On Improving Distributed Transactional Memory Through Nesting and Data Partitioning

Alexandru Turcu

Preliminary Examination Proposal submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
Chao Wang
Jules White
Robert P. Broadwater
Eli Tilevich

November 27, 2012
Blacksburg, Virginia

Keywords: Distributed Transactional Memory, Nested Transactions, On-Line Transaction Processing, Distributed Systems

On Improving Distributed Transactional Memory Through Nesting and Data Partitioning

Alexandru Turcu

(ABSTRACT)

Distributed Transactional Memory (DTM) is an emerging, alternative concurrency control model that aims to overcome the challenges of distributed-lock based synchronization. DTM employs transactions in order to guarantee consistency in a concurrent execution. When two or more transactions conflict, all but one need to be delayed or rolled back.

Transactional Memory supports code composability by nesting transactions. Nesting however can be used as a strategy to improve performance. The closed nesting model enables partial rollback by allowing a sub-transaction to abort without aborting its parent, thus reducing the amount of work that needs to be retried. In the open nesting model, sub-transactions can commit to the shared state independently of their parents. This reduces isolation and increases concurrency. Checkpointing is an alternative model to closed nesting.

In this thesis we propose three extensions to the existing Transactional Forwarding Algorithm (TFA). Our extensions are N-TFA, TFA-ON and TFA-CP, and support closed nesting, open nesting and checkpointing, respectively. We implement these algorithms in a Java DTM framework and evaluate them. This represents the first study of transaction nesting in the context of DTM, and contributes the first DTM implementation which supports closed nesting, open nesting or checkpointing.

Closed nesting through our N-TFA implementation proved insufficient for any significant throughput improvements. It ran on average 2% faster than flat nesting, while performance for individual tests varied between 42% slowdown and 84% speedup. The workloads that benefit most from closed nesting are characterized by short transactions, with between two and five sub-transactions.

Open nesting, as exemplified by our TFA-ON implementation, showed promising results. We determined performance improvement to be a trade-off between the overhead of additional commits and the fundamental conflict rate. For write-intensive, high-conflict workloads, open nesting may not be appropriate, and we observed a maximum speedup of 30%. On the other hand, for lower fundamental-conflict workloads, open nesting enabled speedups of up to 167% in our tests.

Transaction checkpoints, using our TFA-CP implementation, showed 1-10% benefit over flat transactions when all the overheads are factored out. When also including the effects of continuations, we obtained contradictory results, ranging from 2x slowdown up to 6x speedup.

In addition to the three algorithms, we also develop Hyflow2, a high-performance DTM framework for the Java Virtual Machine, written in Scala. It has a clean Scala API and a compatibility Java API. Hyflow2 was on average two times faster than Hyflow on high-contention workloads, and up to 16 times faster in low-contention workloads.

Our major proposed post-preliminary work is in the area of automatic data partitioning. An existing technique generates partitioning schemes that maximize local, single-node transactions which are fast and minimize distributed transactions. We plan to extend this technique with support for independent transactions, a recently proposed, light-weight distributed

transaction model. After that, we plan to develop a mechanism to automatically convert a generic atomic-block into specialized code that is partition-aware and uses the most efficient transaction model applicable. Finally, want to convert blocks that require using the two-phase commit coordinated transaction model, into multiple separate transactions that use more efficient models.

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

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Until recently, CPU manufacturers were able to increase the performance of their devices by running them at ever higher frequencies. However around 2004, this trend became unsustainable, and adding multiple processing cores on the same chip became the new standard. Software developers were forced to embrace *concurrency* as the means for their programs to run faster, or perform more advanced processing in a short time.

Handling concurrency correctly is a difficult task. The simple strategy of using a single global lock may be easy to implement, but it hardly brings any performance benefits, effectively executing all *critical sections* sequentially. Using fine-grained locks to protect individual pieces of data enables the much desired *scalability*, but is inherently error prone. Any mistakes can lead to hard to trace problems such as *deadlocks* and *race-conditions*. Moreover, due to the randomness of concurrency, these problems may not manifest during testing, misleading the programmer to ship a defective product.

Using locks also makes code composition difficult. Suppose a library uses locks to control access to a hash-table and the programmer needs to apply two hash-table operations in an atomic manner in order to hide the intermediary state from other threads. In this situation, he or she may introduce extra locks protecting both data structures at the same time, but this can lead to race conditions or loss of performance if not implemented carefully. Alternatively, the programmer may try to expose the implementation of the library in order to understand and then extend its locking mechanism, but again, this is error prone and moreover it contradicts the encapsulation concept of object-oriented programming.

1.1 Software Transactional Memory

Transactional Memory (TM) was proposed to bring a successful abstraction from the database community, the *transaction*, into regular multi-processor programming [28]. Transactions

were originally developed to provide four important properties: atomicity, consistency, isolation and durability (the ACID properties). In this context, atomicity (or failure atomicity) means that the operations making up a transaction either all execute to completion, or they appear as if they never started executing. This effectively prevents a transaction from executing partially and leaving the system in an inconsistent state. The isolation property prevents a transaction from observing the intermediary states that another parallel transaction may produce while running. Thus, ACID transactions are *serializable*: although they may execute concurrently, the overall effect is the same as if they executed serially, one after another, without any overlap. The A and I properties give the illusion that a transaction either executes at a single instant in time (i.e. **atomic execution**), or not at all.

When a transaction executes successfully, it is said to **commit**. Otherwise, it **aborts** and leaves no evidence that it ever started executing. If a transaction has to abort, it may retry a fixed number of times.

Transactional Memory can be supported in hardware (Hardware Transactional Memory — HTM) [36, 49], in software (Software Transactional Memory — STM) [58] or a combination of the two (hybrid TM) [17]. STM has the unique advantage of being able to run on commodity hardware. The drawback however is a degradation in performance, as reads and writes aren't simple memory operations anymore, but complex functions that implement the TM protocols.

During a transaction's execution *conflicts* may take place. A conflict is said to occur when two transactions try to access the same memory location, and at least one of those accesses is a write.

Conflict detection can take place at different times. Using *pessimistic concurrency control* the conflicts are detected at the time the memory operations are performed, while under *optimistic concurrency control* the detection is postponed and conflicting transactions are allowed to keep running, but not to commit. The two approaches work best in different situations: the optimistic strategy gives better results when conflicts are rare (*low contention* workload) whereas the pessimistic one performs better under *high contention* workloads.

Once a conflict is detected, it has to be resolved. In order to resolve the conflicts, there are two alternatives:

- *Delay* one of the transactions in order to allow the other one to complete, then continue with its execution. (This only works for eager conflict detection.)
- *Abort* one of the transactions and retry it later.

In Transactional Memory, *version management* refers to the methods employed by the system for managing writes to the memory. A TM system uses *eager version management* or *direct update* when it writes directly to memory [36]. The previous memory content is recorded in an undo-log, which is later used to *roll back* the transaction in the event of an abort.

In such systems, the conflict detection scheme employed must be pessimistic, because write operations that cause conflicts should not be executed.

The alternative is *lazy version management* or *deferred update*. Write operations do not directly affect the main memory, but instead are recorded in a transaction's private redo-log. As a consequence, read operations must also check the redo-log in order to make sure they observe the most recent value of the desired memory location. Upon commit, the changes recorded in the redo-log are saved to the shared memory.

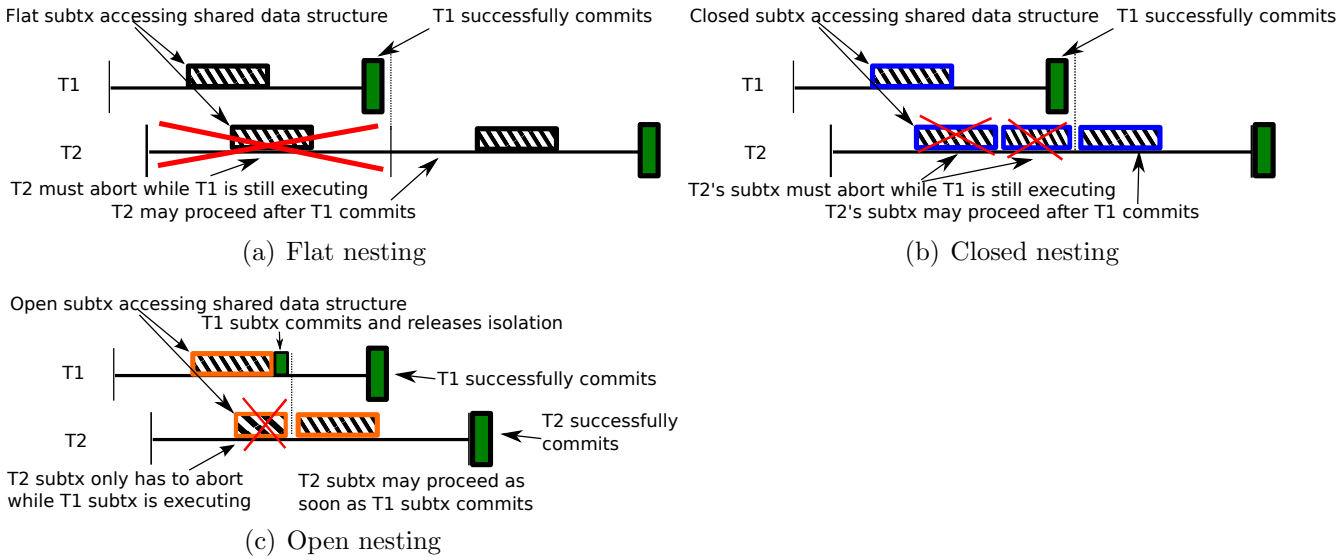
To solve the code composability problem, TM uses the concept of nesting. A transaction is nested when it is enclosed within another transaction. The outer transaction is called parent and the inner transaction is the child. Child transactions can also have their own children, resulting in a tree-like structure. Transactions may have multiple children, leading to inner transactions that can execute concurrently [63]. However, in this work we will only consider linear nesting [41], where each transaction can only have at most one child, and the bottom-most transaction in the chain is the only active transaction.

Three types of nesting models have been previously studied [26, 40]: 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 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
: In 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



(a) Flat nesting

(b) Closed nesting

(c) Open nesting

Figure 1.1: Simple example showing the execution time-line for two transactions under flat, closed and open nesting.

> the programmers. Therefore, open nesting must be used with extreme caution, and is generally only recommended for experts.

We illustrate the differences between the three nesting models in Figure 1.1. Here we consider two transactions, which access some shared data-structure using a sub-transaction. The data-structure accesses conflict at the memory level, but the conflict is not fundamental (we will explain fundamental conflicts later, in Section 3.1.2), 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).

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). 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). In both these scenarios, performance can be improved with nesting by aborting and retrying only the inner-most sub-transaction.

1.2 Distributed Transactional Memory

Lock-based concurrency is even more challenging in the distributed setting. To address that, STM has been extended to distributed systems. Distributed Transactional Memory (DTM) provides the same easy-to-use abstraction of transactions. DTM works can be classified into cache-coherent DTM and cluster-DTM. Cache-coherent DTM [29, 52] maintains copies of the data at the nodes that requires it. A directory protocol is usually employed to locate the primary copy. When a transaction that modifies a data object commits, it invalidates all previous copies of the data and effectively migrates the object to its own node. This approach was proposed by Herlihy and Sun and is called the data-flow execution model [29].

Alternatively, Cluster DTM [13, 7] replicates the data on a set of closely coupled machines. The cluster usually employs a group communication protocol [50], a consensus protocol (i.e., Paxos), or a lease mechanism [34] for ensuring consistency across replicas.

1.3 Performance in DTM

Performance in DTM is paramount. As a new abstraction trying to replace its predecessor, DTM must at least match the performance characteristic of distributed locking. As such, a majority of the research focused on DTM is aiming for increasing its performance. This quest can be observed in research on directory protocols [66, 4], Multi-Version Concurrency Control [35], scheduling transactions [5, 31], transaction protocols [56, 6, 55] and DTM implementations [54, 11, 13].

While transaction nesting was studied extensively in non-distributed TM ([41, 42, 2, 37, 39, 40, 63]), this topic was never touched upon in the DTM literature. Nesting can potentially help improve DTM performance (as it does for non-distributed TM), and we consider important to evaluate any such improvements.

Closed nesting, as a generic partial rollback mechanism, reduces the amount of work that needs to be retried in case of transaction aborts. In the distributed context, such work usually involves opening remote objects — an inherently slow operation due to network latency. For these reasons, we expect to see closed nesting improving DTM performance. We also seek to identify what factors and workload characteristics have an influence on performance.

Open nesting reduces isolation by releasing memory locks early, and thus allows more trans-

actions to execute concurrently without aborting. We expect this will directly translate into greater system throughput (as measured in transactions per second). We again seek to identify influencing factors.

Transaction checkpointing [33] is another partial rollback mechanism, and an alternative to closed nesting. It allows greater flexibility in choosing the rollback target, which should have a positive influence on transaction throughput. On the downside, continuations, the mechanism used to implement checkpoints, have greater overheads. We thus seek to quantify the benefits and overheads of transaction checkpointing, and determine its net effect on throughput.

Finally, irrespective of which algorithms are being used, DTM performance also depends on the efficiency of the system's implementation. Being dissatisfied with our current framework, Hyflow, we develop a new framework from scratch, which we named Hyflow2. In implementing Hyflow2, we focus on performance, ease of use, and rapid prototyping.

1.4 Summary of Current Research Contributions

We design *N-TFA*, an extension to the existing Transactional Forwarding Algorithm (TFA) with support for closed nesting. We show that N-TFA maintains TFA's properties, in particular opacity and strong-progressiveness. We implement N-TFA in Hyflow, a Java DTM framework and evaluate it on a set of five micro-benchmarks. We find an average performance improvement of only 2% compared to flat nesting. Two of the benchmarks saw average performance downgrade (worst slowdown of 42%), while the maximum speedup was 84%. We observe that closed nesting applies best when transactions do not access many objects, and when the number of sub-transactions is between 2 and 5. To the best of our knowledge, this work contributes the first closed nesting implementation for DTM.

We then extend N-TFA to include support for open nesting. We named the resulting algorithm *TFA-ON*. We implement TFA-ON in Hyflow and evaluate it on a set of four micro-benchmarks. Our implementation enabled up to 30% speedup when compared to flat transactions, for write-dominated workloads and increased fundamental conflicts. Under reduced fundamental conflicts workloads, speedup was as high as 167%. We identified that the best speedups occur when transactions are composed of a high number of sub-transactions, and when fundamental conflict rates are low. We also confirm the expectation that high contention after open-nested sub-transactions leads to lower open nesting performance. To the best of our knowledge, this work contributes the first open nesting implementation for DTM.

Next we design TFA-CP, a variant of the TFA algorithm with support for partial rollback via transaction checkpoints. We implement TFA-CP in Hyflow2, using a non-standard JVM which supports continuations. We evaluate TFA-CP using three micro-benchmarks, while also taking measurements designed to evaluate the overheads of continuations. We find that partial rollback gives 1-10% performance improvement once all the overheads are factored

out. We however get contradictory results when including these overheads: on two benchmarks we obtain a significant net slowdown, while on the third benchmark we obtain 3-6x improvements that are unrelated to partial rollback. To the best of our knowledge, this is the first implementation of transaction checkpoints for DTM, and also for a Java TM system.

Finally we design and implement Hyflow2, a novel high-performance DTM framework for the JVM. Hyflow2 is written in Scala and provides a clean, easy-to-use API. Hyflow2 also provides an additional Java compatibility API. Hyflow2 is the first DTM implementation in Scala, with a Java compatibility API, and with support for features such as transaction nesting, checkpointing and conditional synchronization. Internally, Hyflow2 uses the actor model, an alternative concurrency model that employs state separation and message passing. At high contention, Hyflow2 proved on average 2x faster than Hyflow, with a peak of up to 7x at low node counts. At low contention however, Hyflow2 is consistently 8-15x faster.

1.5 Summary of Proposed Post-Prelim Work

After the Preliminary Examination, we propose to seek further DTM performance improvements by employing automatic data partitioning and replication. This technique was previously proposed for an SQL database [15], and uses the fact that distributed transactions are slower than local transactions. The technique proposes data partitioning schemes that maximize the percentage of local, single-node transactions, and was shown to be competitive with the best known manual partitioning schemes.

We aim to develop a partitioning method is aware of the newly proposed independent transaction model [14] (briefly described in Section 2.3). Thus, our partitioning scheme will have two goals. Firstly, it will favor local, single-node transactions over distributed transactions, and secondly, when distributed transactions are required, it will favor the independent transaction model over the more general two-phase commit (2PC) coordinated model. This effort will have a wide applicability to other areas such as traditional databases, transactional storage and new-generation SQL databases, and will not limited to DTM.

We further seek to investigate how to automatically translate a DTM transaction given in the form of an atomic block that ignores any data partitioning concerns, into a set of distributed sub-transactions based on the actual data partitioning scheme in use. This mechanism will help support automatic data migration, and will increase ease-of-use, as the programmer will not need to be aware of the underlying data partitioning.

As our final goal, we want to investigate whether a general, 2PC coordinated transaction could be broken down into several shorter transactions that run under the faster, single-node or independent models. If this transformation is possible, we want to evaluate its effects on consistency, and provide an implementation for doing it automatically.

1.6 Thesis Organization

This thesis proposal is organized as follows. In Chapter 2 we summarize relevant and related previous work. In Chapter 3 we describe our basic system model, our nesting model, and TFA, base algorithm for all our contributions. Chapter 4 introduces and evaluates N-TFA, our closed-nested extension to TFA. Open nesting and TFA-ON are discussed in Chapter 5. In Chapter 6 we present TFA-CP and we evaluate checkpointing as an alternative partial rollback mechanism. In Chapter 7 we introduce Hyflow2, our new-generation DTM framework. Chapter 8 concludes the thesis and further discusses proposed post-preliminary work.

Chapter 2

Previous and Related Work

2.1 Distributed Transactional Memory

DTM was first proposed by Herlihy and Sun [29] as an alternative to standard distributed transactions using *Two-Phase Locking* and *Two-Phase Commit Protocols* (2PC) as is standard in database environments. They use a *dataflow*-based approach where transactions execute on a fixed node while the data migrates to the transactions that requires it. One claimed advantage of this approach is that it does not require a distributed commit protocol, making successful commits fast. In order to manage the location of data, the authors propose a distributed cache-coherence protocol called Ballistic. This protocol, alongside a contention manager, manage data conflicts and ensure its consistency. On the downside, it relies on an existing distributed queuing protocol, Arrow [18], that does not take contention into account, and due to its hierarchical structure, scalability is limited — the entire structure needs rebuilding every time a node joins or leaves the network.

Zhang and Ravindran [66] developed the Relay protocol which takes transactional conflicts into account and scales better due its use of peer-to-peer data structures. The authors also introduce Location Aware Cache-coherence protocols (LAC, [65]), where nodes closer to the data (in terms of communication cost) are guaranteed to locate the object earlier. They show that LOC protocols, in conjunction with the optimal Greedy contention manager, improve the *makespan* competitive ratio, a measure of the efficiency of a transaction execution.

Unlike previous proposals, which do not tolerate unreliable links, Attiya et al. present Combine [4], a directory protocol that works even in the presence of partial link failures and non-FIFO message delivery. Combine is however still not network partition tolerant.

Bocchino et al. took an implementation based approach and developed Cluster-STM [7]. They observe that remote communication overheads are the main impediment for scalability, and thus try to make an appropriate set of design choices, sometimes different than other

cache-coherent STMs. Examples of such choice are aggregating the remote communication with data communication, and using a single access-set rather than separate read and write-sets.

Kotselidis et al. developed DiSTM [34], an DTM system optimized for clusters. DiSTM can be configured with three cache coherence protocols. TCC [24], an existing decentralized protocol, suffers from large traffic overheads at commit time, as transactions broadcast their read and write-sets. These overheads are avoided using two newly proposed lease-based protocols, at the expense of introducing lease bottlenecks and an additional validation step. In benchmarks, no one protocol achieved greater performance, but rather the best protocol choice depended was dependent on the amounts of contention and network congestion.

In contrast to cache-coherent DTM, replicated DTM stores multiple writable copies of the data, and this is a promising approach for achieving fault-tolerance. D2STM is the first replicated DTM system. Introduced by Couceiro et al [13], it provides strong consistency even in the presence of failures by using a non-blocking distributed certification scheme. This scheme is inspired by recent database replication research [43, 45], but employs Bloom filters in order to reduce the overheads of replica coordination, at the expense of an increased probability of false aborts.

Carvalho et al. introduce Asynchronous Lease Certification (ALC, [10]), a low-overhead mechanism for replica coordination, that enables up to ten times lower commit latencies. ALC relies on a group communication service providing *Atomic Broadcast* and *Uniform Reliable Broadcast* primitives [23], in order to acquire leases for the subset of replicated data that a transaction will modify.

Romano et al. report in [50] on implementing a web application using Distributed Transactional Memory, and the experience of its first two years in production. The authors make several important observations, such as the workload being comprised of only 2% write transactions, and the average write-set being orders of magnitude smaller than the average read-set. In [51], they show how DTM would be an appropriate programming model for applications running in cloud environments (i.e., clusters of hundreds of nodes or more), and point to several research directions that would help reach this goal.

A number of researchers focused on consistency criteria weaker than serializability in in order to improve DTM performance. In particular, Multi-Version Concurrency Control (MVCC) and its associated consistency criterion, Snapshot Isolation (SI) have the advantage of not generally having to abort read-only transactions. MVCC has been extensively studied in the database environments and multi-processor STMs [9, 48, 47].

Several DTM systems also use MVCC. ALC [10] relies on MVCC to enable only acquiring leases for writes. Peluso et al. introduce the GMU protocol [46], which is the first protocol to provide Snapshot Isolation and Genuine Partial Replication (i.e., only nodes replicating used data are involved in the transaction protocol). GMU relies upon several mechanisms. It employs a new scheme based on Vector Clocks (VC) to determine which version of an object

to be returned by a read operation, and to achieve agreement upon next object version and the VC value attached to committed transactions. Additionally, prepared transactions wait in a commit queue (sorted by a particular VC entry) before they are allowed to commit. The commit operation is effectuated in a standard Two Phase Commit (2PC) fashion. By disseminating the VC of the oldest transaction still running, old object versions can be safely garbage collected.

2.2 Nesting in Transactional Memory

Nested transactions (using closed nesting) originated in the database community and were thoroughly described by Moss in [38]. His work focused on the popular two-phase locking protocol and extended it to support nesting. In addition to that, he also proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection.

Open nesting also originates in the database community [21], and was extensively analyzed in the context of undo-log transactions and the two-phase locking protocol [64]. 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 [40]. 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 [39], explaining how using multiple levels of abstractions can help differentiate between fundamental and false conflicts and thus improve concurrency. Ni et al. also discuss the implications of open nesting in [42], and additionally provide the first open nesting implementation for STM.

Moravan et al. [37] 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 innermost sub-transaction. In this work, open nesting was only applicable to a few benchmarks, but it enabled speedups of up to 100%.

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

From a different perspective, Herlihy and Koskinen propose transactional boosting [27] 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 (or a suspension mechanism) and abstract locks. Boosting works with an existing concurrent data structure (which it treats as a black box), captures a different (possibly better) performance-complexity balance than pure open nesting, and is easier to use and reason about.

2.3 Other Unconventional Database Systems

While not considered Distributed Transactional Memory, a recent line of research is proposing a complete rewrite of conventional database systems. Stonebraker et al. argue [60] that conventional DBMS systems, while trying to be provide a solution applicable to a wide range of problems such as on-line transactional processing (OLTP), data-warehousing, and stream processing, in reality they do a bad job at all such problems. In [25] the authors analyze how is the CPU time spent in a conventional DBMS and find out that only a fraction of time is used to do useful work, while the vast majority of time is spend in tasks such as resource management, locking and synchronization. Thus, they propose a new architecture that specifically targets OLTP workloads, and they implement H-Store [30] to demonstrate its superiority. H-Store stores all the data in the main-memory. Durability is achieved using network-based replication instead of disk-backed logs. Data is horizontally partitioned across multiple sites, and each such repository is backed up by a number of replicas. At each repository, transactions are processed in a single thread. A majority of transactions in common OLTP workloads only require data from a single site. Such single-repository transactions are run by H-Store without any concurrency control. H-Store is able to outperform a leading commercial DBMS by almost two orders of magnitude, while still guaranteeing strong consistency (serializability).

Cowling and Liskov improve upon H-Store by introducing independent transactions in Granola [14]. Under this model, single-shot distributed transactions can execute without coordination between nodes on the critical path as long as the same commit/abort decision can be individually taken by each repository without an agreement protocol. Read-only and non-aborting single-shot transactions are good candidates for this model. Repositories vote on a proposed time-stamp for each transaction and then execute all transactions in time-stamp order. Voting can be handled asynchronously in a thread separate to the thread executing transactions.

Aguilera et al. tackle a similar problem and propose minitransactions [3] as a way to achieve good performance and scalability. Minitransactions are the result of applying a number of optimizations to the standard two-phase commit protocol. They significantly reduce the number of round-trips required to commit a transaction, at the expense of having a severely constrained transaction primitive (essentially a multi-object compare-and-set) that requires

all its accessed data to be specified in advance. Using this primitive, the authors quickly implemented a cluster file-system and a group communication service.

Corbett et al. present Spanner, a scalable, multi-version, globally distributed and synchronously replicated database in use at Google [12]. Spanner manages to guarantee external consistency while distributing data at data-centers around the world by exposing time uncertainty and including it in the transaction protocol. Spanner however requires dedicated timekeeping hardware such as GPS devices and atomic clocks.

We summarize relevant related DTM work in Table 2.1, along with some representative STM works.

	STM						DTM						Other			
	McRT-STM [57]	Deuce STM [32]	Scala STM [8]	SwissTM [20]	RingSTM [59]	NOrec [16]	Cluster-STM [7]	DiSTM [34]	D2STM [13]	ALC / D2STM [10]	GMU / Infinispan [46]	TFA / Hyflow [52]	Sinfonia	H-Store	Granola	Current Work
Language	C++	Java	Scala	C++	C	C	C	Java	Java	Java	Java	Java	C++	C++	Java	Java/Scala
Serializable	Y	Y	Y	Y	Y	Y	Y	Y				Y	Y	Y	Y	Y
MVCC									Y	Y	Y					
Replicated								Y	Y	Y	Y		Y	Y	Y	
Fault-Tolerant									Y	Y	Y		Y	Y	Y	
Closed Nesting	Y			-		Y										Y
Open Nesting																Y
Checkpoints																Y
Strong Atomicity	?		Y		?	?		?		Y	?		Y	Y	Y	Y
Conditional Sync	?					Y										Y

Table 2.1: Summary of related work

Chapter 3

Preliminaries and System Model

3.1 System Model

As in [29], we consider a distributed system with a set of nodes $\{N_1, N_2, \cdots\}$ that communicate via message-passing links.

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

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

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.

Our implementation executes transactions using the redo-log approach. During the transaction's execution, all object accesses are stored in two temporary buffer called the *read-set* and the *write-set*. At commit-time, if the transaction is still valid, changes are propagated to the shared state.

When an object is read from the globally committed memory (i.e., the shared state), its value is stored in the read-set. Similarly, when an object is written, the value is temporarily buffered in the write-set and does not affect the shared state. 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.

A detailed description of the basic protocol (TFA) will be given in Section 3.2.

3.1.1 Nesting Model

Our nesting model is based on Moss and Hosking [40]. 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.

With transactional nestings, let $parent(T_k)$ denote the parent (enclosing) transaction of a transaction T_k. A root transaction has $parent(T_k) = \emptyset$. Each transaction may only have one active child, i.e. parallel nested transactions are outside the scope of this work. A parent transaction may execute sub-transactions using any of the three nesting models: flat, closed, or open. We denote this by defining the *nesting model* of any sub-transaction T_k:

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

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

Let's briefly examine how the four important transactional operations behave in the context of transaction nesting. As mentioned above, each transaction maintains a redo-log of the operations it performs in the form of a read-set and a write-set. Reading an object O_k first looks at the current transaction's (T_k) read and write-sets. If a value is found, it is immediately returned. Otherwise, depending on the transaction's nesting model, two possibilities arise:

- For $nestingModel(T_k) = OPEN$, the object is fetched from the globally committed memory. This case includes the root transaction.
- For $nestingModel(T_k) = CLOSED$, the read is attempted again from the context of $parent(T_k)$.

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

The commit of a closed-nested transaction T_k merges $readset(T_k)$ into $readset(parent(T_k))$ and $writeset(T_k)$ into $writeset(parent(T_k))$. 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 5.1.2.

3.1.2 Multi-Level Transactions

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

It is therefore desirable to separate transactions into multiple levels of abstraction. By making the operations shorter at the lower memory level, isolation at that level is released earlier, thus enabling increased concurrency. This breaches serializability and must be used with care. 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 [64]). 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 [27]. For example, in deleting two different elements from a set, the final state is the same regardless of which of the deletes executes first. In contrast, inserting and deleting the same item from a set can not commute: which of the two operations executes last will determine the state of the set.

In order to achieve level-by-level serialization, non-commutative higher-level operations, when executed by two concurrent transactions, must conflict. Such a conflict is called *fundamental*, as it is essential for a correct execution. One such mechanism for detecting fundamental 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.

Abstract locks can be used to implement read/write locking, mutual exclusion, or even more complex scenarios, such as compatibility matrices (for encapsulating higher-level reasoning about commutativity of abstract operations, e.g., in [27])

3.1.3 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 [39], TM implementations have multiple alternatives for dealing with that situation (such as leaving the parent data-set unchanged, updating it in-place, dropping it altogether, and others), which may be confusing for the programmers using those implementations. We thus decide to disallow this behavior in TFA-ON: open sub-transactions may not update 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 run-time 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 [42] – regular programmers can then use those libraries to take advantage of open nesting benefits.

3.2 Transactional Forwarding Algorithm

TFA [53, 56] was proposed as an extension of the Transactional Locking 2 (TL2) algorithm [19] for DTM. It is a data-flow based, distributed transaction management algorithm that provides atomicity, consistency, and isolation properties for distributed transactions. TFA replaces the central clock of TL2 with independent clocks for each node and provides a means to reliably establish the "happens before" relationships between significant events. TFA uses optimistic concurrency control, buffering all operations in per-transaction read and write sets, and acquiring the object-level locks lazily at commit time. Objects are up-

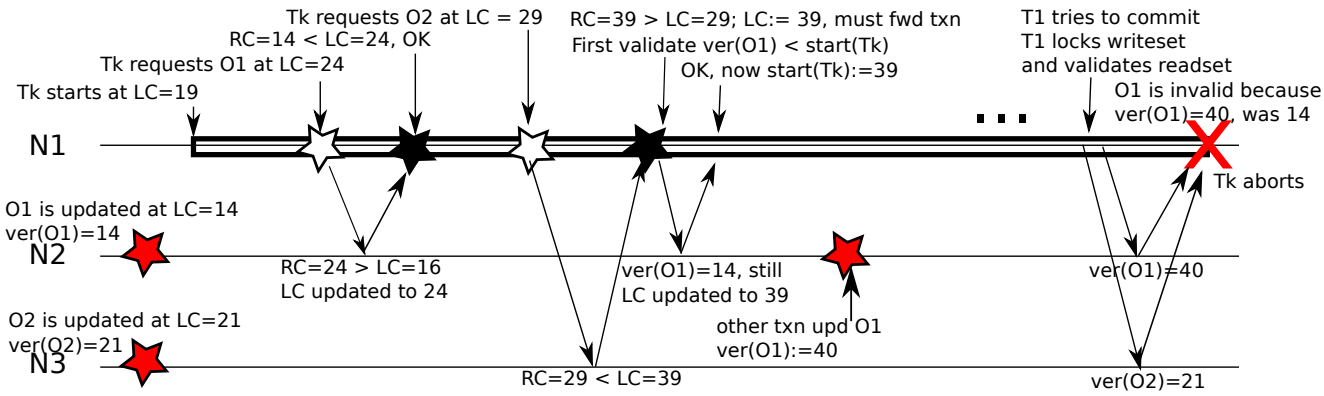


Figure 3.1: Transactional Forwarding Algorithm Example, from [62]

dated once all locks have been successfully acquired. Failure to acquire a lock aborts the transaction, releasing previously acquired locks and thus avoiding deadlocks.

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

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

We illustrate TFA with an example. In Figure 3.1, 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_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).

3.3 Hyflow DTM Framework

Two of our proposed algorithms were implemented and evaluated in Hyflow [52, 53], a DTM framework for Java. Hyflow's design attempts to be modular by allowing for pluggable support for lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols. Hyflow extends upon Deuce STM [32] and relies on automatic byte-code rewriting to provide an API based on annotations, without requiring compiler or JVM support. Hyflow along with its programming interface and its shortcomings will be described in Section 7.1.

Chapter 4

Closed Nesting

We extend TFA to support Closed Nesting and partial aborts. The resulting algorithm, Nested Transactional Forwarding Algorithm (N-TFA) was implemented in Hyflow and evaluated.

4.1 N-TFA Algorithm Description

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.

Starting from these assumptions, it is straightforward to implement the rules described in Section 3.1.1. 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 [56]. 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). We must note that two variations of N-TFA could be obtained based on whether merge commits are conditioned by a read-set validation or occur unconditionally. In what follows we will only refer to unconditional merge-commits, because

any extraneous validations proved in our experiments to have high overheads that decreased performance while bringing no benefits.

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 again 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 4.1). 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}

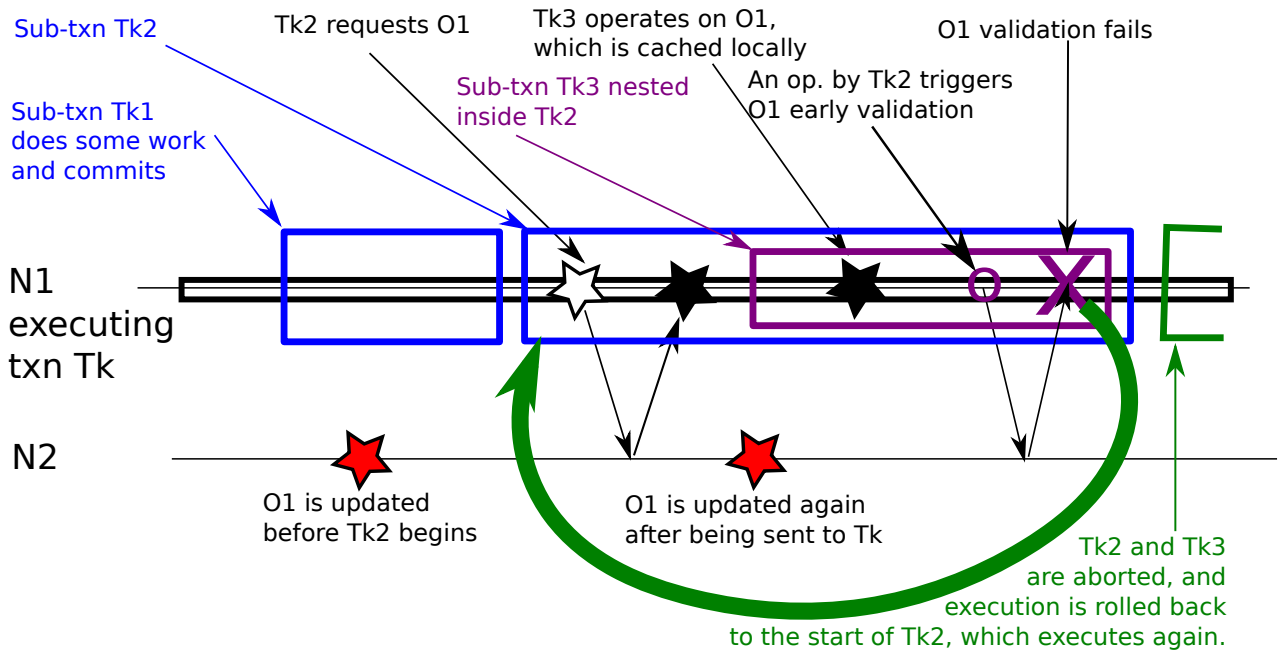


Figure 4.1: Nested Transactional Forwarding Algorithm Example

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.

4.2 Properties

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

Opacity [22] 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 4.2.1. *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 [56], 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 4.2.2. *N-TFA ensures strong progressiveness.*

Proof. This follows immediately from the proof that TFA is strongly progressive [56], 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.

□

4.3 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.

4.3.1 Implementation Details

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 was deemed unnecessary for our research purposes.)

4.3.2 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 [6, 11].

We targeted the effect of several parameters:

- Ratio of read-only transactions to total transactions (denoted in figure legends with

%).

- 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.

4.3.3 Experimental Results

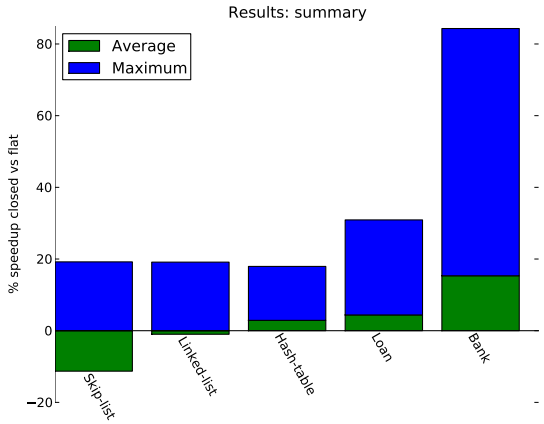


Figure 4.2: Performance change by benchmark.

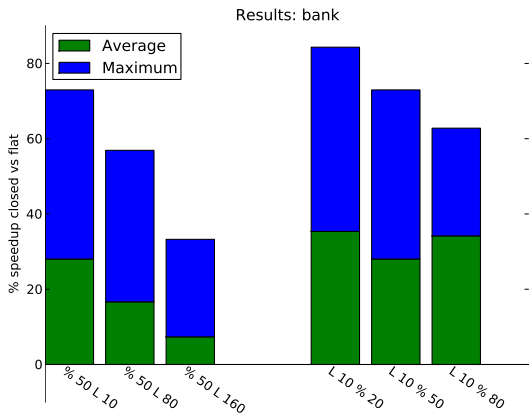


Figure 4.3: Bank monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions.

The results of our experiments are shown in Figures 4.2-4.10. Figure 4.2 shows a summary view of the improvement for each of our benchmarks. Figures 4.3-4.9 provide details on each of the benchmarks. Finally, Figure 4.10 looks at the scalability of N-TFA.

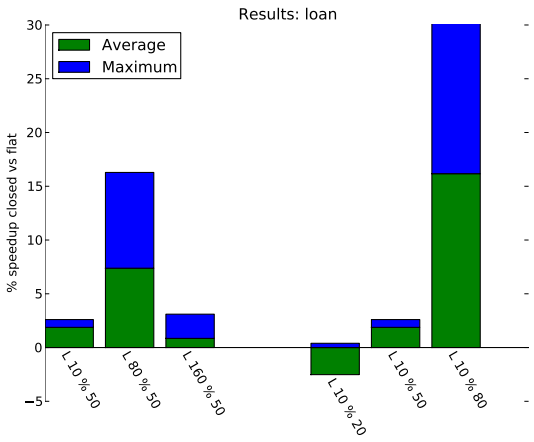


Figure 4.4: Loan monetary application. First group varies transaction length while keeping read-ratio constant. Second group varies the read-ratio for short transactions.

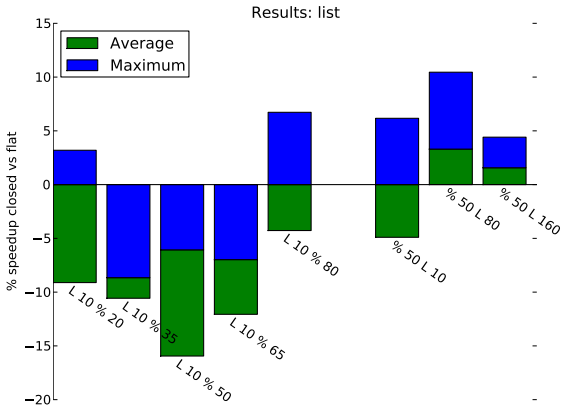


Figure 4.5: Linked-list micro-benchmark. First group varies read-ratio for short transactions. In the second group, transaction length is varied.

The performance of closed nesting varies significantly compared to flat nesting (see Figure 4.2). The single worst slowdown recorded was 42%, while the best speedup was 84%. Across all experiments, closed nesting proved to be on average 2% 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 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 4.6 groups 1 and 3) and Skip-List (Figure 4.8 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 4.4 group 2 and Hash-table with $c = 5, o = 7$ in Figure 4.7 group 3). Other cases showed a sweet spot in the middle of the range (Hash-table with $c = 3, o = 7$ in Figure 4.7 groups 2 and 3). Yet other cases show the opposite effects: performance negatively correlated with read-ratio on Skip-list (see Figure 4.9 groups 1 and 3), or worst performance in the middle of the range (Skip-list in Figure 4.9 group 2, Bank in Figure 4.3 group 2

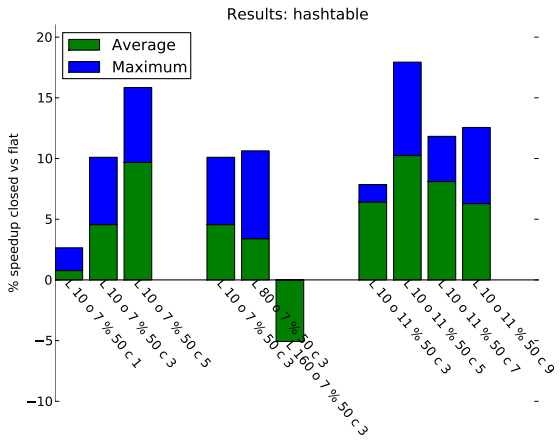


Figure 4.6: Hash-table micro-benchmark. First group shows increasing number of calls on hash-tables with 7 buckets. Second group shows the effect of increasing transaction length. Third group shows increasing number of calls on hash-tables with 11 buckets.

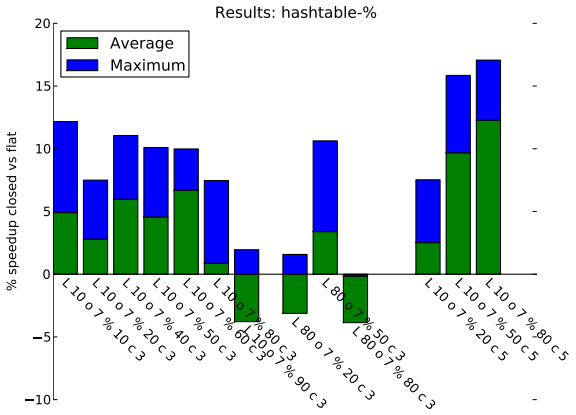


Figure 4.7: Hash-table read-ratio plot. First group shows the effect of increasing read-ratio on short transactions with 3 calls. Second group shows the effect of the same parameter on longer transactions. Third group targets transactions with 5 calls.

and, most obviously, Linked-list in Figure 4.5 group 1). Transaction length has a similar unpredictable influence: negative correlation on Bank (Figure 4.3 group 1) and Hash-table (Figure 4.6 group 2), middle range peak on Loan (Figure 4.4 group 1) and middle range dip on Skip-list (Figure 4.9 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 4.6 between groups 1 and 3).

From the experiment to evaluate closed nesting's scalability (Figure 4.10), 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 performance increases relative to flat nesting.

4.4 Conclusion

We presented N-TFA, an extension of the Transactional Forwarding Algorithm that implements closed nesting in a Distributed Software Transactional Memory system. N-TFA

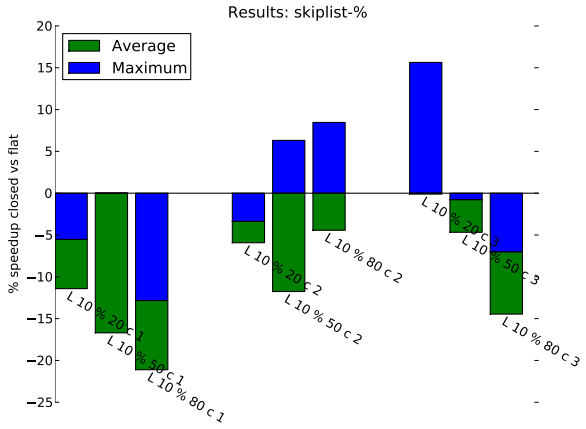


Figure 4.8: Skip-list micro-benchmark. Shows the effect of increasing read-ratio on tests with one, two and three operations performed in a transaction, respectively.

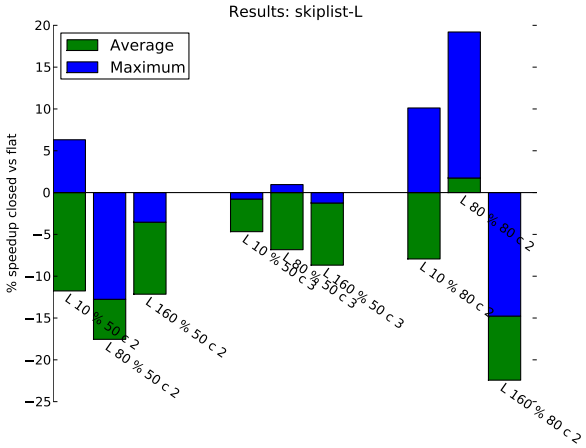


Figure 4.9: Skip-list micro-benchmark. Shows the effect of increasing transaction length. First group contains transactions with 2 calls. For the second group, the number of calls is 3. The last group has 80% reads.

guarantees opacity and strong progressiveness. We implemented N-TFA in the HyFlow DTM framework, thus providing, to the best of our knowledge, the first DTM 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 limited cases.

We determined that closed nesting applies better 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 somewhat better than TFA, although the performance dips at around 19 concurrent transactions per object.

Closed nesting however fails as a simple, general purpose method for increasing DTM performance. This is because in two out of five benchmarks, closed nesting is on average slower than flat transactions. In another two benchmarks the average speedup was less than 5%.

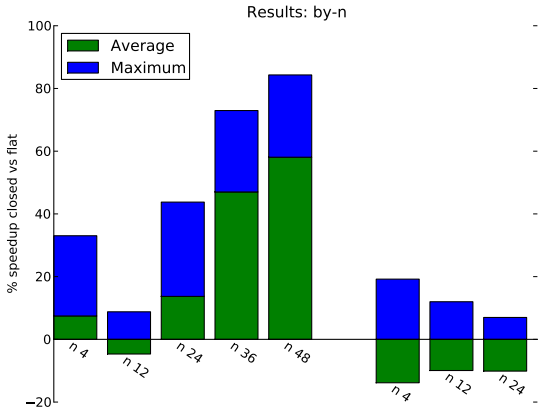


Figure 4.10: Effect of number of nodes participating in the experiment. First group shows the Bank benchmark with 16 threads/node. Second group shows the Skip-list micro-benchmark, with 4 threads/node.

Chapter 5

Open Nesting

We first describe the Transactional Forwarding Algorithm with Open Nesting (TFA-ON), TFA's extension to support open nesting. We then describe key details of its implementation in the HyFlow DTM framework. We proceed to evaluate it experimentally and analyze the results.

5.1 TFA with Open Nesting (TFA-ON)

We describe TFA-ON with respect to the TFA algorithm and N-TFA (Section 4.1), its closed-nesting extension. The low-level details of TFA were summarized in Section 3.2, 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 5.1.

When transactional forwarding is performed, all the read-sets up to the innermost open-

```
class Txn {

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

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

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

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

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

Figure 5.1: Simplified source code for supporting Open Nesting in TFA's main procedures.

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.

5.1.1 Abstract Locks

Additionally, TFA-ON has to deal with abstract lock management and the execution of commit and compensating actions (as explained in Section 3.1.2). 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).

5.1.2 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. Closed-nested ancestors simply pass these handlers to their own parents when they commit, but they have to execute the compensating actions in case they abort.

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

Figure 5.2 shows how a transaction would look within our system. Notice how the onAbort and onCommit handlers must request (open) the objects they operate on. They cannot rely on the copy opened by the original transaction, as 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();
```

Figure 5.2: Simplified transaction example for a BST insert operation. Code performing the actual insertion is not shown.

5.1.3 Transaction Context Stack

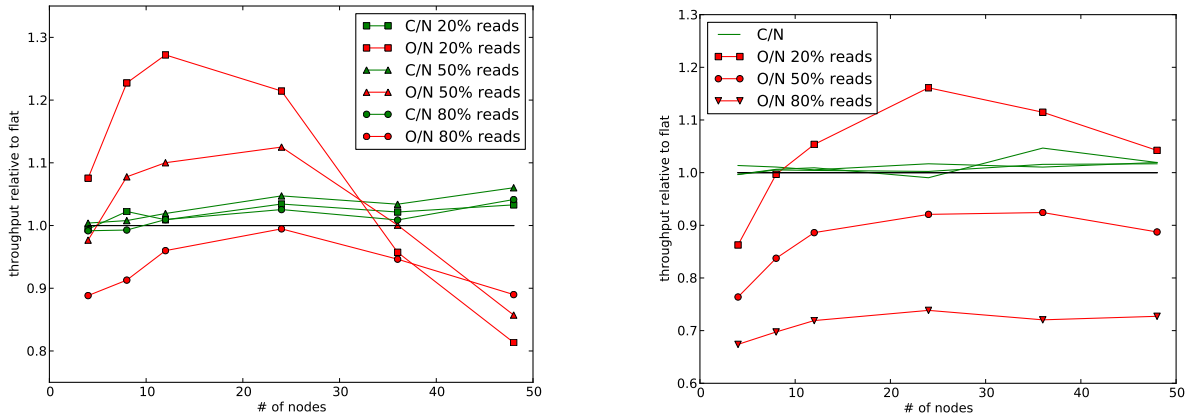
Meta-data for each transaction (such as read and write-sets, starting time, etc.) is stored in Transaction Context objects. While originally in HyFlow each thread had its own context object, in order to support nesting, we arrange the context objects in thread-local stacks. Each sub-transaction (closed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

5.2 Evaluation

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

5.2.1 Experimental Settings

The performance of TFA-ON was experimentally evaluated using four distributed micro-benchmarks including three distributed data structures (skip-list, hash-table, binary search



(a) Skip-list

(b) Hash-table

Figure 5.3: Performance relative to flat transactions, with $c = 3$ calls per transaction and varying read-only ratio. Both closed nesting and open nesting are included.

tree), and an enhanced counter application.

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

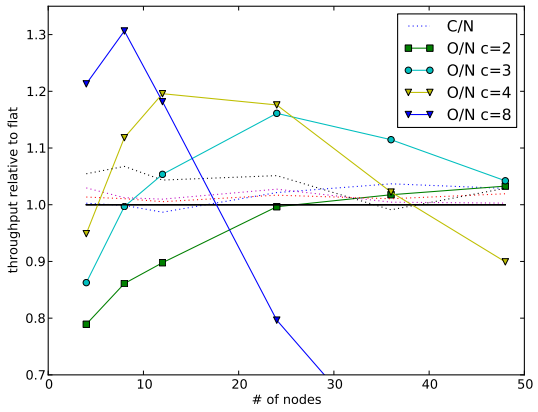
- Committed/aborted transactions.
- Committed/aborted sub-transactions (closed and open nesting).
- Committed/aborted compensating/commit actions (open nesting only).
- Waiting time after aborted (sub-)transactions (for exponential back-off).

Other data that we recorded includes:

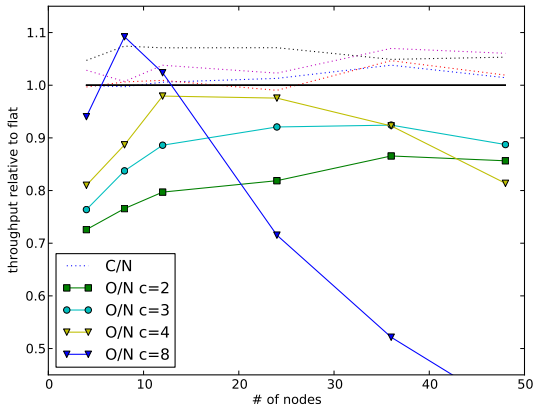
- Number of objects committed per (sub-)transaction.
- Which sub-transaction caused the parent transaction to abort.

Unfortunately, we cannot compare our results with any competitor DTM, as none of the two competitor DTM frameworks that we are aware of support open nesting [6, 11].

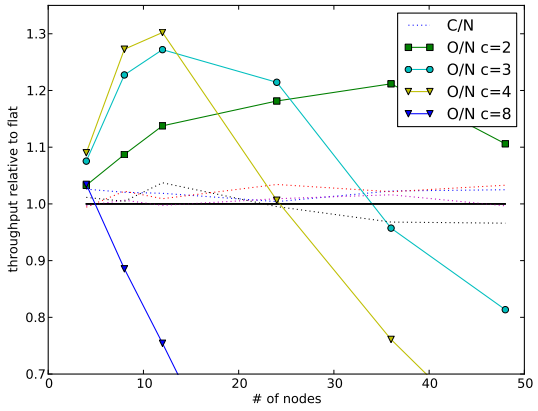
The skip-list, hash-table, and BST benchmarks instantiate three objects each, then perform a fixed number of random set operations on them using increasing number of nodes. Three important parameters characterize these benchmarks:



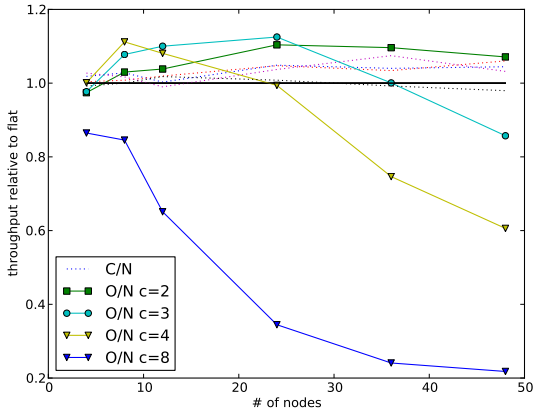
(a) Hash-table 20% reads



(b) Hash-table 50% reads



(c) Skip-list 20% reads



(d) Skip-list 50% reads

Figure 5.4: 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.

- 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 fundamental conflicts. Unless otherwise stated, we used $k = 100$.

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

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

5.2.2 Experimental Results

For all the data-structure micro-benchmarks, we observed that open nesting's best performance improvements occur at low read-only ratio workloads. Figure 5.3 shows how open nesting throughput climbs up to a maximum and then falls off faster than either flat or closed nesting as contention increases due to more nodes accessing the same objects. Figure 5.3 also shows the effect that read-only ratio has on the throughput. It is noticeable that on read-dominated workloads, open nesting actually degraded performance. Closed-nesting constantly stayed in the 0-10% improvement range throughout our experiments (closed nesting behavior is uninteresting and will henceforth be either omitted from the plots or shown without identification markers to reduce clutter).

Focusing on write-dominated workloads ($r = 20$ and $r = 50$), Figure 5.4 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 $c = 8$ [61]. On skip-list it is noticeable that at high contention ($c = 8$) the region of maximum benefit disappears and the performance decreases monotonously.

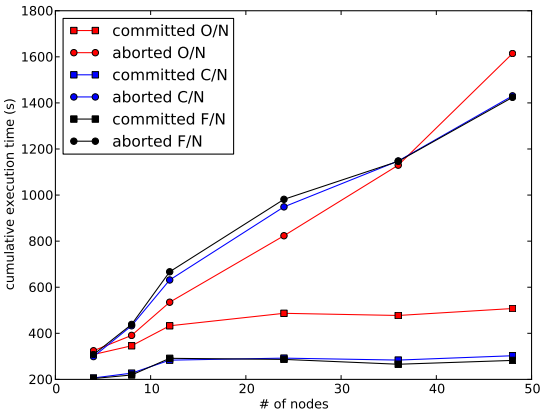
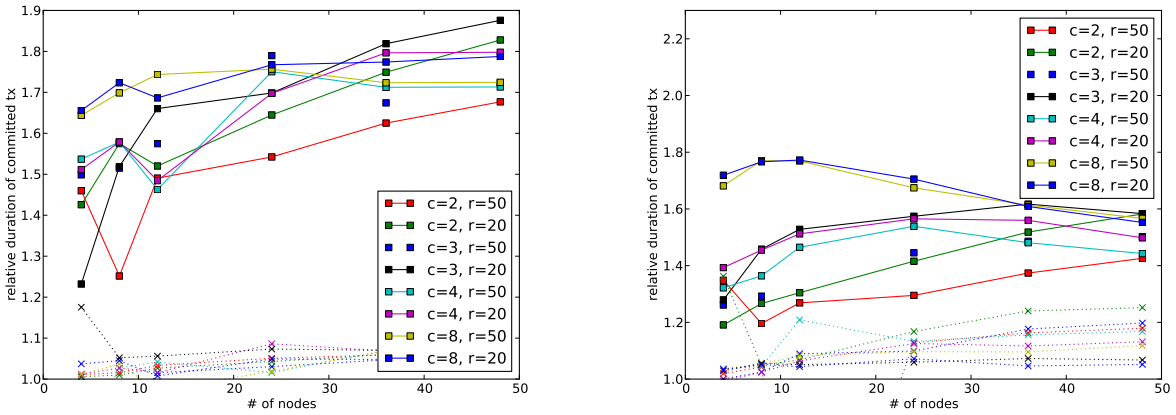


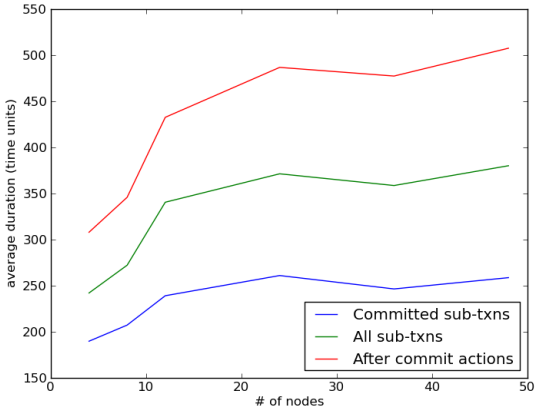
Figure 5.5: Time spent in committed vs. aborted transactions, on hash-table with $r = 20$ and $c = 4$. Lower lines (circle markers) represent time spent in committed transactions, while the upper lines (square markers) represent the total execution time. The difference between these lines is time spent in aborted transactions.



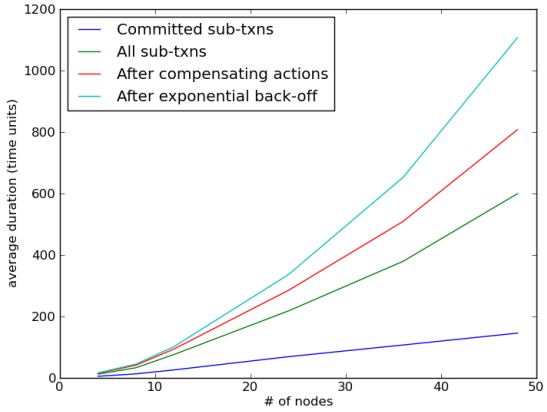
(a) Hash-table

(b) Skip-list

Figure 5.6: Overhead of successful open-nested transactions. Plotted is the relative ratio of the average time taken by successful open-nested transactions to the average time taken by successful flat transactions. Closed-nested transactions are also shown, with dotted markers and without identification.

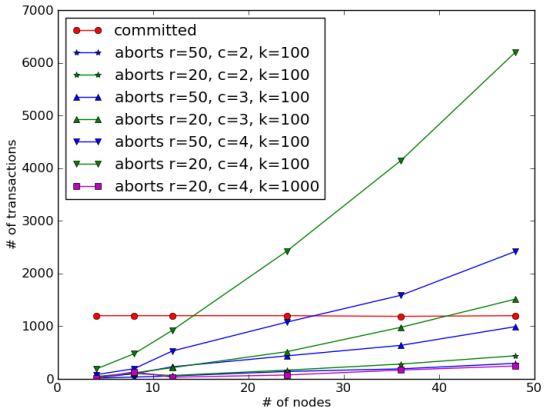


(a) Committed transactions

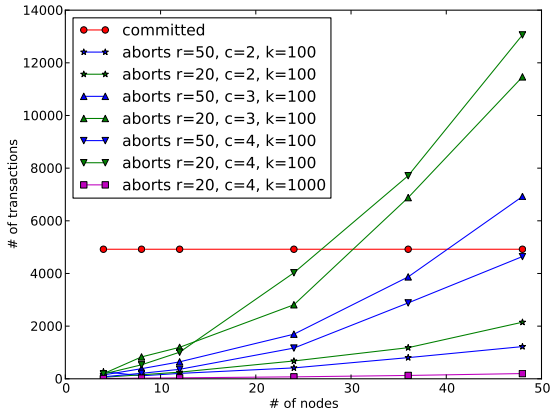


(b) Aborted transactions due to abstract lock acquisition failure

Figure 5.7: Breakdown of the duration of various components of a transaction under open nesting, on hash-table with $r = 20$ and $c = 4$.

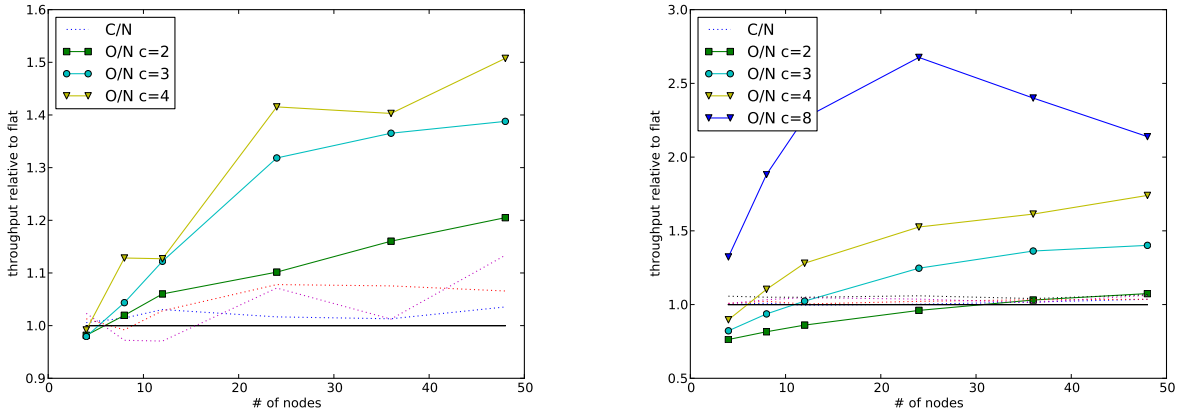


(a) Hash-table



(b) Skip-list

Figure 5.8: Number of aborted transactions under open nesting, with various parameters. The figure shows the effect of read-only ratio, number of calls, and key domain size. Note that all aborts depicted in this plot are full aborts due to abstract lock acquisition failure. The number of committed transactions is fixed for each experiment.



(a) Skip-list

(b) Hash-table for r=20

Figure 5.9: Throughput relative to flat nesting with increased key space $k = 1000$ and write-dominated workloads $r = 20$.

These observations can be explained by examining how is the time spent when using open nesting. Figure 5.5 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 5.6 shows that the average overheads of open nesting relative to flat transactions (50-80% on hash-table and 40-50% on skip-list) are significant and higher than that of closed nesting (3-7% on hash-table and 5-16% on skip-list). We observe the overheads are benchmark dependent, and are lower for workloads which access more objects in every sub-transaction. This is apparent when comparing Figures 5.6(b) and 5.6(a), and further experiments we have performed with higher nodal levels on skip-list [61] confirm our observation.

On the other hand, the time taken by aborted transactions in open nesting (Figure 5.5) 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 5.7(a) and 5.7(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 5.8. 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 fundamental 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 5.8. Increasing the number of calls leads to a rapid increase in the number of aborts. However, the key space k has a more pronounced effect. Setting $k = 1000$ reduced the frequency of fundamental conflicts and abstract lock contention. As a result, the number of aborts as compared to other configurations in Figure 5.8 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 5.9, we show throughput increase up to 51% on Skip-list (at $c = 4$ and $r = 20$) and up to 167% on Hash-table (at $c = 8$ and $r = 20$). Benefits for open nesting become possible even in non-write-dominated workloads: with $c = 3$ on skip-list, we have found 12% improvement at $r = 80$ and 21% improvement at $r = 50$ [61].

In our enhanced counter micro-benchmark we observed improvements consistent with our previous findings (plot in [61]). 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).

5.3 Conclusion

We presented TFA-ON, an extension of the Transactional Forwarding Algorithm that supports open nesting in a Distributed Transactional Memory system. We implemented TFA-ON in the HyFlow DTM framework, thus providing, to the best of our knowledge, the first DTM implementation to support open nesting. Our TFA-ON implementation enabled up to 30% speedup when compared to flat transactions, for write-dominated workloads and increased fundamental conflicts. Under reduced fundamental conflicts workloads, speedup was as high as 167%.

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

Chapter 6

Checkpoints

In this chapter we discuss transactional checkpoints in the context of Distributed Transactional Memory. We start by introducing checkpoints and discussing their motivation. We then examine continuations, the control-flow mechanism required for implementing checkpoints. Next we introduce TFA-CP, the TFA extension with support for checkpoints, and implement it in Hyflow2. Finally, we evaluate it experimentally.

6.1 Transaction Checkpointing Background

Using checkpoints as an alternative to nested transactions was proposed in the Transactional Memory community by Koskinen and Herlihy [33], and since then has largely been ignored.

Using nesting (in particular the closed nesting model), transactions may rollback to any of their ancestors' boundaries in order to resolve conflicts. However, once a sub-transaction commits, it becomes unavailable as a rollback destination, and instead its parent has to be restarted (unless the parent has also committed). This model is disadvantageous for workloads that have many sub-transactions nested at the same level, because this increases the amount of potentially valid work that needs to be aborted and uselessly re-executed in case of a conflict.

Checkpointing addresses this issue by allowing execution to return to any previously saved state (checkpoint) within the current transaction, regardless of whether the sub-transaction encompassing that checkpoint is still active or not. This allows developing a very fine grained rollback mechanism, which can identify the exact operation execution needs to rollback to in order to resolve the current conflict.

6.2 Continuations

In order to restart execution from arbitrary points in a program's control-flow graph, specialized mechanisms are needed. Nested transactions usually employ *exceptions* as a method to *pass control up*, but this restricts the potential destinations to boundaries of ancestor caller methods. *Continuations* are a more general mechanism that allow reverting the program execution to any previously saved state. Continuations work by saving and restoring the execution context of the current thread, i.e., processor registers and stack contents. By themselves, continuations do not affect program data (i.e., the heap).

Unfortunately, standard JVMs do not provide any support for continuations. To enjoy continuations in Java, one would have to either use a non-standard JVM (e.g., Avian JVM, DaVinci JVM with the *continuation* patch) or employ a byte-code rewriting library (e.g., JavaFlow, LightWolf).

We experimented with the DaVinci JVM and with the JavaFlow library. The former is faster, because continuations are implemented in native code. It requires the users to run a non-standard JVM, which is not easily available, and needs to be compiled from older source code.

The latter choice, JavaFlow, is able to run on stock JVM, but is slower because it implements the continuation mechanism in Java code. JavaFlow stores all local variables in a per-thread stack structure which replaces and emulates the regular call stack. This replacement stack is under the control of the library: it is backed-up when suspending a continuation and later restored when resuming it.

Both JavaFlow and DaVinci support resuming the same continuation multiple times – a feature that is essential for implementing transaction checkpoints. DaVinci however required a small modification to enable this feature, while JavaFlow proved unstable and difficult to use, especially in conjunction with Scala.

6.2.1 TFA with Checkpoints (TFA-CP)

TFA-CP is an extension of the TFA algorithm which supports checkpoints instead of nested transactions. Figures 6.1 and 6.2 show several key operations of TFA-CP. Of interest is the *startCheckpointedExec* routine which acts as an event-loop: it repeatedly passes execution to a user-supplied block of code, which is to be executed transactionally. The user-code, during its execution, calls DTM library functions, which are potential checkpoint locations. The system may use any of these calls to trigger recording a checkpoint. When it does, execution is passed back to the event-loop thus creating a new continuation. This continuation is stored alongside the current read and write-sets as the new checkpoint within the context of the current transaction. Finally, the execution is passed back to the user-code by resuming the previously created continuation.

```
class HaiTxnLevel(val parLevel: HaiTxnLevel, cont: Continuation) {
  //...
}
class HaiInTxn extends InTxn {
  // Wrapper for the transaction. Needs to be a Runnable to be started as
  // a continuation.
  class HaiRunCkpt[Z](block: InTxn => Z) extends Runnable {
    def run() {
      // Execute block
      val res = block(HaiInTxn.this)
      // Transaction finished, return from continuation to commit.
      CheckpointStatus.set(CheckpointSuccess(res))
      Continuation.suspend()
    }
  }

  // Main entry point for transaction. Takes a function (block)
  // as an argument. Returns the value returned by a successful
  // transactional execution of the given function.
  def startCheckpointedExec[Z](block: InTxn => Z): Z = {
    // This var stores the list of checkpoints.
    var level = new HaiTxnLevel(null, null)

    // Start executing the transaction
    var cont = Continuation.startWith(new HaiRunCkpt(block))
    // Here we returned from the first continuation

    while (true) {
      // Why did the continuation return?
      CheckpointStatus.get match {
        case res: CheckpointInterim =>
          // Transaction requests a new checkpoint, store it.
          level = new HaiTxnLevel(level, cont)
          level.mergeFrom(level.parLevel)
          // Continue
          cont = Continuation.continueWith(cont)
        case res: CheckpointFailure =>
          // Conflict detected. Transaction requests rollback.
          val restartRes = restartAtInvalidLevel(block)
          level = restartRes._1
          cont = restartRes._2
        case res: CheckpointSuccess =>
          // This transaction is over, attempt commit.
          if (tryCommit()) {
            // Success
            return res.result.asInstanceOf[Z]
          } else {
            // Commit failed, rollback to appropriate checkpoint.
            val restartRes = restartAtInvalidLevel(block)
            level = restartRes._1
            cont = restartRes._2
          }
      }
    }
  }
```

Figure 6.1: Simplified code for TFA-CP.

```
  // Rolls back transaction
  def restartAtInvalidLevel[Z](block: InTxn => Z):
          Tuple2[HaiTxnLevel, Continuation] = {
    // Find out how far back we need to abort
    var level = _currentLevel.root
    while (level.status == Txn.Active && level.activeChild != null) {
      level = level.activeChild
    }
    // Resume execution at that checkpoint
    var cont = level.cont
    level = new HaiTxnLevel(level.parLevel, cont)
    if (level.parLevel != null) {
      level.mergeFrom(level.parLevel)
    }
    if (cont != null) {
      // If we have a continuation, resume there
      cont = Continuation.continueWith(cont)
    } else {
      // Otherwise, re-start at the beginning
      cont = Continuation.startWith(new HaiRunCkpt(block))
    }
    return (level, cont)
  }

  // Performs transactional forwarding
  def forward(rclk: Long) {
    if (rclk > _currentLevel.root.startTime) {
      // Check for read-set validity
      if (rsValidate) {
        // Valid read-set, update Txn start time
        _currentLevel.root.startTime = rclk
      } else {
        // Invalid read-set, abort
        // Since we abort to the last valid level, it is OK to update starting time
        _currentLevel.root.startTime = rclk
        CheckpointStatus.set(CheckpointFailure())
        Continuation.suspend()
      }
    }
  }
}
// Checkpointing-enabled directory manager
class CpDirectory extends Directory {
  override def open[T](id: String): T = {
    // Record checkpoint checkpoint
    CheckpointStatus.set(CheckpointInterim())
    Continuation.suspend()
    // Continuation returned, open as usual
    return super.open(id)
  }
}
```

Figure 6.2: Simplified code for TFA-CP. (continued)

Except for recording a checkpoint as described above, there are two other occasions when execution is passed from the user-code to the event-loop: on transaction completion and on the detection of a conflict. In the first case, the event-loop is tasked to commence the commit operation, and upon success, the loop is terminated. In the second case, and also if the commit fails, the system determines which checkpoint should the transaction be reverted to, in order to resolve the conflict while aborting a minimal amount of work. The appropriate continuation is then resumed, passing control back to the user-code.

TFA-CP currently stores checkpoints before object retrieval operations. The amount of checkpointing can be configured by specifying the probability P that each object retrieval would record a checkpoint. $P = 100$ is the finest-grained strategy that does not store superfluous checkpoints, and always allows resuming execution at the exact operation that would resolve the current conflict. $P < 100\%$ can be used to reduce the total number of checkpoints recorded and thus, any overhead associated with capturing continuations.

Checkpoints are presently stored in a doubly-linked list, with new checkpoints inserted at the head of the list. Each checkpoint stores the complete read and write-sets of the current transaction at the time the checkpoint is taken. This makes read operations fast, at the cost of increased memory consumption.

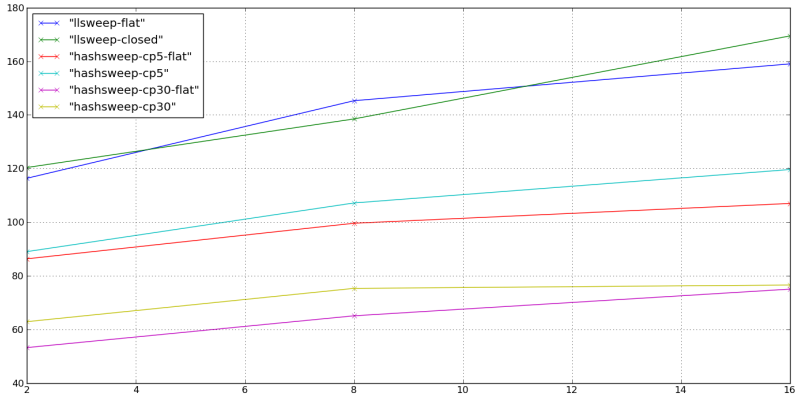
6.3 Evaluation

TFA-CP was implemented in Hyflow2, and evaluated on three micro-benchmarks and up to 16 nodes. We only present results for the continuations implemented using the DaVinci JVM. We summarily dismiss results using the JavaFlow library, due to the library's significantly inferior performance, incompatibility with certain Scala constructs, and extreme instability in tests.

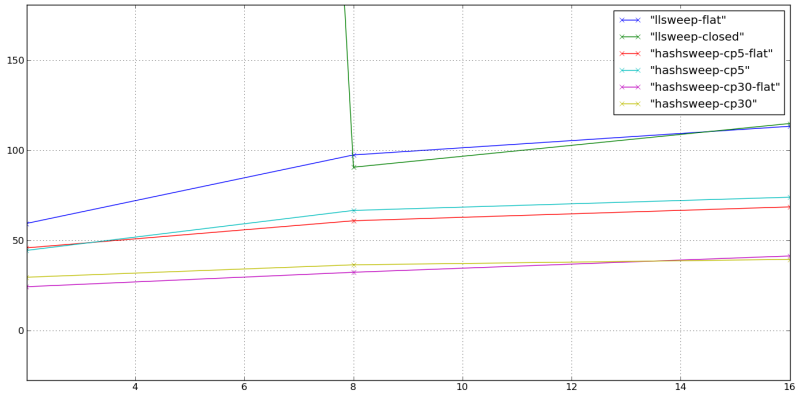
The results are presented in Figures 6.3 through 6.6. The X-axis represents the number of nodes, while the Y-axis shows the total throughput in transactions per second. Certain plots are missing data-points, or appear to have off-the-chart values. We believe these occurrences are caused by bugs either in our implementation or in the continuations back-end implementation. We will thus summarily dismiss such anomalous data-points.

The several data series in the plot reflect the following measurements:

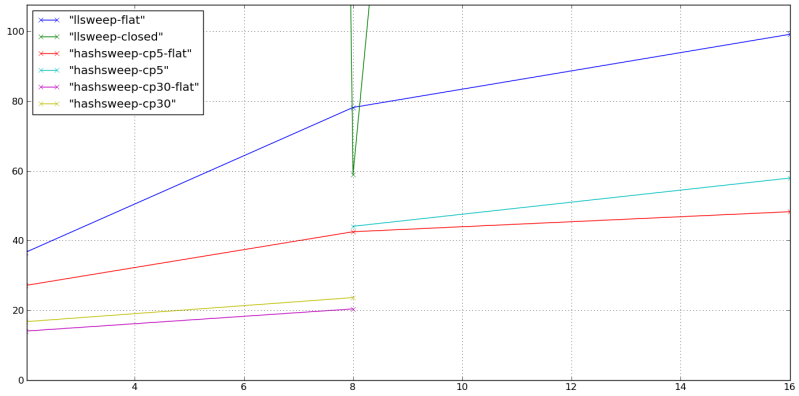
- Flat nesting - Regular TFA transactions without partial rollback. Marked in the legend as *flat*.
- Closed nesting - N-TFA transaction with partial rollback enabled by closed nesting. Marked in the legend as *closed*.
- Checkpointing - TFA-CP transactions with partial rollback enabled by transaction



(a) 3 ops

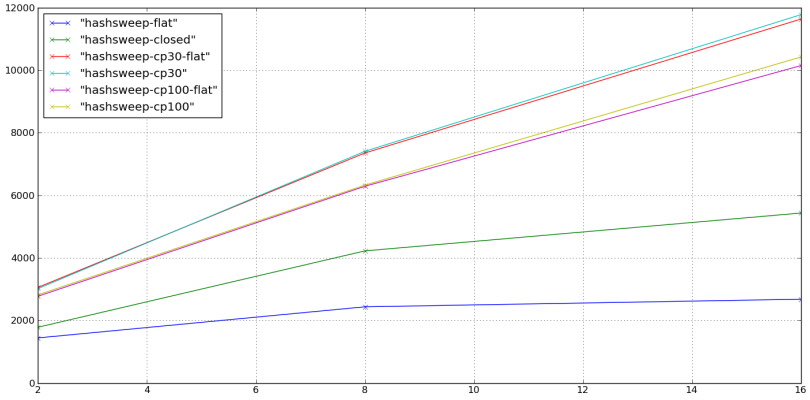


(b) 5 ops

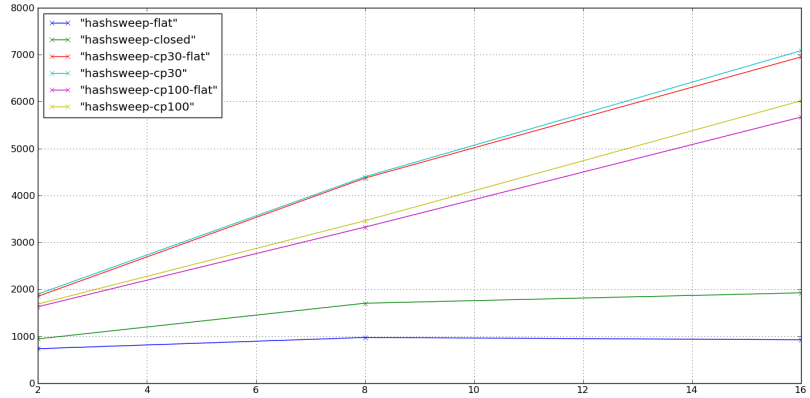


(c) 7 ops

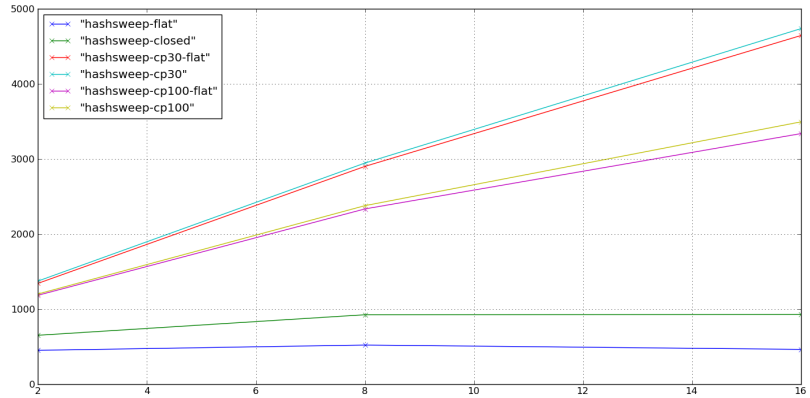
Figure 6.3: Results on the Linked-List benchmark.



(a) 3 ops

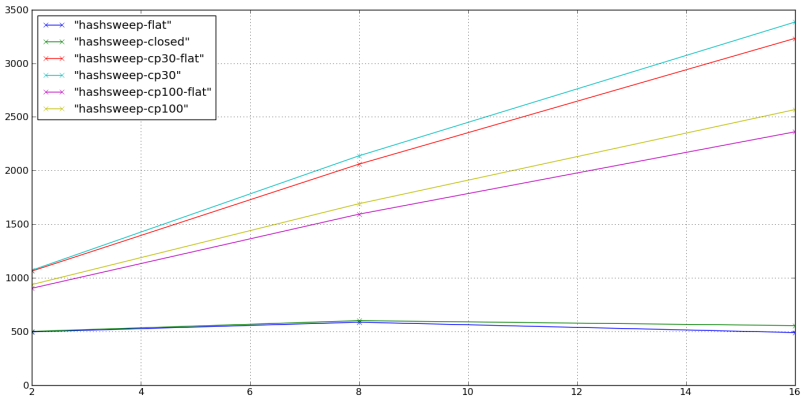


(b) 5 ops

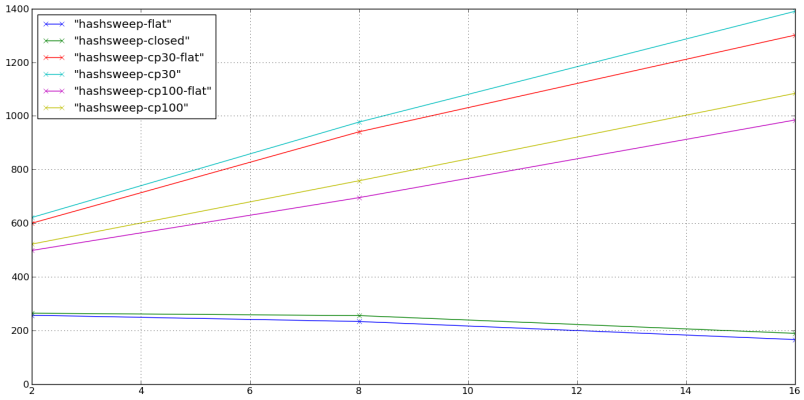


(c) 7 ops

Figure 6.4: Results on the Hash-table benchmark.

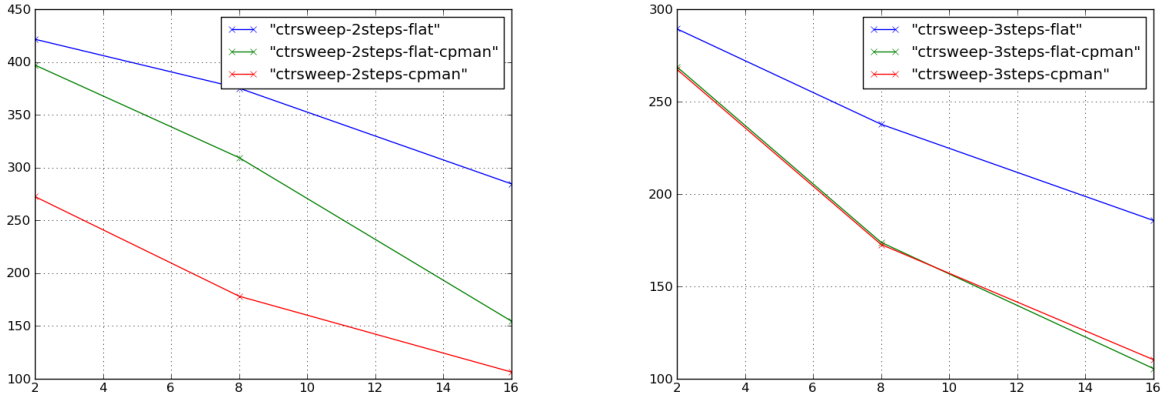


(a) 9 ops



(b) 15 ops

Figure 6.5: Results on the Hash-table benchmark (continued).



(a) 2 ops

(b) 3 ops

Figure 6.6: Results on the Enhanced Counter synthetic benchmark.

checkpoints. Marked in the legend as *cpX*, where *X* is the percentage of object opens that trigger a checkpoint to be captured.

- Flat Checkpointing - This measurement includes the overhead of capturing checkpoints, but disables partial rollback in order to factor out its performance effects. Marked in the legend as *cpX-flat*.

Figure 6.3 shows performance on the Linked-list benchmark. This benchmark is characterized by long sub-transactions that access many objects. Each transaction is comprised of 3-7 sub-transactions. The plots show the high overheads of continuations for this kind of workload (25-50%). It can be observed how the performance decreases when taking more checkpoints (i.e., 5% of object opens compared to 30%). The plots also show a 0-10% performance benefit due to partial rollback after factoring out the overheads, but this benefit is insufficient to offset the overheads.

Figures 6.4 and 6.5 show transaction throughput on the Hash-table benchmark. This benchmark has short sub-transactions that access only two objects, the hash-table meta-object and the inner bucket object, and is highly concurrent. Transactions are comprised of 3 to 15 sub-transactions. The results on this benchmark were unexpected. Simply enabling the continuations mechanism gives a significant 3-6x performance increase compared to flat transactions. We can not yet explain this behavior. Additionally enabling partial rollback gives a further 1-10% improvement, consistent with the previous benchmark. We further notice that the benefit due to partial rollback increases with the length of the transaction: for transactions comprised of 3 operations the benefit is negligible, while transactions with 15 operations see up to 10% performance increase.

Figure 6.6 shows the results on our final benchmark, an Enhanced Counter application. This

benchmark is comprised of a configurable number of stages. Each stage accesses a configurable subset of objects, from a dedicated pool of counters. After each stage, a checkpoint is manually triggered. The plots again show significant checkpointing overheads. It is unexpected however that enabling partial rollback does not improve performance, but rather degrades it.

6.4 Conclusions

Our experiments with TFA-CP determined that partial rollback with checkpointing generally shows a 1-10% improvement compared to flat transactions, if all the overheads are factored out. In two of our benchmarks the overheads were however too great and the net effect was a performance degradation. The third benchmark however experienced 3-6x performance improvement by solely enabling checkpoints (i.e., without partial rollback). We can not currently explain this behavior, but we suspect it is related to the amount of data stored in each checkpoint (i.e., the size of the stack backup). We leave a more in-depth analysis for future work.

Chapter 7

Hyflow2: A High-Performance DTM Framework in Scala

In this chapter we introduce Hyflow2, our high-performance DTM framework for the JVM, written in Scala. We start by looking at Hyflow and describing our reasons for choosing to rewrite it from scratch. We then present Hyflow2's programming interface, and finally, we evaluate its performance relative to its predecessor.

7.1 The Hyflow DTM framework. Motivation

Hyflow is a DTM framework written in Java. Its main purpose is to enable research and experimentation in the Distributed Transactional Memory area. Hyflow is built on top of the Deuce STM library and the Aleph communication framework, two research projects that are not actively maintained. Hyflow's design attempts to be modular by allowing for pluggable network transports, transactional algorithms, directory protocols and contention managers. The interfaces to the various components are however not flexible enough, and hard-coded links exist between some of the modules' implementations. Additionally, changes were made to the two underlying libraries in order to work around their limitations. All these reasons make Hyflow a difficult code base to work with and extend.

Hyflow (just like the underlying Deuce STM) relies on automatic byte-code rewriting to provide an API based on annotations. As seen in Figure 7.1, the user marks the methods to be executed transactionally as *@Atomic*. A Java Agent rewrites such methods into two polymorphic copies: the first copy has the same signature as the original method, and it initiates a new transaction (or reuses an already running transaction, if available) and then calls the second copy within the context of this transaction. The second copy is a transacted version of the original method's byte-code. It takes an additional argument (a transaction context), and replaces all field reads and writes with transactional read and write operations.

```
@Atomic
void transfer(Account a1, Account a2, int amount)
{       withdraw(a1, amount);
        deposit(a2, amount);
}
@Atomic
void withdraw(Account a, int amount) {
        a.value -= amount;
}
@Atomic
void deposit(Account a, int amount) {
        a.value += amount;
}
```

Figure 7.1: Example of the Hyflow API. Transactions are marked using the @Atomic annotation.

Any method calls within transacted code are modified to also pass the transaction context argument.

The automatic instrumentation also touches on methods not marked as *@Atomic*, by creating an additional transacted copy of the method as described above. When such method is called outside any transaction, the original byte-code is executed. When methods are called within a transaction (by transacted code), the addition of the transaction context argument leads to executing the transacted versions of the methods.

This approach works particularly well for a simple multiprocessor transactional memory system because the instrumented byte-code can be made very fast: no extra objects need to be instantiated (the transactional context object can be reused), method calls can be kept to a minimum (the transactional read and write operations can be inlined), and only one thread-local variable lookup needs to be performed at the beginning of the transaction. However, this model is particularly poor for rapid prototyping, essential for researchers, because of the low-level nature of byte-code instrumentation. Moreover, the potential speed benefits of this model become negligible when dealing with distributed systems, where network accesses are the most costly operations. Modern JVMs with state-of-the-art Just-in-Time (JIT) compilation and garbage collection further minimize the benefits of the byte-code rewriting approach.

Being dissatisfied with Hyflow, we decided to design and implement a better DTM framework, Hyflow2. Our aims for Hyflow2 are as follows:

- High-performance. In order for DTM to have any traction, it needs to be at least similar in speed with the existing systems it aims to replace. Thus, performance is paramount.
- Rapid prototyping. We want our framework to be accessible for DTM researchers, in order to encourage progress in this exciting field. As a side-note, our chosen language

```
val ctr = Ref(0)
atomic { implicit txn ⇒
  ctr() = ctr() + 1
}
```

Figure 7.2: An example transaction in ScalaSTM (common usage).

```
val ctr: Ref[Int] = Ref[Int](0)
atomic.apply(new Function1[InTxn,Unit] {
  def apply(implicit txn: InTxn): Unit = {
    ctr.update(ctr.apply(txn) + 1)(txn)
  }
})
```

Figure 7.3: A more verbose version of the code in Figure 7.2, with several Scala syntactic shortcuts written explicitly.

for implementing Hyflow2, Scala, is excellent for the purpose of rapid-prototyping.

- Easy to use. Besides being a vehicle for DTM research, we also want Hyflow2 to be used by regular programmers. Thus, an easy, clean and familiar API is desirable.

7.2 Hyflow2 API

Hyflow2 API is based on the excellent ScalaSTM API. In fact, Hyflow2 tries to reuse ScalaSTM's interfaces wherever possible, and partially implements a back-end for the ScalaSTM API.

7.2.1 ScalaSTM

ScalaSTM is an STM API for Scala under consideration to be included in the Scala standard library in an upcoming release. The API allows for pluggable back-end implementations, and it ships with a reference implementation, CCSTM[8]. Hyflow2 inherits all features described in this section.

Transactions in ScalaSTM are defined using *atomic blocks*, as shown in Figure 7.2. To enable programming using this syntax, *atomic* is a *TxnExecutor* object whose *apply* method takes a function as its only argument and executes this function as a transaction. The "implicit txn =>" construct denotes that the function passed to *apply* takes one implicit argument, the transaction context object.

ScalaSTM uses *transactional references* (*Ref*s) as a container for the values that are to be accessed using transactional semantics. The Ref containers mediate all access to the data

```
def takeFirst(): T = atomic {
  implicit txn =>
    val old_head = this.head()
    if (old_head == null)
      retry // do not proceed if empty
    this.head() = old_head.next
    return old_head.value
}
```

Figure 7.4: Conditional synchronization using retry. Transaction can only proceed once there is at least one item in the list.

within. To access a value of a Ref *ref1* within a transaction, one would use *ref1()* – i.e., call ref1.apply() – or ref1.get() as an alternative syntax. To change the value of the Ref inside a transaction, one should use *ref1() = v* – i.e., call ref1.update(v) – or alternatively, *ref1.set(v)*.

All of these methods (apply, get, update and set in class Ref) take a transaction context object (i.e., an instance of the class *InTxn*) as an additional, implicit argument. Implicit arguments in Scala code may be omitted, as long as the compiler can find in scope a variable of the appropriate type marked with the *implicit* keyword. In Figure 7.2, the *txn* object is automatically passed to the apply() and update() methods. Figure 7.3 shows how Scala interprets the code in Figure 7.2.

This mechanism using implicit arguments and Refs leads to a clean syntax with relatively little redundant code (only the "implicit txn =>" construct and the function call "()" characters are superfluous). Another benefit of this mechanism is *strong atomicity* for all Refs. Strong atomicity is the desirable property of a TM system which protects against concurrent access of a memory location from both transactional code and non-transactional code (for contrast, a *weakly atomic* TM system would have an undefined behavior in this situation). Accesses to a Ref's contents via the *apply* or *update* methods require an implicit transaction context object to be in scope, otherwise compilation fails. This requirement is satisfied inside an atomic block as explained in the previous paragraph. Outside atomic blocks however, no transaction context value is implicitly available, so calls to *apply* or *update* would lead to compilation errors. Single-operation transactions are used to allow accessing Refs outside atomic blocks. *ref1.single.get()* would, for example, spawn a transaction for the sole purpose of retrieving ref1's value.

ScalaSTM allows temporarily aborting a transaction using the *retry()* method. This is usually used for enforcing preconditions. Suppose for example the *takeFirst* operation on a queue (Figure 7.4). When the queue is empty, this operation may invoke retry, effectively blocking until at least one element is available. This behavior is called *conditional synchronization*. After calling *retry*, the transaction should only execute again once any of the values it has read is updated, otherwise it will follow the same execution path and call *retry* again. A simplistic implementation may, however, blindly restart the transaction after an exponential back-off.

```
class Account(val _id: String) extends AObj {
  val type = field("")  // a string field
  val value = field(0)  // an integer field
  Hyflow2.dir.register(this) // Register with the directory manager
}
```

Figure 7.5: Hyflow2 Object example for a bank account.

```
def deposit(accId: String, amount: Int) = atomic {
  implicit txn =>
    val acc = Hyflow2.dir.open[Account](accId)
    val newVal = acc.value() + amount
    acc.value() = newVal
    returm newVal
}
```

Figure 7.6: Hyflow2 transaction example. Transaction must open an object before operating on it.

7.2.2 Hyflow2 Objects

While in ScalaSTM transactions operate on Refs directly, Hyflow2 introduces an additional layer – the Hyflow2 Object – as a container for Refs (see Figure 7.5). This layer is needed in order to solve the data distribution problem. On a single node objects can be referred to using a JVM reference, but for multiple nodes, this extra mechanism is required for identifying objects.

An Hyflow2 Object mixes in the AObj Scala trait [1] and is Hyflow2's basic unit of data. Each Hyflow2 Object (henceforth referred to as AObj) has a unique identifier, which Hyflow2 uses to locate the object. This key is usually specified by the user at the object's creation, by passing it as an argument to the constructor.

Each AObj is composed from one or more fields. Fields are specialized Refs that maintain their association with the enclosing AObj and their order number within that object. Fields are created by calling the *AObj.field* method inside the object's constructor, and passing it an initial value.

7.2.3 Hyflow2 Directory Manager

The Directory Manager (DM) is Hyflow2's module that keeps track of the objects' location. When an AObj instance is created, it registers itself with the DM (Figure 7.5). If the object later migrates to a different node, it updates its registration with the DM.

The Directory Manager also handles retrieving objects from their owner nodes over the

[1] A Scala trait is similar to a Java interface. A class can therefore mix in (i.e., implement) multiple traits. However unlike interfaces, Scala traits may contain implementation.

```
// Simple open-nested transaction without abstract locks or commit or abort handlers
atomic.open { implicit txn =>
  val ctr = Hyflow2.dir.open[Counter]("id")
  ctr.value() += 1
}
// Open-nested transaction that acquires a single abstract lock
atomic.open("abslock0") { implicit txn =>
  val ctr = Hyflow2.dir.open[Counter]("id")
  ctr.value() += 1
}
// More complex usage case, with abort and commit handlers. Lock is held after commit.
atomic.open { implicit txn =>
  acquireAbsLock("absLock0")
  val ctr = Hyflow2.dir.open[Counter]("id")
  ctr.value() += 1
} onAbort { implicit txn =>
  val ctr = Hyflow2.dir.open[Counter]("id")
  ctr.value() -= 1
} onCommit { implicit txn =>
  holdAbsLock("absLock0")
}
```

Figure 7.7: Open nesting in Hyflow2

network. This operation is called *opening* (see Figure 7.6). It requires the identifier of the requested object and it generally caches a copy of the requested object on the local node.

7.3 Transaction Nesting

Hyflow2 includes support for nested atomic blocks. In this section we first briefly describe the three nesting models previously studied in TM [26, 40]: flat, closed and open. Next we introduce the API support for nesting in Hyflow2, and explain how it works.

7.3.1 Nesting API

Flat and closed nesting are semantically equivalent and can be used interchangeably. Unlike Hyflow, we decided not to expose the decision of which of the two models to use in the regular user-facing API. Hyflow2 may use any of these models to handle nested atomic blocks. Currently, the decision is fixed based on a configuration value, but in the future it could be made adaptively at runtime.

Open nesting on the other hand requires API support. Following the style of ScalaSTM, in Hyflow2 we propose the following syntax (see Figure 7.7):

- An open nested transaction should be started with *atomic.open* . The body of the transaction follows in braces, just like for regular transactions.

```
new OpenNestingBlock(
  atomic.open { implicit txn =>
    // Atomic bloc is wrapped in an OpenNestingBlock
  }
).onCommit( { implicit txn =>
    // handler is passed to onCommit method. After
    // registering the callback, onCommit executes the block wrapped above.
  }
)
```

Figure 7.8: Expanded code showing mechanism for defining commit/abort handlers.

- Following the transaction's body two optional blocks may be specified. These blocks are introduced by *onCommit* and *onAbort*, and represent the transaction's commit and abort handlers, respectively. The handlers themselves are executed as open-nested transactions, so they must accept the implicit transaction context argument. If both handlers are present, their order is not important.
- If an open-nested transaction requires the acquisition of an single abstract lock which is known in advance, the lock's identifier can be passed as a string argument to *atomic.open* . The lock will be acquired before the open-nested transaction can commit, and will be released automatically as part of the transaction's abort and commit handlers. These handlers do not need to be present in the code, the lock will be released anyway (see Figure 7.7).
- For any other abstract lock scenarios, the locks must be acquired within the sub-transaction's body using *acquireAbsLock*. These locks too will be automatically released as part of the sub-transaction's abort and commit handlers.
- If for any reasons an abstract lock should be kept beyond the sub-transaction's commit or abort, *holdAbsLock* must be called in the commit and/or abort handler. Any such lock will be propagated to the innermost open-nested ancestor transaction and will be released upon its commit or abort.

7.3.2 Discussion and Language Mechanisms

We consider *atomic.open* a semantically cleaner way of denoting open-nesting transactions than the previously suggested *openatomic* keyword [42]. Our syntax logically breaks down into two terms. The first term, *atomic* is the same as the marker for regular atomic blocks. The second term, *open*, appears as a property of the resulting transaction. By contrast, *openatomic* as a separate keyword, gives the impression the effect is totally unrelated with that of the *atomic* keyword.

When evaluating an *atomic.open* block, the *open* method is called on the *atomic* object of type *TxnExecutor*, and it receives the function to be executed transactionally as a parameter.

```
STM.atomic(new Runnable {
  public void run() {
    Counter ctr = Hyflow2.dir().<Counter>open("ctr")
    ctr.set(ctr.get() + 1);
} } );
```

Figure 7.9: ScalaSTM Java compatibility API.

```
new Atomic<Boolean> {
  public Boolean atomically(InTxn txn) {
    Counter ctr = Hyflow2.dir().<Counter>open("ctr");
    ctr.value.set(ctr.value.get() + 1);
    return true;
  }
  public void onCommit(InTxn txn) {
    // Commit handler, omit if not needed
  }
  public void onAbort(InTxn txn) {
    // Abort handler, omit if not needed
  }
}.execute();
```

Figure 7.10: Hyflow2 Java compatibility API using the Atomic class.

Declaring the *onCommit* and *onAbort* handlers is more complex: blocks are evaluated last to first, wrapping what is above in a special *OpenNestingBlock* container object, and calling *onCommit*/*onAbort* on this object. The object is saved in a thread-local variable. When finally, *atomic.open* is invoked, it checks if there is any *OpenNesingBlock* object registered for the current thread and uses it, if any. See Figure 7.8 for an expanded example. This mechanism is also used in ScalaSTM to implement the *orElse* keyword (*orElse* provides the means to execute alternative atomic blocks if the original ones fail).

7.4 Java Compatibility API

Scala provides excellent interoperability with Java. As a result, many of the operations described above will just work when invoked from Java code either directly, or in a slightly different form (for example, methods *ref1.get*, *ref1.set*, *Hyflow2.dir.open*, *retry* becomes *Txn.retry*, etc.). Several of the more advanced Scala features that we use in the Hyflow2 API are however not supported from Java code, so we need to provide additional mechanisms to obtain the same results.

7.4.1 Defining Transactions

ScalaSTM already provides a way for starting transactions from Java which uses the Callable and Runnable interfaces for defining the transaction's body (Figure 7.9). The transaction

```
public class Counter extends jAObj {
  Ref<Integer>  value = field(0);
  public Counter() {
    Hyflow2.dir().register(this);
  }

  // This method is an example transaction. It is not part of the Hyflow2 Object definition.
  public static void increment(final String id) {
    new Atomic {
      public void atomically(InTxn txn) {
        Counter ctr = Hyflow2.dir().<Counter>open(id);
        // The first way of accessing Refs works only from an Atomic class
        // due to the txn parameter
        ctr.set(ctr.get(txn) + 1, txn);
        // The second way of accessing Refs also works using a Runnable
        ctr.single.set(ctr.single.get() + 1);
      }
    }.execute();
} }
```

Figure 7.11: Scala-style Hyflow2 Object definition in Java. Notice how accessing Refs in this style is more verbose.

context argument isn't used anymore – instead, all transactional operations need to dynamically determine the context object at run-time. If no transaction exists for the current thread, a single-operation transaction is created automatically. This mechanism, however, does not define the abort and commit handlers required for open-nesting.

To support open-nesting, Hyflow2 provides an Atomic abstract class with three methods: *atomically*, *onCommit* and *onAbort*. User code must subclass it and provide at least the implementation for *atomically* (see Figure 7.10). If implementations are provided for the other two methods, they will be used as commit and abort handlers. Unlike ScalaSTM's Java API, a transactional context object is passed to the transaction as an argument. Our reasons for doing so will become clear in Section 7.4.2.

7.4.2 Defining Hyflow2 Objects

Inheriting from a Scala trait in Java code is non-trivial. To allow a simpler way of defining Hyflow2 Objects in the Java API, we provide an abstract class called *jAObj*, which users must subclass.

Fields may be declared in two ways, which we named the Scala and the Java styles. This decision influences how the fields are later accessed from both Scala and Java code. The Scala way of declaring fields was already described in Section 7.2.2, and only differs cosmetically (see Figure 7.11). However, choosing to declare fields the Scala way makes Java code accessing that field more verbose: either the transaction context object needs to be passed explicitly to each *Ref.get / Ref.set* call (this object is available by sub-classing the *Atomic* abstract class as mentioned in Section 7.4.1), or *Ref Views* must be used to determine the

```
public class Counter extends jAObj {
  Ref.View<Integer>  value = jfield(0);
  public Counter() {
    Hyflow2.dir().register(this);
  }
  // Example transaction
  public static void increment(final String id) {
    STM.atomic(new Runnable {
      public void run() {
        Counter ctr = Hyflow2.dir().<Counter>open(id);
        ctr.set(ctr.get() + 1);
    } } );
} }
```

Figure 7.12: Java-style Hyflow2 Object definition in Java. Compact Ref access.

context at run-time by calling *Ref.single.get* or *Ref.single.set* instead of simply *Ref.get* or *Ref.set*. The Scala style of declaring Refs is thus recommended when the application mostly in Scala.

For applications written mostly in Java (or even Java-only), the Java style of declaring fields makes Java code more compact. Fields are declared using *jfield* instead of *field* and their type becomes *Ref.View* instead of *Ref* (see Figure 7.12). Java code can now access the fields using the shorter *ref1.get()*, etc. Note that the actual method invoked is now *Ref.View.get()* and determines the transaction context object dynamically at run-time. When using the Java style, the Scala compiler will not complain if a Ref.View is accessed outside an atomic block. Instead, it would fire a single-operation transaction.

7.5 Mechanisms and Implementation

Our implementation uses the actor model using Akka.

7.5.1 Actors and Futures

Akka is a very efficient actor model implementation for the JVM. The actor model can lead to very fast implementations because it reduces the need for thread context switching. Actor libraries generally do their own user-space scheduling, as opposed to relying on the OS scheduler, and prohibit blocking function calls (such as disk access. etc). Instead, actors send messages to each other and respond to the messages they receive – it is an event-based programming model.

An important part of Akka's interface are *Futures*. Futures represent the result of a computation that is expected to complete at some later time. Futures can be used when a thread sends a request to an actor and expects a response. Instead of waiting for the response to

arrive, the method sending the request immediately returns a Future object. The thread can register a callback to be executed when the response is received, query the Future periodically, or even block for the result. Computations can also be composed by chaining or aggregating Futures, thus reducing the number of times a thread needs to block and improving performance. Futures, as well as actors, receive and process messages and events using a configurable thread-pool.

7.5.2 Network Layer

Akka actors provide network transparency. They can seamlessly communicate across JVM and machine boundaries. Actor instances are identified using *ActorRef* objects. *ActorRef*s can be sent across the network while still maintaining their association with the correct actor. *ActorRef*s can then be used on the remote machine to communicate to the original actor. In conjunction with Futures, this makes it easy for developers to handle communication in an efficient manner.

Internally, Akka uses Netty for communicating over the network. Netty is a fast, asynchronous event-driven network application framework. It uses the non-blocking, high performance Java New I/O API. Netty also uses a configurable thread-pool for servicing received messages.

7.5.3 Serialization

Serialization is the process of converting an object to a format that can be sent through the network, and back. Traditionally, Java objects must implement a *Serializable* interface in order to enable this functionality. The standard Java serializer however is notorious for its poor performance. Fortunately, Akka provides an API for custom serializers, so we implemented an adapter for the Kryo library[2]. Kryo is one of the fastest JVM serialization frameworks, and is compatible with Scala.

7.5.4 Hyflow2 Architecture

Hyflow2 has a modular architecture. Depending on their function, module implementations need to comply to certain interfaces. Hyflow2 currently provides the following interfaces: lock service, object store, object directory, barrier service and cluster manager. A module implementation consists of a singleton object that complies to one of these interfaces and is used for sending requests to the module and an actor which services such requests. Modules communicate between each other and with the transactions' threads using message passing and Futures.

[2]http://code.google.com/p/kryo/

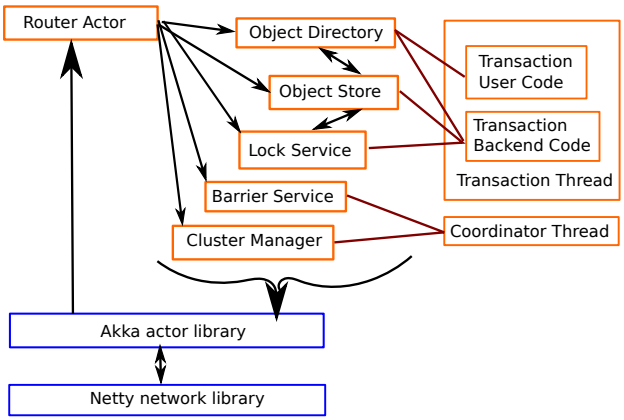


Figure 7.13: Hyflow2 system diagram

The lock service module handles acquiring, releasing and verifying the status of object and/or field locks. The object store module holds the objects themselves and handles queries, updates and validations (version checks). Due to their tight coupling, the lock service and object store can be combined in a single module. The object directory tracks object locations: it handles queries, updates, and it can also send notifications to interested transactions when an object is updated. The cluster manager tracks which nodes participate in Hyflow2 transactions, and is currently implemented by delegating a coordinator node (in the future, gossip protocols could be implemented). The barrier service lets multiple nodes coordinate their execution and is used mostly for benchmarking. An additional module is tasked with gathering statistics from all participating nodes. Figure 7.13 shows a system diagram including Hyflow2 modules and their interactions with the user-code and underlying libraries.

Each node has a router actor which serves as a gateway for all request messages (response messages do not pass through the gateway). The router actor dispatches messages to the appropriate module based on the message's type (Java class). This design allows every message to contain additional payload data, which can be processed in a consistent way. For example, the Transactional Forwarding Algorithm (TFA) which Hyflow2 implements needs to attach an integer (the node-local clock value) to each message sent over the network [56]. Instead of requiring every module to attach payloads to all the messages they send and receive, payloads are handled automatically in the message's base class constructor on the sender node, and is processed on the receiver node by the router actor.

7.5.5 Conditional Synchronization

Hyflow2 is the first DTM implementation to support distributed conditional synchronization. This feature was implemented by maintaining a waiting list of transactions which are blocked on each object. When they execute, transactions record all objects they access in

the transaction's read-set. When a transaction calls *retry*, it adds itself to the waiting lists of all objects which it has previously read, then blocks. Waiting lists are maintained by the Object Directory. When an object is updated, the Directory is notified, and in turn notifies all transactions on that object's waiting list. Because the message adding a transaction to an object's waiting list may arrive after the object is updated, the object version is checked as well: if the transaction is waiting on an old version of the object, the notification is sent right away. Otherwise, a transaction could be waiting unnecessarily for a condition that is already satisfied.

7.5.6 Parallel Object Open

This is another feature provided by Hyflow2 that can speed up certain transactions. Since objects are usually retrieved from remote nodes, the open operation is time-consuming. When a transaction needs multiple objects and knows their identity in advance, it can use the parallel object open operation to reduce the number of network round-trips required for acquiring a copy for each required object.

7.5.7 Performance

Thread context switches and network round-trip time are important bottlenecks. The choice of libraries we used in Hyflow2 was made with the purpose of addressing these issues. Akka and Netty are event-driven libraries and attempt to minimize thread context switches. We configured their internal thread pools to a minimum size that produces the greatest performance. Also, we specifically targeted serialization in our quest for performance because it lays on the critical path of sending a message over the network.

7.6 Experimental Evaluation

Hyflow2 was evaluated experimentally using a suite of one pseudo-macro-benchmark (bank monetary application) and four micro-benchmarks (counter and the skip-list, linked-list and hash-table data structures). Since we do not seek to evaluate the TFA algorithm but rather the framework's performance, we compare against Hyflow which also implements TFA. Comparisons between Hyflow and other distributed transactional memory libraries implementing different algorithms are available elsewhere [56], and have shown that Hyflow outperforms competitors under most circumstances. Experiments only targeted flat nesting.

Experiments were ran on a testbed consisting of one 48-core and three 24-cores AMD Opteron machines. The operating system used is Ubuntu Linux 10.04 Server. Every node communicates with every other node via TCP links above a Gigabit Ethernet connection. The

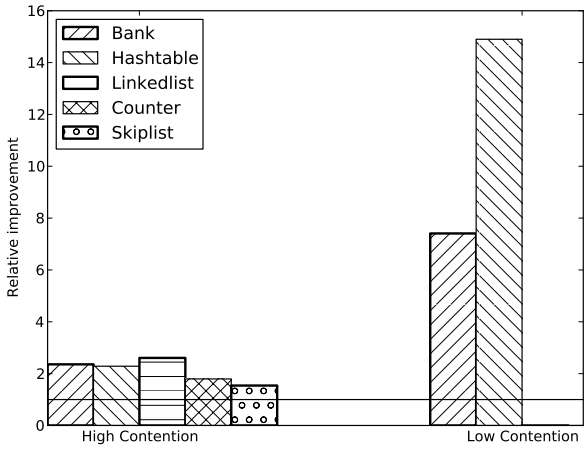


Figure 7.14: Summary of relative performance across benchmarks.

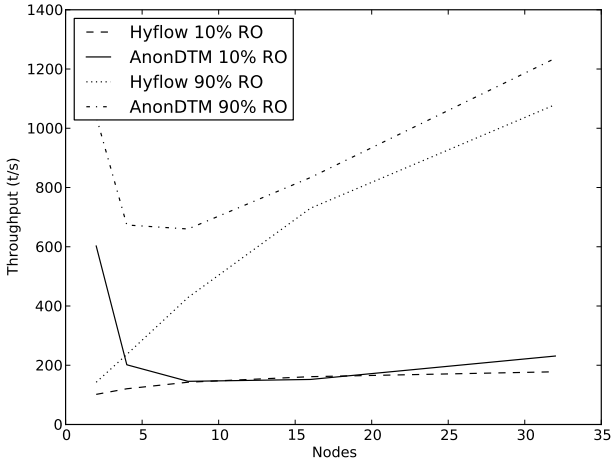


Figure 7.15: Throughput on Bank, for the high-contention workload, for different ratios of read-only transactions.

network is not saturated.

The JVM used is the 64-bit HotSpot(TM) Server VM. Benchmarks were run with Just-in-Time (JIT) compilation enabled. Each test was allowed a warm-up period to compensate for compilation and class loading overheads before measurement was started.

We evaluated Hyflow2 under two different scenarios. The first is a high-contention scenario. Up to 32 virtual nodes were spawned on a single 48-core machine. Each node is a JVM process whose execution is restricted to a single core, and transactions are spawned using a single thread for each node. Communication takes place via the *loopback* network interface. Each benchmark is configured with a minimal number of objects, such that contention is maximized. The ratio of read-only transactions was also varied between 10%, 50% and 90% reads. While this is not a genuinely distributed environment, it allows emphasizing the efficiency of the implementation.

The low-contention scenario is more realistic, with a large number of objects and many simultaneous transactions. Five nodes were spawned, with each node being allocated 24 cores, for a total of 120 cores. Each node spawns transaction using 96 threads, or 4 threads per each core.

Figure 7.15 provides details on one of the benchmarks, bank. The figure follows the throughput as the number of nodes is increased from two to 32 nodes in the high-contention scenario. Hyflow2 is very fast at a low number of nodes – up to 7 times faster than Hyflow. When the number of nodes is in middle of the range, the performance becomes comparable. Then, as more nodes are added, Hyflow2's performance benefit keeps increasing up to about 30%. Other benchmarks observed slightly different trends. For example, in hashtable, Hyflow2 is 60% faster or more compared to Hyflow throughout all our tests.

In the low-contention scenario, the results are completely different. Hyflow is barely able to maintain a reasonable throughput. After running transactions for about a minute, large pauses cause the throughput to drop significantly (we did not verify but believe this is due to the garbage collector). We therefore limited the experiment's duration such as this penalty is not incurred. Even so, Hyflow2 is about one order of magnitude faster. For example, on bank configured with 10,000 accounts and 50% read-only transactions, Hyflow managed to run 3,324 tps whereas Hyflow2 reached 24,623 tps. Results are summarized in Figure 7.14.

Chapter 8

Conclusions

In this thesis proposal we made several contributions aimed at improving the performance of Distributed Transactional Memory. We focused on the previously proposed Transactional Forwarding Algorithm, and extended it with support for closed nesting, open nesting and transaction checkpoints. We also presented Hyflow2, our new-generation DTM framework for the Java Virtual Machine.

Closed nesting through our TFA extension, N-TFA, proved insufficient for any significant throughput improvements. It ran on average 2% faster than flat nesting, while performance for individual test varied between 42% slowdown and 84% speedup. The observed behavior was highly workload dependent. Two of our benchmarks saw average slowdown. The workloads that benefit most from closed nesting are characterized by short transactions, with between two and five sub-transactions.

Open nesting, as exemplified by our TFA-ON implementation, showed promising results. We determined performance improvement to be a trade-off of the overhead of additional commits and the fundamental conflict rate. For write-intensive, high-conflict workloads, open nesting may not be appropriate, and we observed a maximum speedup of 30%. On the other hand, for lower fundamental-conflict workloads, open nesting enabled speedups of up to 167% in our tests. We identified that open nesting has a performance sweet spot in the middle of contention range: if contention is too low, the overheads of open nesting do not justify using it over flat nesting. If on the other hand contention is too high, the aborts caused by open nesting and the compensating actions they require cause throughput degradation.

Transaction checkpoints, using our TFA-CP implementation, showed 1-10% benefit over flat nesting when all the overheads are factored out. These overheads are in the 40-50% range for two out of three benchmarks, making them impossible to be matched by the benefits of partial rollback. The final benchmark, however, behaved unusually. On hash-table, a benchmark characterized by many short sub-transactions, simply using continuations and without regard to partial rollback, caused throughput to increase by 3-6x. We plan to

investigate the cause of this anomaly in the future.

Finally, we introduced Hyflow2, a new, high-performance DTM framework for the JVM. Hyflow2 is written in Scala and has a clean Scala API and a compatibility Java API. Hyflow2 is internally implemented using the actor model, and was on average two times faster than Hyflow on high-contention workloads, and up to 16 times faster, and more stable, on high-throughput, low-contention workloads.

8.1 Proposed Post-Prelim Work

A natural post-preliminary goal is to clean up the research already presented in this proposal into a more complete dissertation. To this purpose, we plan to test our algorithms (N-TFA, TFA-ON and TFA-CP) on industrial-strength benchmarks such as TPC-C, TPC-E, AuctionMark, TATP and others, where applicable. These tests will be ran on Hyflow2, to take advantage of the new, high-performance architecture and to contribute to the thesis' cohesiveness. We additionally want to figure out the cause behind our contradictory results when using checkpoints.

We next plan to move away from the cache-coherent, data-flow architecture and the TFA protocol. We want to provide fault-tolerance through network-based replication, similar to the Granola [14] and H-Store [30] projects described in Section 2.3. Also from Granola we adopt the independent transaction model, a lighter weight alternative to two-phase commit (2PC) coordinated transactions. Independent transactions, while distributed, do not incur the high cost of 2PC coordination on the critical transaction execution path, and are thus faster.

Based upon these assumptions, we propose to focus on increasing DTM performance via automatic data partitioning and replication. Previous work touched upon this topic in an SQL database environment. Schism [15] employs k-way graph cut heuristics in order to determine partitioning schemes which minimize 2PC coordinated transactions and maximize local, single-node transactions due to their relative performance difference. Their technique was shown to be competitive with the best known manual partitioning schemes on the benchmarks the authors used.

Our plan for future research is as follows.

8.1.1 Independent-Transaction Aware Automatic Partitioning

Our first goal is to extend the above-mentioned technique to also support independent transaction. Our technique would promote single-node transactions first and independent transactions next, while leaving 2PC coordinated transactions only as a last resort.

This can be done by analyzing a workload trace after encoding it as a graph. Data objects are represented in this graph as nodes. A transaction is encoded in this graph as clique: if objects O_1, O_2, ... O_N are accessed together in a transaction, then edges exist in the graph that connect every pair of objects out of O_1, O_2, ... O_N. Once the sample workload is encoded in this form, the graph can be partitioned into k subgraphs, such that a minimal number of edges are severed. The resulting partitioning scheme promotes single-node transactions, and the edges that were severed represent distributed transactions.

All independent transactions must meet the following criteria:

- The same abort decision can be reached at every repository independently. This can occur in two cases:
  - Transactions that never abort, for example, read-only transactions.
  - The abort decision is taken based on data replicated at all nodes.
- There are no data dependencies between branches executed at different nodes.

In the second step of this process, all atomic blocks will be analyzed to detect the applicability of the above mentioned criteria. This analysis can be performed statically either at compile time (by writing a Scala compiler plug-in) or post-compilation by using the byte-code output. Alternatively, the analysis could be performed dynamically at run-time by constructing a Markov model of all possible execution states, as in [44].

Based on this analysis, potential independent transactions can be identified and the partitioning scheme refined to reflect this knowledge.

8.1.2 Automatic Atomic-Block Refactoring

Our second goal is to automatically convert all atomic blocks in a program into the appropriate transaction model. Atomic blocks will be programmed by users without any regard for the partitioning scheme. An analysis step similar to the one described in the previous section will be applied to identify the most appropriate transaction model (i.e., single-node, independent and coordinated) for each atomic block.

Once the transaction model is identified, the atomic block must be split into several sub-transactions, one for each partition the atomic block executes at. This could be done at byte-code level, grouping data dependency relationships based on the partition the data belongs to. The multiple groups would then be reconstructed into specialized byte-code for each partition. Alternatively, thanks to Scala's nature as a functional programming language, it may be feasible and possibly simpler to implement this transformation as a compiler pass. For simplicity, the transformation may also restrict certain language features in the input code, for instance, loops, closures, and others.

If data dependencies cross partition borders, two alternatives are possible:

- Disallow this behavior. Encountering this condition means the partitioning scheme is wrong.
- Automatically pass the data between the two partitions as part of a coordinated transaction.

Such an automatic conversion would make the DTM system easy to use, as the user does not need to be aware of neither the underlying partitioning scheme, nor of the three different transaction models available. Automatic conversion also enables partitioning scheme evolution and automatic data migration, to keep up with dynamic workload demands.

8.1.3 Conversion of Coordinated Transactions

Our third goal is to determine whether the remaining coordinated transactions could possibly be broken down into two or more stages of single-node and/or independent transactions. Consider for example a bank transfer from account A to account B, where the accounts reside in different repositories. Withdrawing from account A first needs to check whether the funds are available, and if not, abort the transaction. The transfer must thus be implemented as a coordinated transaction, because the second repository has no way of independently determining whether the transaction would abort.

This transaction can however be converted into a two non-coordinated transactions:

- First a single-node transaction checks the available balance on A and somehow reserves the amount to be transferred (e.g., using an escrow account or locking account A altogether).
- Then an independent transaction that unconditionally withdraws the amount from A and deposits it on B.

We would like to devise a technique which would be able to analyze a coordinated transaction like the one described above, and propose a refactoring into a sequence of more efficient transactions (i.e., single-node and independent). This would most probably be achieved by employing compile-time or post-compilation byte-code analysis, in order to extract the sequence of execution states that leads to the decision of whether to abort the transaction. This sequence would then be removed from the body of the original transaction, and executed separately as a single-node transaction (let us call it the *guard transaction*). The guard transaction will be able to determine whether the second phase may proceed. If so, the guard transaction would acquire locks on the data it used to make its decision, and signal the second phase transaction to proceed.

This simple approach may not be possible in more complex transactions, e.g., if there are multiple execution paths that lead to aborts, or if the transaction logically breaks down in more than two phases. In these cases, we want to investigate whether we can relax the consistency properties of such a transaction, in order to allow a conversion as previously described.

Bibliography

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

[2] Kunal Agrawal, I.-Ting Angelina Lee, and Jim Sukha. Safe open-nested transactions through ownership. In Daniel A. Reed and Vivek Sarkar, editors, *PPOPP*. ACM, 2009.

[3] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, November 2009.

[4] Hagit Attiya, Vincent Gramoli, and Alessia Milani. A provably starvation-free distributed directory protocol. In *Proceedings of the 12th international conference on Stabilization, safety, and security of distributed systems*, SSS'10, pages 405–419, Berlin, Heidelberg, 2010. Springer-Verlag.

[5] Hagit Attiya and Alessia Milani. Transactional scheduling for read-dominated workloads. *J. Parallel Distrib. Comput.*, 72(10):1386–1396, October 2012.

[6] Annette Bieniusa and Thomas Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *IPDPS* [1].

[7] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. Software transactional memory for large scale clusters. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 247–258, New York, NY, USA, 2008. ACM.

[8] Nathan G. Bronson, Hassan Chafi, and Kunle Olukotun. Ccstm: A library-based stm for scala. Scala Days, April 2010.

[9] Joao Cachopo and Antonio Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, December 2006.

[10] Nuno Carvalho, Paolo Romano, and Luis Rodrigues. Asynchronous lease-based replication of software transactional memory. In *Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware*, Middleware '10, pages 376–396, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] Nuno Carvalho, Paolo Romano, and Luis Rodrigues. A generic framework for replicated software transactional memories. In *NCA*. IEEE Computer Society, 2011.

[12] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's globally-distributed database. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, OSDI'12, pages 251–264, Berkeley, CA, USA, 2012. USENIX Association.

[13] Maria Couceiro, Paolo Romano, Nuno Carvalho, and Luis Rodrigues. D2stm: Dependable distributed software transactional memory. In *Proceedings of the 2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*, PRDC '09, pages 307–313, Washington, DC, USA, 2009. IEEE Computer Society.

[14] James Cowling and Barbara Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, USENIX ATC'12, pages 21–21, Berkeley, CA, USA, 2012. USENIX Association.

[15] Carlo Curino, Evan Jones, Yang Zhang, and Sam Madden. Schism: a workload-driven approach to database replication and partitioning. *Proc. VLDB Endow.*, 3(1-2):48–57, September 2010.

[16] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. Norec: streamlining stm by abolishing ownership records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 67–78, New York, NY, USA, 2010. ACM.

[17] Peter Damron, Alexandra Fedorova, Yossi Lev, Victor Luchangco, Mark Moir, and Daniel Nussbaum. Hybrid transactional memory. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 336–346, New York, NY, USA, 2006. ACM.

[18] Michael J. Demmer and Maurice Herlihy. The arrow distributed directory protocol. In *Proceedings of the 12th International Symposium on Distributed Computing*, DISC '98, pages 119–133, London, UK, UK, 1998. Springer-Verlag.

[19] David Dice, Ori Shalev, and Nir Shavit. Transactional locking ii. In Shlomi Dolev, editor, *DISC*, volume 4167 of *Lecture Notes in Computer Science*. Springer, 2006.

[20] Aleksandar Dragojevic, Rachid Guerraoui, and Michal Kapalka. Stretching transactional memory. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming*

language design and implementation, PLDI '09, pages 155–165, New York, NY, USA, 2009. ACM.

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

[22] Rachid Guerraoui and Micha Kapaka. Opacity: A correctness condition for transactional memory, 2007.

[23] Rachid Guerraoui and Luís Rodrigues. *Introduction to Reliable Distributed Programming*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[24] Lance Hammond, Vicky Wong, Mike Chen, Brian D. Carlstrom, John D. Davis, Ben Hertzberg, Manohar K. Prabhu, Honggo Wijaya, Christos Kozyrakis, and Kunle Olukotun. Transactional memory coherence and consistency. In *Proceedings of the 31st annual international symposium on Computer architecture*, ISCA '04, pages 102–, Washington, DC, USA, 2004. IEEE Computer Society.

[25] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, SIGMOD '08, pages 981–992, New York, NY, USA, 2008. ACM.

[26] Tim Harris, James R. Larus, and Ravi Rajwar. *Transactional Memory, 2nd edition*. Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2010.

[27] Maurice Herlihy and Eric Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In Siddhartha Chatterjee and Michael L. Scott, editors, *PPOPP*, pages 207–216. ACM, 2008.

[28] Maurice Herlihy and J. Eliot B. Moss. Transactional memory: architectural support for lock-free data structures. In *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, pages 289–300, New York, NY, USA, 1993. ACM.

[29] Maurice Herlihy and Ye Sun. Distributed transactional memory for metric-space networks. In *DISC*, 2005.

[30] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2):1496–1499, August 2008.

[31] Junwhan Kim and Binoy Ravindran. Scheduling closed-nested transactions in distributed transactional memory. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 179–188, Washington, DC, USA, 2012. IEEE Computer Society.

[32] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with java stm.

[33] Eric Koskinen and Maurice Herlihy. Checkpoints and continuations instead of nested transactions. In *SPAA*, 2008.

[34] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Lujn, Chris Kirkham, and Ian Watson. Distm: A software transactional memory framework for clusters. In *In Proc. of the International Conference on Parallel Processing (ICPP*, pages 51–58, 2008.

[35] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 198–208, New York, NY, USA, 2006. ACM.

[36] Kevin E. Moore, Jayaram Bobba, Michelle J. Moravan, Mark D. Hill, and David A. Wood. Logtm: Log-based transactional memory. In *in HPCA*, pages 254–265, 2006.

[37] 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 John Paul Shen and Margaret Martonosi, editors, *ASPLOS*, pages 359–370. ACM, 2006.

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

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

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

[41] J. Eliot B. Moss and Antony L. Hosking. Nested transactional memory: model and architecture sketches. *Sci. Comput. Program.*, 63(2):186–201, December 2006.

[42] 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*, 2007.

[43] Marta Patino-Martinez, Ricardo Jimenez-Peris, Bettina Kemme, and Gustavo Alonso. Scalable replication in database clusters. In *Proceedings of the 14th International Conference on Distributed Computing*, DISC '00, pages 315–329, London, UK, UK, 2000. Springer-Verlag.

[44] Andrew Pavlo, Evan P. C. Jones, and Stanley Zdonik. On predictive modeling for optimizing transaction execution in parallel oltp systems. *Proc. VLDB Endow.*, 5(2):85–96, October 2011.

[45] Fernando Pedone, Rachid Guerraoui, and André Schiper. The database state machine approach. *Distrib. Parallel Databases*, 14(1):71–98, July 2003.

[46] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues. When scalability meets consistency: Genuine multiversion update-serializable partial data replication. In *Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on*, pages 455–465. IEEE, 2012.

[47] Dmitri Perelman, Anton Byshevsky, Oleg Litmanovich, and Idit Keidar. Smv: selective multi-versioning stm. In *Proceedings of the 25th international conference on Distributed computing*, DISC'11, pages 125–140, Berlin, Heidelberg, 2011. Springer-Verlag.

[48] Dmitri Perelman, Rui Fan, and Idit Keidar. On maintaining multiple versions in stm. In *Proceedings of the 29th ACM SIGACT-SIGOPS symposium on Principles of distributed computing*, PODC '10, pages 16–25, New York, NY, USA, 2010. ACM.

[49] Matteo Pusceddu, Simone Ceccolini, Gianluca Palermo, Donatella Sciuto, and Antonino Tumeo. A compact transactional memory multiprocessor system on fpga. In *Proceedings of the 2010 International Conference on Field Programmable Logic and Applications*, FPL '10, pages 578–581, Washington, DC, USA, 2010. IEEE Computer Society.

[50] Paolo Romano, Nuno Carvalho, Maria Couceiro, Luis Rodrigues, and Joao Cachopo. Towards the integration of distributed transactional memories in application servers' clusters. In *Quality of Service in Heterogeneous Networks*, volume 22 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 755–769. Springer Berlin Heidelberg, 2009.

[51] Paolo Romano, Luís Rodrigues, Nuno Carvalho, and João P. Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *Operating Systems Review*, 44(2):1–6, 2010.

[52] M. M. Saad and B. Ravindran. Supporting stm in distributed systems: Mechanisms and a java framework. In *TRANSACT*, San Jose, California, USA, June 2011.

[53] Mohamed M. Saad. Hyflow: A high performance dstm framework. Master's thesis, Virginia Tech, 2011.

[54] Mohamed M. Saad and Binoy Ravindran. Hyflow: a high performance distributed software transactional memory framework. In Arthur B. Maccabe and Douglas Thain, editors, *HPDC*, pages 265–266. ACM, 2011.

[55] Mohamed M. Saad and Binoy Ravindran. Snake: Control flow distributed software transactional memory. In Xavier Défago, Franck Petit, and Vincent Villain, editors, *SSS*, volume 6976 of *Lecture Notes in Computer Science*, pages 238–252. Springer, 2011.

[56] Mohamed M. Saad and Binoy Ravindran. Transactional forwarding algorithm. Technical report, Virginia Tech, January 2012.

[57] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh, and Benjamin Hertzberg. Mcrt-stm: a high performance software transactional memory system for a multi-core runtime. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 187–197, New York, NY, USA, 2006. ACM.

[58] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, PODC '95, pages 204–213, New York, NY, USA, 1995. ACM.

[59] Michael F. Spear, Maged M. Michael, and Christoph von Praun. Ringstm: scalable transactions with a single atomic instruction. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, SPAA '08, pages 275–284, New York, NY, USA, 2008. ACM.

[60] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, VLDB '07, pages 1150–1160. VLDB Endowment, 2007.

[61] Alexandru Turcu and Binoy Ravindran. On open nesting in distributed transactional memory. Technical report, Virginia Tech, 2012.

[62] Alexandru Turcu, Binoy Ravindran, and Mohamed M. Saad. On closed nesting in distributed transactional memory. In *TRANSACT*, 2012.

[63] Haris Volos, Adam Welc, Ali-Reza Adl-Tabatabai, Tatiana Shpeisman, Xinmin Tian, and Ravi Narayanaswamy. Nepaltm: Design and implementation of nested parallelism for transactional memory systems. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 123–147, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[65] Bo Zhang and Binoy Ravindran. Location-aware cache-coherence protocols for distributed transactional contention management in metric-space networks. In *Proceedings of the 2009 28th IEEE International Symposium on Reliable Distributed Systems*, SRDS '09, pages 268–277, Washington, DC, USA, 2009. IEEE Computer Society.

[66] Bo Zhang and Binoy Ravindran. Dynamic analysis of the relay cache-coherence protocol for dtm. In *IPDPS* [1].