of a write-buffer, just as in TFA. Sub-transactions are executed serially and therefore always observe the correct values. The second condition for opacity is thus satisfied.

Transactions always observe a consistent system state. When N-TFA loads a new object with a version newer than the outermost transaction's starting time, it validates all objects observed by any child sub-transaction. This behavior is identical to the original TFA, satisfying the third condition for opacity. □

Strong Progressiveness is TFA's **progress property**. On a transactional memory system, strong progressiveness implies the following:

- A transaction without any conflicts must commit.
- Among a set of transactions conflicting on a single shared object, at least one of them must commit.

Theorem 5.2. *N-TFA ensures strong progressiveness.*

Proof. This follows immediately from the proof that TFA is strongly progressive [17], because the behavior of top-level transactions is identical for both TFA and N-TFA. This is because, sub-transactions as implemented by N-TFA do not introduce any operations that can disturb progress:

- External transactions are not affected because no objects are changed and the node-local clocks are not incremented upon merge model commits.
- Sub-transactions are aborted and retried such that any invalid objects will be re-opened on retry.
- After a validation procedure, no invalid objects will remain in a transaction that does not abort.

□

5.2 Effect on performance

There are two possible performance benefits of N-TFA over the original TFA. First, N-TFA introduces the possibility of partial rollback, when a conflicting sub-transaction can be retried by itself, without aborting its parent. This implies that less work needs to be retried and should lead to increased performance.

A second possible performance benefit due to N-TFA w/V is the fact that conflicts can be detected earlier. Under TFA, objects are validated at commit time and when remote commits are detected by comparing local and remote clock values. N-TFA w/V additionally validates the objects at sub-transaction commit time. In order to reason whether this can lead to any performance benefit, we need to consider several questions:

- In what cases is the number of validations performed under TFA too small and thus the additional validations of N-TFA would improve performance?
- In what cases TFA performs enough validations such that any extra validations are not warranted for?
- Is the cost of the validation operation small enough to warrant introducing extra validations?

Intuitively, if commits occur often in the system, early validations are also frequent, thus negating the possible benefits of N-TFA. This is linked to the proportion of read-only transactions in the workload: greater the number of read-only transactions, greater should be the benefit of N-TFA w/V. It is still unclear whether N-TFA w/V is to be preferred at all when compared to N-TFA w/o V.

Other parameters are less straightforward to reason about. Such parameters that may have effects are: number of objects loaded by a sub-transaction, amount of processing done by a sub-transaction, depth of nesting, number of children a parent transaction may have, etc. The effects of these parameters can however be evaluated experimentally.

6. Experimental evaluation

We implemented N-TFA in order to quantify the performance impact of closed nesting in the distributed STM environment. We also seek to identify the kinds of workloads that are most appropriate for using closed nesting instead of flat transaction.

6.1 Implementation details

We implemented N-TFA by extending HyFlow, the Java Distributed STM framework proposed by Saad [19], which in turn is based on the Deuce STM [20]. HyFlow's architecture is modular, allowing pluggable support for lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols. HyFlow doesn't require compiler or JVM support, and presents to the programmer a clean interface based on annotations.

In order to support nesting, we inserted an additional layer of logic between the code of a parent transaction and the code of its sub-transactions. This extra logic handles the partial rollback mechanism and the merge-commits. It was designed to be flexible and to provide support for all three types of nesting: flat, closed and open. While it supports flat nesting and could, in theory, be automatically inserted for every function call within a transaction, doing so would unnecessarily degrade performance.

Instead, we chose to manually insert this logic only in those locations where spawning sub-transactions is desirable. The downside of this approach, at least for now, is that the programmer must acknowledge the difference between regular function calls and closed-nested sub-transactions and write his or her code accordingly. Regular function calls must pass a transactional context variable as an additional parameter (compared to non-transactional code). Methods that spawn sub-transactions do not need any extra parameters, but must include the code implementing the extra logic mentioned above. (Modifying the automatic instrumentation present in both Deuce STM and HyFlow to support this behavior is future work.)

6.2 Experimental settings

The performance of N-TFA was experimentally evaluated using a set of distributed benchmarks consisting of two monetary applications (bank and loan) and three micro-benchmarks (linked list, skip list, and hash table). We record the throughputs obtained when running the benchmarks with the same set of parameters under both closed and flat nesting, and we report on the relative difference between them. Most of our figures relay two values: the average and the maximum. The average value represents multiple runs of the experiment under increasing number of nodes, while the maximum settles on the number of nodes that gives the best results in favor of closed nesting. Unfortunately, we cannot compare our results with any competitor D-STM, as none of the two competitor D-STM frameworks that we are aware of support closed nesting or partial aborts [21, 22].

We targeted the effect of several parameters:

- Ratio of read-only transactions to total transactions (denoted in figure legends with %) was already discussed in Section 5.2.
- Length of transaction in milliseconds (L) is used in some tests to simulate transactions that perform additional expensive processing and therefore take longer time.

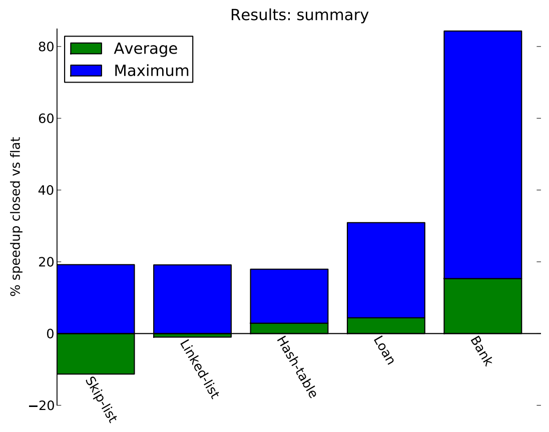

Figure 5. Performance change by benchmark.

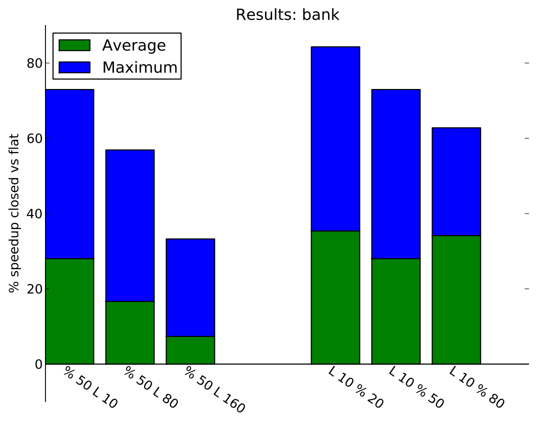

Figure 6. Bank monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions.

- Number of objects (o) is used to control the amount of contention in the system. The meaning of this number is benchmark-dependent.
- Number of calls (c) controls the number of operations performed per test. In closed-nested tests, this directly controls the number of sub-transactions.

Our experiments were conducted using up to 48 nodes. Each node is an AMD Opteron processor clocked at 1.9GHz. We used the Ubuntu Linux 10.04 server operating system and a network with 1ms end-to-end link delay. Each node spawns transactions using up to 16 parallel threads, resulting in a maximum of 768 concurrent transactions. While this number may not seem high, we focused on high-contention scenarios by only allowing a low number of objects in the system.

6.3 Experimental results

Our experiments have shown that N-TFA w/V performs consistently worse than N-TFA w/o V. The cost of object validations is too great to justify introducing more validations than absolutely necessary, even when all objects are validated in parallel. Thus, from this

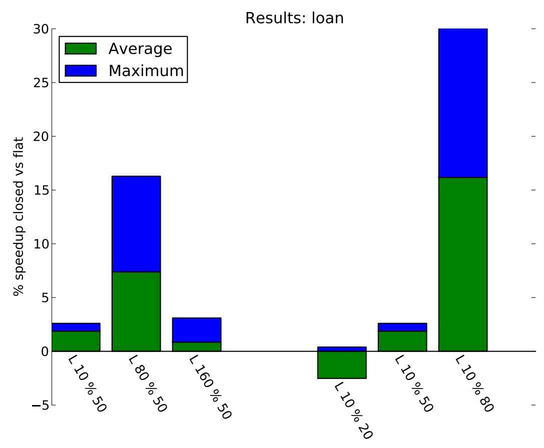

Figure 7. Loan monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions.

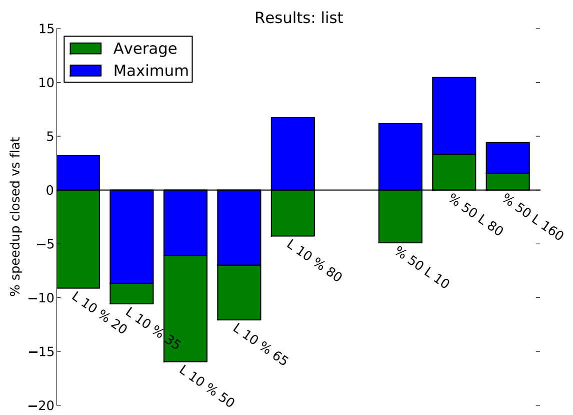

Figure 8. Linked-list micro-benchmark. First group varies read-ratio for short transactions. In the second group, transaction length is varied.

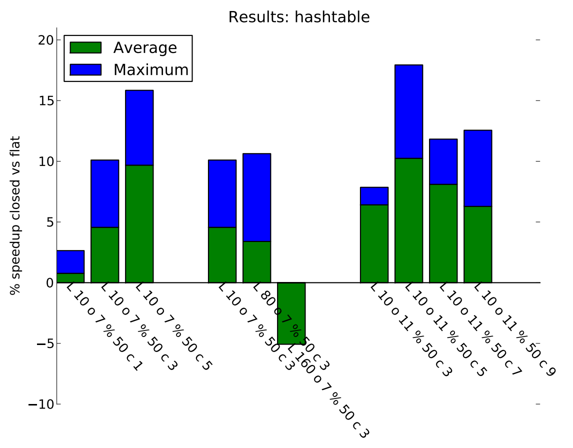

Figure 9. Hash-table micro-benchmark. First group shows increasing number of calls on hash-tables with 7 buckets. Second group shows the effect of increasing transaction length. Third group shows increasing number of calls on hash-tables with 11 buckets.

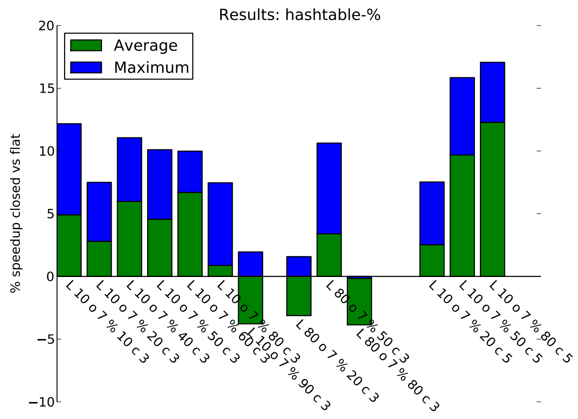

Figure 10. Hash-table read-ratio plot. First group shows the effect of increasing read-ratio on short transactions with 3 calls. Second group shows the effect of the same parameter on longer transactions. Third group targets transactions with 5 calls.

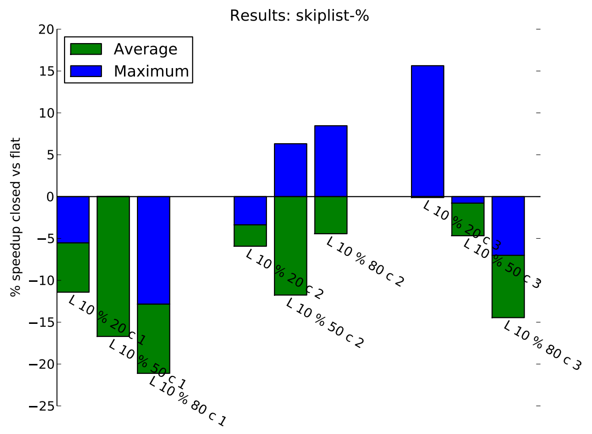

Figure 11. Skip-list micro-benchmark. Shows the effect of increasing read-ratio on tests with one, two and three operations performed in a transaction, respectively.

point on, we will only discuss *N-TFA without validation* and will refer to it as simply N-TFA. All results we present refer to N-TFA w/o V.

The results of our experiments are shown in Figures 5-13. Figure 5 shows a summary view of the improvement for each of our benchmarks. Figures 6-12 provide details on each of the benchmarks. Finally, Figure 13 looks at the scalability of N-TFA.

The performance of closed nesting varies significantly compared to flat nesting (see Figure 5). The single worst slowdown recorded was 42%, while the best speedup was 84%. Across all experiments, closed nesting illustrated 2% (on average) faster than flat nesting. However, the performance improvements depend strongly on the workload. Within our benchmarks, closed nesting performed worst for Skip-list (10.4% average slowdown) and best for Bank (15.3% average speedup).

These results lead us to believe that in workloads where each transaction accesses many different objects (like in Linked-list and Skip-list), closed nesting will be slower than flat transactions. On

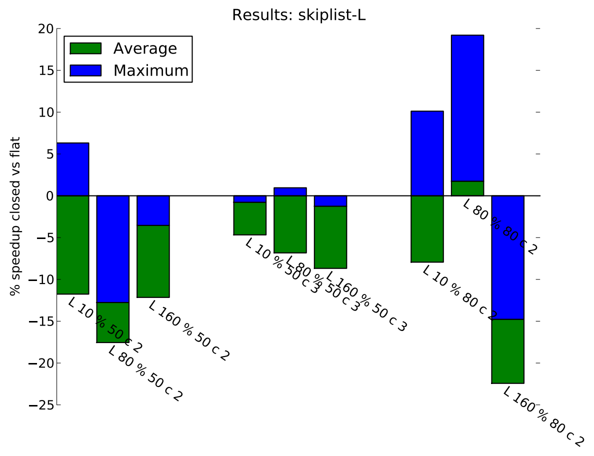

Figure 12. Skip-list micro-benchmark. Shows the effect of increasing transaction length. First group contains transactions with 2 calls. For the second group, the number of calls is 3. The last group has 80% reads.

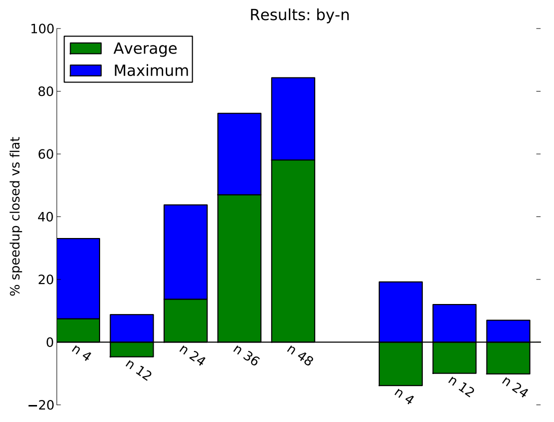

Figure 13. Effect of number of nodes participating in the experiment. First group shows the Bank benchmark with 16 threads/node. Second group shows the Skip-list micro-benchmark, with 4 threads/node.

the other hand, in workloads where transactions access few objects (like Bank, Loan and Hash-table), greater benefit can be obtained from closed nesting.

The most reliable parameter to influence the behavior of closed nesting appears to be the number of calls. In both Hash-table (Figure 9 groups 1 and 3) and Skip-List (Figure 11 between groups), we observe that the best performance is achieved with around 2-5 calls per transaction (workload dependent), after which it declines.

The other parameters that we observed (read ratio and transaction length) did not lead to any consistent trends. In some cases, increased read-ratio lead to better performance (e.g. Loan in Figure 7 group 2 and Hash-table with $c = 5, o = 7$ in Figure 10 group 3). Other cases showed a sweet spot in the middle of the range (Hash-table with $c = 3, o = 7$ in Figure 10 groups 2 and 3). Yet other cases show the opposite effects: performance negatively correlated with read-ratio on Skip-list (see Figure 12 groups 1 and 3), or worst performance in the middle of the range (Skip-list in Figure 12 group 2, Bank in Figure 6 group 2 and, most obviously, Linked-list in Figure 8 group 1). Transaction length has a similar unpredictable influence: negative correlation on Bank (Figure 6 group 1) and Hash-table (Figure 9 group 2), middle range peak on Loan (Figure 7 group 1) and middle range dip on Skip-list (Figure 12 group 1).

The *number of objects* parameter was only varied in one benchmark (Hash-table), so we cannot formulate any trends. This parameter did not apply in other benchmarks such as as Linked-list and Skip-list. In our particular case we observe that closed nesting seems to benefit somewhat from the reduced contention enabled by more hash buckets (Figure 9 between groups 1 and 3).

From the experiment to evaluate closed nesting's scalability (Figure 13), we observe that the performance drops with increasing nodes until about 19 concurrent transactions per object (as seen on Bank in group 1: 12 nodes $\times$ 16 threads / 10 objects). After that threshold, closed nesting performs increasingly better than flat nesting.

7. Future Work

An important direction for future work is to improve average-case performance. One possible approach is to consider checkpoints for performing partial aborts [15]. Checkpoints may prove more flexible than closed nesting: it may be possible to tune (manually or automatically) the locations of inserted checkpoints and thereby improve the performance gained from partial aborts.

Another direction of future work is to reduce the incidence of conflicts across the board by allowing transactions to read from older versions of objects – i.e., combining multi-version (distributed) concurrency control with closed nesting.

8. Conclusions

We presented N-TFA, an extension of the Transactional Forwarding Algorithm that implements closed nesting in a Distributed Software Transactional Memory system. N-TFA guarantees opacity and strong progressiveness. We implemented N-TFA in the HyFlow D-STM framework, thus providing (to the best of our knowledge) the first-ever D-STM implementation to support closed nesting. Our N-TFA implementation, although is on average only 2% faster than flat transactions, enables up to 84% speedup in certain cases.

We determined that closed nesting best applies for simple transactions that access few objects. The number of simple sub-transactions is important for the performance of closed-nesting, and we found that N-TFA performs best with 2-5 sub-transactions. N-TFA scales better than TFA, although the performance dips at around 19 concurrent transactions per object.

References

[1] M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss, Transactional memory: Architectural support for lock-free data structures, in in Proceedings of the 20th Annual International Symposium on Computer Architecture, 1993, pp. 289300

[2] D. Dice, O. Shalev, and N. Shavit, Transactional Locking II, in In Proc. of the 20th Intl. Symp. on Distributed Computing, 2006.

[3] N. Shavit and D. Touitou, Software transactional memory, in Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing, ser. PODC 95. New York, NY, USA: ACM, 1995, pp. 204213. [Online]. Available: http://doi.acm.org/10.1145/224964.224987

[4] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, LogTM: Log-based transactional memory, in In Proc. 12th Annual International Symposium on High Performance Computer Architecture, 2006.

[5] M. Herlihy and Y. Sun, Distributed transactional memory for metric-space networks, in In Proc. International Symposium on Distributed Computing (DISC 2005). Springer, 2005, pp. 324338

[6] Bo Zhang, Binoy Ravindran, Dynamic analysis of the Relay cache-coherence protocol for distributed transactional memory, in IPDPS 10. Washington, DC, USA: IEEE Computer Society, 2010.

[7] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo, Cloud-TM: harnessing the cloud with distributed transactional memories, SIGOPS Oper. Syst. Rev., vol. 44, pp. 16, April 2010

[8] T. Harris, J. Larus, and R. Rajwar, Transactional Memory, 2nd edition, Synthesis Lectures on Computer Architecture, vol. 5, no. 1, pp. 1263, 2010. [Online]. Available: http://www.morganclaypool.com/doi/abs/10.2200/S00272ED1V01Y201006CAC011

[9] J. E. B. Moss and A. L. Hosking, Nested transactional memory: Model and architecture sketches, Science of Computer Programming, vol. 63, no. 2, pp. 186 201, 2006. Special issue on synchronization and concurrency in object-oriented languages.

[10] J. E. B. Moss. Nested Transactions: An Approach to Reliable Distributed Computing. MIT Press, March 1985.

[11] Gray, J., and Reuter, A. Transaction Processing: Concepts and Techniques. Morgan Kaufmann, 1993.

[12] 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, SIGOPS Oper. Syst. Rev., vol. 40, pp. 359370, October 2006.

[13] J. E. B. Moss, Open nested transactions: Semantics and support, in Workshop on Memory Performance Issues, 2006.

[14] K. Agrawal, I.-T. A. Lee, and J. Sukha, Safe open-nested transactions through ownership, SIGPLAN Not., vol. 44, pp. 151162, February 2009.

[15] E. Koskinen and M. Herlihy, ”Checkpoints and continuations instead of nested transactions”, in Proc. SPAA, 2008, pp.160-168.

[16] M. M. Saad, ”HyFlow: A High Performance Distributed Software Transactional Memory Framework”, Master’s Thesis, ECE Dept., Blacksburg, VA, USA, April 2011.

[17] M. M. Saad and B. Ravindran. Transactional Forwarding Algorithm: Technical Report. Technical report, ECE Dept., Virginia Tech, January 2011

[18] R. Guerraoui and M. Kapalka, Opacity: A Correctness Condition for Transactional Memory, EPFL, Tech. Rep., 2007.

[19] M. M. Saad and B. Ravindran. ”Hyow: A high performance distributed software transactional memory framework”. In HPDC ’11: Proceedings of the 20th IEEE International Symposium on High Performance Distributed Computing, San Jose, California, USA, June 2011.

[20] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In MultiProg 2010: Third Workshop on Programmability Issues for Multi-Core Computers, 2010

[21] A. Bieniusa and T. Fuhrmann. ”Consistency in hindsight: A fully decentralized STM algorithm,” Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on , vol., no., pp.1-12, 19-23 April 2010, doi: 10.1109/IPDPS.2010.5470446 . Implementation available at https://proglang.informatik.uni-freiburg.de/projects/dstm/

[22] N. Carvalho, P. Romano, and L. Rodrigues. A Generic Framework for Replicated Software Transactional Memories (short paper). The 10th IEEE International Symposium on Network Computing and Applications (IEEE NCA11), Cambridge (MA), USA, August 2011. Implementation available at http://code.google.com/p/genrstm/