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
Transaction Memory (TM) is a promising model for program- ming concurrency 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 composable 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 composable 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:

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.

Besides providing support for code compositability, 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). Abort 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 HyFlow1 and the previously proposed TFA algorithm to support closed nesting. The resulting algorithm is named Nested Transac-

---

1 HyFlow is available on-line, as an open-source project, at http://hyflow.org
tional 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 generalization 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 rigidity 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 1, 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 $LC_1 = 29$, $N_3$’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: $\{N_1, N_2, \ldots, N_n\}$. 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 $O = \{O_1, O_2, \ldots\}$ 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 $T = \{ T_1, T_2, \ldots \}$ 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 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 should 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.

In case one or more objects are detected as invalid, the upper-most 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 2). The top-level transaction \( T_1 \) 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

N-TFA maintains the properties of the original TFA, in particular, opacity and strong progressiveness. The proofs are straightforward extensions to the original TFA proofs, and are sketched below.

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.
- 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 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 3-11. Figure 3 shows a summary view of the improvement for each of our benchmarks. Figures 4-10 provide details on each of the benchmarks. Finally, Figure 11 looks at the scalability of N-TFA.
The performance of closed nesting varies significantly compared to flat nesting (see Figure 3). 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 Linked-list and Skip-list), closed nesting will be slower than flat transactions. On 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 7 groups 1 and 3) and Skip-List (Figure 9 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 5 group 2 and Hash-table with $c = 5, o = 7$ in Figure 8 group 3). Other cases showed a sweet spot in the middle of the range (Hash-table with $c = 3, o = 7$ in Figure 8 groups 2 and 3). Yet other cases show the opposite effects: performance negatively correlated with read-ratio on Skip-list (see Figure 10 groups 1 and 3), or worst performance in the middle of the range (Skip-list in Figure 10 group 2, Bank in Figure 4 group 2 and, most obviously, Linked-list in Figure 6 group 1). Transaction length has a similar unpredictable influence: negative correlation on Bank (Figure 4 group 1) and Hash-table (Figure 7 group 2), middle range peak on Loan (Figure 5 group 1) and middle range dip on Skip-list (Figure 10 group 1).

The number of objects parameter was only varied in one benchmark (Hash-table), so we cannot formulate any trends. This parameter was only varied in one benchmark (Hash-table), so we cannot formulate any trends. This parameter was only varied in one benchmark (Hash-table), so we cannot formulate any trends.
rameter 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 7 between groups 1 and 3).

From the experiment to evaluate closed nesting’s scalability (Figure 11), 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 × 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.

Acknowledgments

This work is supported in part by NSF CNS 0915895, NSF CNS 1116190, and NSF CNS 1130180.

References


