HyflowCPP: A Distributed Transactional Memory framework for C++

Sudhanshu Mishra

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

Master of Science
in
Computer Engineering

January 28, 2013
Blacksburg, Virginia

Keywords: Distributed Transactional Memory, Transactional Framework, C++,
Distributed Concurrency

HyflowCPP: A Distributed Transactional Memory framework for C++

Sudhanshu Mishra

(ABSTRACT)

The distributed transactional memory (DTM) abstraction aims to simplify the development of distributed concurrent programs. It frees programmers from the complicated and error-prone task of explicit concurrency control based on locks (e.g., deadlocks, livelocks, non-scalability, non-composability), which are aggravated in a distributed environment due to the complexity of multi-node concurrency. At its core, DTM's atomic section-based synchronization abstraction enables the execution of a sequence of multi-node object operations with the classical serializability property, which significantly increases the programmability of distributed systems.

In this thesis, we present the first ever DTM framework for distributed concurrency control in C++, called *HyflowCPP*. HyflowCPP provides distributed atomic sections, and pluggable support for concurrency control algorithms, directory lookup protocols, contention management policies, and network communication protocols. The framework uses the Transaction Forwarding Algorithm (or TFA) for concurrency control. While there exists implementations of TFA and other DTM concurrency control algorithms in Scala and Java, and concomitant DTM frameworks (e.g., HyflowJava, HyflowScala, D2STM, GenRSTM), HyflowCPP provides a uniquely distinguishing TFA/DTM implementation for C++. In addition, HyflowCPP supports strong atomicity, transactional nesting models including closed and open nesting (supported using modifications to TFA), and checkpointing.

We evaluated HyflowCPP through an experimental study that measured transactional throughput for a set of micro- and macro-benchmarks, and comparing with competitor DTM frameworks. Our results revealed that HyflowCPP achieves up to 600% performance improvement over competitor Java DTM frameworks including D2STM, GenRSTM, HyflowScala and HyflowJava, which can be attributed to the competitors' JVM overhead and rudimentary networking support. Additionally, our experimental studies revealed that checkpointing achieves up to 100% performance improvement over flat nesting and 50% over closed nesting. Open nesting model achieves up to 140% performance improvement over flat nesting and 90% over closed nesting.

This work is supported in part by US National Science Foundation under grants CNS 0915895, CNS 1116190, CNS 1130180, and CNS 1217385. Any opinions, findings, and conclusions or recommendations expressed in this work are those of the author and do not necessarily reflect the views of the National Science Foundation.

Dedication

I dedicate this thesis to my family and friends.

Without their support, this would not have been possible

Acknowledgments

I would like to thank my advisor, Dr. Binoy Ravindran, for his help and guidance on both technical and personal topics. It has been an honor to work under him and I am highly thankful to him for his trust in me.

I would also like to thank Dr. Robert Broadwater and Dr. Mark Jones for serving on my committee and providing their valuable feedback and direction. In addition, I would like to thank all of my colleagues at the Systems Software Research lab. I would particularly like to thank Alex Turcu, Mohd. Saad and Aditya Dhoke for their support and encouragement. It was a pleasure to work with them and perform interesting research in area of Distributed Transactional Memory.

Finally, I would like to thank my family and friends for all the love and support they have given me, without which this thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As Moore's law [80] is approaching towards the scaling limit, and processor clocks are hitting the power wall [54], chip manufacturers are progressively endowing the multi-core CPU architecture [86]. Augmenting computing power by increasing processor clock rates and transistor scaling are no longer feasible [87]. Consequently, application performance can only be improved by exposing greater concurrency in software that exploits the hardware parallelism [47].

Writing concurrent code is a daunting task for an ordinary programmer. Such code requires proper synchronization and co-ordination between different parallel tasks. Lock-based concurrency control is often used as the synchronization abstraction. Coarse-grain locking is simple to implement, and protects concurrent code by locking over a single, large critical section, but permits little concurrency. On the other hand, fine-grain locking decomposes a large critical section into a set of smaller critical sections to obtain greater concurrency. However, implementing fine grained locks is highly complex and prone to programmer errors such as deadlocks, livelocks, lock convoying, and priority inversion [68]. Additionally, lock-based synchronization is non-composable. For instance, a thread-safe collection may support atomic insertion or removal of elements using an internal lock. However, removing an element from one collection and inserting it into another in a thread safe manner cannot be performed safely using the collection's internal lock. Rather, this requires additional synchronization support for locking multiple collections at the same time [93]. These issues with locks make concurrent code difficult to understand, program, modify, and maintain.

Concurrency control has been well studied in the field of database systems, where *transactions* have been a highly successful abstraction to make different operations access a database simultaneously without observing interference [36]. Transactions guarantee the four so-called ACID properties [36]: atomicity, consistency, isolation, and durability. This behavior is implemented by controlling access to shared object and undoing the actions of a transaction that did not complete successfully (i.e., roll back). Inspired by this success, transactional memory (TM) was proposed as an alternative model for accessing shared in-memory objects, without exposing locks in the programming interface, to avoid the drawbacks of locks. TM originated as a hardware solution (HTM) [45], was later extended in software, called STM [82], and subsequently to a hybrid model [28]. With TM, programmers organize code that read/write shared objects as transactions, which appear to execute atomically. Two transactions conflict if they access the same object and one access is a write. When that happens, a conflict resolution policy (e.g., contention manager [44]) resolves the conflict by aborting one and allowing the other to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). In addition to providing a simple programming model, TM provides performance comparable to lock-based synchronization [79].

The difficulties of lock-based concurrency control are exacerbated in distributed systems, due to (additional) distributed versions of their centralized problem counterparts: distributed deadlocks, distributed livelocks, and distributed lock conveying are orders-of-magnitude harder to debug than in multiprocessors. Moreover, code composability is even more desirable in distributed systems due to the difficulties of reasoning about distributed concurrency, and the need to support replication or incorporate backup mechanisms that are needed to cope with failures (one of the raison d'être for building distributed systems). The success of multiprocessor TM has therefore similarly motivated research on distributed TM.

Distributed TM (DTM) can be classified based on the system architecture: cache-coherent DTM (cc DTM) [46, 98, 100, 101, 78, 53, 76, 75, 91], in which a set of nodes communicate with each other by message-passing links over a communication network, and a cluster model (cluster DTM) [20, 61, 57, 26, 73, 74], in which a group of linked computers works closely together to form a single computer. The most important difference between the two is communication cost. cc DTM assumes a *metric-space* network (i.e., the communication cost between nodes form a metric), whereas cluster DTM differentiates between local cluster memory and remote memory at other clusters. cc DTM uses a cache-coherence protocol to locate and move objects in the network, satisfying object consistency properties.

Similar to multiprocessor TM, DTM provides a simple distributed programming model (e.g., locks are entirely precluded in the interface), and performance comparable to distributed lock-based concurrency control [26, 53, 20, 73, 74, 61, 98, 99, 101, 78, 53, 76, 75].

To define the thesis problem space and describe the thesis's research contributions, we now discuss key dimensions of the TM problem space.

1.1 The Transactional Memory Problem Space

The dimensions of the TM problem space that are of interest to this thesis include memory consistency models, concurrency control mechanisms, replication mechanisms, atomicity semantics, and partial abort models. We discuss each of these in the subsections that follow.

1.1.1 Memory Consistency Models

A *memory consistency* model [10] is an agreement between the TM framework and the programmer that specifies the memory access rules. The rules ensure that the memory remains consistent and predictable after different memory operations. If programmer follows the rules, the framework guarantees memory consistency. The memory operations are generally provided at a low level of abstraction such as read, write, or compare and swap. The memory consistency models that are typically used in TM and DTM frameworks include serializability [15], opacity [37], and snapshot isolation [14].

Serializability is a strong consistency requirement and requires that all transactions execute in complete isolation i.e., concurrent transactional execution must be equivalent to sequential execution. Two or more transactions can execute at the same time only if the equivalence to a serial execution can be maintained. DTM works like D2STM [26], Cluster-STM [20], and GenRSTM [23] support serializability.

Opacity is a stronger consistency requirement in comparison to serializability. It not only requires are transactions to execute in isolation, but also prevents even non-committed transactions from accessing inconsistent states of system. DTM works like HyflowJava [78] and HyflowScala [88] provided opacity guarantee.

Strong consistency requirements like serializability and opacity can potentially cause lower performance. Thus, in recent years, many researchers have focused on weaker consistency models, such as snapshot isolation (SI) and eventual consistency (EC) [92].

With *snapshot isolation*, a transaction takes an individual snapshot of the memory at the start of the transaction. When the transaction finishes, it commits only if the values of objects in its snapshot have not been updated by other committed transactions. DecentSTM [16] is a good example of snapshot isolation DTM work. *Eventual consistency* guarantees that if no new updates are made for a period of time, all objects will be updated to their most recent values. Amazon Dynamo [29] is a distributed concurrency product based on EC.

Even though weaker consistency models provide higher performance by relaxing the consistency requirement, it forces programmers to embrace the relaxed consistency models. Such models typically pose a greater challenge for ordinary programmers, as now they must understand all the subtleties of complex consistency properties to ensure program correctness. This is likely why, the serializability property, despite its negative impact on performance, has gained significant traction, especially in database settings.

1.1.2 Concurrency Control Mechanisms

Concurrency control ensures that concurrent changes made by transactions do not violate the consistency model supported by the system – i.e., it ensures the correctness of results generated through concurrent operations with respect to the consistency model at hand. Concurrency control mechanisms often use locks for a period of time. Broadly, there are two types of concurrency control mechanism: pessimistic concurrency control and optimistic concurrency control.

Pessimistic concurrency control (PCC) [69] is based on the premise that all object accesses will result in conflicts. Therefore, as soon as any object access is made in a transaction, locks are acquired on the object. Depending upon the lock type (e.g., shared, exclusive), an object may be shared with other transactions. The well-known two phase locking protocol [59] is an example of a PCC mechanism.

Optimistic concurrency control (OCC) [42] is based on the premise that conflicts are possible, but very rare. Therefore, no locks are acquired at the time of object access, and objects are read from, and written to without acquiring any lock. Before commit, a validation step is performed to check for any conflicting transaction. If a conflict is detected, the current transaction is aborted and retried. OCC generally involves four steps:

1. Begin: marks the start of a transaction.
2. Modify: read and write operations are performed on an object.
3. Validate: verifies whether any conflicting updates were made on accessed objects.
4. Commit/Abort: based on validation, in this step, the transaction is committed or aborted.

1.1.3 Replication

Replication is a commonly used technique in databases for fault-tolerance and improved concurrency [50]. In the DTM setting, the classical replication solution can be viewed as a primary-backup model [22, 35], where the primary object copy is used to update all backup copies. In case of primary failure, the backup copies are used to service transactions. The switch from primary to backup copies creates many problems for the transactions in progress, at the time of failure. Many systems send log updates to backup copies, to allow in-progress transactions to continue from a certain logged state [96]. In case of high update workload, too much overhead for full replication renders primary-backup approach inefficient. In such conditions a partial replication [13] technique is used, which replicates only some specific entries and attributes. The entries and attributes to be replicated are specified based on partial replication protocol. Another popular replication technique is state machine approach [81]. In the replicated state machine approach, one machine is the master and the others are slaves. Master receives all the requests from clients and executes them on all replicas in the same order ensuring identical state for each replica.

Replication is also viewed as a way to increase performance through localization. Replication reduces the network communication cost for the distributed read transactions significantly as all the requested objects are available locally. Same cost advantage in not available for write transactions, as each write updates all the replicas of objects, which incurs additional messaging cost. In such cases, partial replication can provide an intermediate approach for replication. In some cases, partial replication with weaker consistency has already shown magnitudes of performance improvement for distributed transactions. [72]

1.1.4 Atomicity Semantics

Two important atomicity semantics studied in the TM literature include strong atomicity and weak atomicity. Strong atomicity [9] ensures that all inconsistent states of the system are hidden from non-transactional code. In essence, strong atomicity treats non-transactional code as executing in its own singleton atomic transaction. This behavior prevents non-transactional code from accessing any unprotected shared variables outside the transactional code.

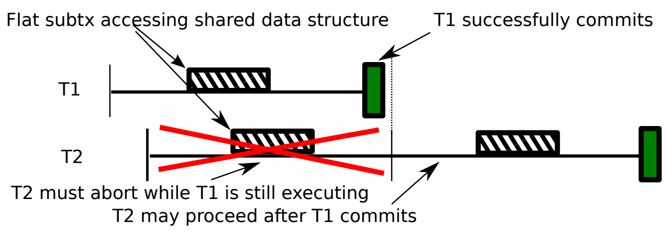
In contrast to strong atomicity, the weak atomicity [19] property ensures that transactional code executes atomically only with respect to other transactional code. This means that, non-transactional code can still access shared variables in an inconsistent state. Stronger atomicity thus clearly promotes programmability: it helps the programmer by preventing any potentially buggy interleaving.

1.2 Partial Abort Models

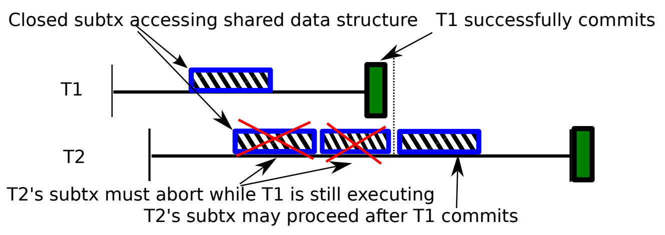
In TM and DTM, the two common techniques that have been studied to improve performance through partial transactional abort, instead of full transactional abort, include *nesting* [63] and *checkpointing* [56]. A transaction is said to be nested when it encloses some other transaction. Nesting also promotes code composability when integrating with external libraries: it enables a transaction to invoke calls to a library that are transactional. Different transactional nesting models have been studied in the past: *Flat*, *Closed* [91], and *Open* [89].

Flat nesting is the simplest form of nesting, which simply ignores the existence of transactions in inner code. All operations are executed in the context of the outermost transaction. Aborting any inner transaction causes the parent transaction to abort. Thus, no partial rollback can be performed with this model. Flat nesting does not provide any performance improvement over non-nested transactions.

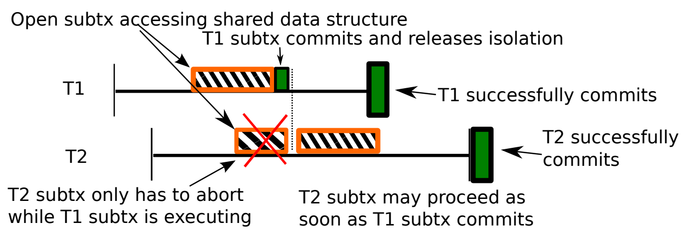
Closed nesting allows inner transactions to abort individually. Abort of an inner transaction does not lead to abort of the parent transaction. However, inner transactions' commit are not visible outside the parent transaction. An inner transaction commits to the internal memory


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

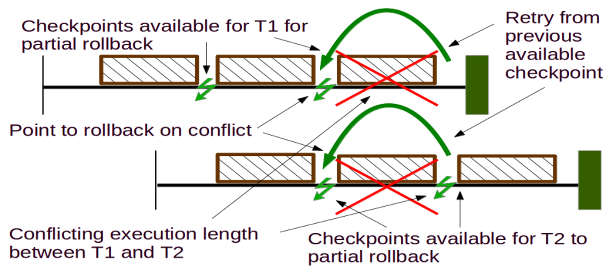
(a) Flat nesting


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

(b) Closed nesting


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

(c) Open nesting


Checkpoints available for T1 for partial rollback
Retry from previous available checkpoint
Point to rollback on conflict
Conflicting execution length between T1 and T2
Checkpoints available for T2 to partial rollback

(d) Checkpointing

Figure 1.1: Simple example showing the execution time-line for two transactions under different transactional nesting models and checkpointing.

of its parent transaction, and the commit is visible outside of the parent transaction only when the parent transaction commits.

Open nesting uses the higher level of abstraction for memory access to avoid any false conflict occurring at memory levels. It allows inner transactions to commit or abort individually, and their commits are visible globally to all other transactions. In case of abort of the outermost transaction, due to any fundamental conflict at higher abstractions, all the inner transactions are roll-backed through predefined compensating action for each inner transaction.

Checkpointing does not view a transaction as a composition of multiple inner transactions, but as a process containing various states. [55] The idea is to save the transaction execution state at various points, called 'checkpoints', and use them to partially rollback to a checkpointed state in case of a conflict. This allows the transaction to redo the work only for that part of the transaction in which the conflict occurs. The transaction execution state is generally saved using continuations [32].

Figure 1.1 [90] illustrates the difference between the various transactional nesting models and checkpointing using two transactions $T1$ and $T2$. In flat nesting, upon a conflict at a shared object between transactions $T1$ and $T2$, $T2$ must fully abort. $T2$ can later restart and commit, when the shared object is released at the end of $T1$'s execution.

In case of closed nesting, $T2$'s inner transaction incurs an abort. However, it saves all other previously committed inner transactions from being aborted. Inner transactions can continue as soon as $T1$ releases the shared object.

With open nesting, the aborted inner transaction of $T2$ does not have to wait until the end of $T1$'s execution, as in closed nesting. Instead, it can continue as soon as $T1$'s inner transaction, which uses the shared object, commits and releases the object.

With checkpointing, upon a conflict, $T2$ can resume from the last valid checkpoint and keep retrying until $T1$ releases the conflicting object at the end of its execution.

1.3 Thesis Contribution: a DTM Framework for C++

Most of the existing DTM implementations are created in VM-based languages (e.g. Java, Scala). Examples include Cluster-STM [20], D2STM [26], DiSTM [57], GenRSTM [23], HyflowJava [78], and HyflowScala [88]. A few DTM implementations such as DMV [61] (C++), Cluster-STM [20] (C), and Sinfonia [12] (C++) were developed in C & C++. However, DMV [61] and Cluster-STM [20] are implemented as proofs-of-concept for DTM algorithms, and Sinfonia [12] is designed as a backup and restore service, rather than as general-purpose DTM frameworks.

C++ is one of the most popular programming languages [5]. It is implemented in wide variety of hardware and operating system platforms. Many high-performance production systems are developed in C++, instead of VM-based languages, usually, to overcome the performance issues inherent of VM-based languages [49]. Therefore, a C++ framework for DTM is highly desirable as that enables DTM support for such a popular language and provides high performance in C++ applications using DTM concurrency control. In C++, execution states can be saved without incurring significant overheads [3]. Memory management is also manual, which eliminates overheads of garbage collection (GC) in VM environments, which is superfluous anyway since a DTM framework does its own memory management. Thus, a C++ DTM framework is a useful, previously unexplored environment, for examining the differences and trade-offs between different transactional nesting and checkpointing models.

Motivated by this observation, we design and implement the first ever DTM framework for C++: HyflowCPP. HyflowCPP has a modular architecture and provides pluggable support for DTM protocols, cache coherence protocols, contention management policies, and networking protocols. HyflowCPP provides a simple atomic section-based DTM interface that excludes locks. The current implementation uses the TFA algorithm [77], which is a dataflow-based cc DTM protocol. Additionally, HyflowCPP supports strong atomicity, partial abort models including closed and open nesting, and checkpointing. None of the past DTM systems support all these, and in particular for C++.

Table 1.1 summarizes HyflowCPP's contribution. Each row of the table describes a DTM implementation, and each column describes a DTM property. Thus, the table entries describe the features supported by the different DTMs, illustrating HyflowCPP's uniquely

Implementation	Serializability	Replication	MultiVersioning	Strong Atomicity	checkpointing	Closed-Nesting	Open-Nesting	Target Language
DMV [61]	✓	✓	✗	✗	✗	✗	✗	C++
Cluster-STM [20]	✓	✗	✗	✓ [1]	✗	✗	✗	C & SQL
DiSTM [57]	✓	✗	✗	✗	✗	✗	✗	Java
D2STM [26]	✓	✓	✗	✗	✗	✗	✗	Java
AGGRO [70]	✓	✓	✗	✗	✗	✗	✗	Java
Sinfonia[2] [12]	✓	✓	✗	✗	✗	✗	✗	C++
GenRSTM [23]	✓	✓	✗	✗	✗	✗	✗	Java
GTM [83]	✓	✗	✗	✗	✗	✗	✗	Chapel
HyflowJava [78]	✓	✗	✗	✗	✗	✓	✓	Java
HyflowScala [88]	✓	✗	✗	✓	✓	✗	✗	Scala
Granola [27]	✓	✗	✗	✗	✗	✗	✗	Java
HyflowCPP	✓	✗	✗	✓	✓	✓	✓	C++
Decent RSTM [16]	✗	✓	✓	✓	✗	✗	✗	Java
GMU [72]	✗	✓	✓	✗	✗	✗	✗	Java

[1] Supported through programming restriction
[2] Developed as service for backup and restore

Table 1.1: Comparison of DTM implementations and HyflowCPP's unique features.

distinguishing features.

We evaluated HyflowCPP through a set of experimental studies that measured transactional throughput on an array of micro and macro-benchmarks, and compared with past VM-based DTM systems such as GenRSTM, DecentSTM, HyflowJava, and HyflowScala. Key result from our experimental studies is that, HyflowCPP outperforms its nearest competitor HyflowScala maximum up to 6 times for a highly distributed system i.e., single core is equivalent to single node and only one transactional thread run on a single core. Other competitors like GenRSTM, DecentSTM, and HyflowJava perform even worse in comparison to HyflowScala. In highly distributed systems, the lower network latency and high CPU utilization of HyflowCPP actually converts into higher throughput gains. In contrast, when we ran our experiments with each node having multiple transactional threads, we achieved maximum throughput improvement up to 2 to 5 times only in comparison to HyflowScala as network latency dominance reduces and CPU utilization increases in HyflowScala.

Additionally, we show that HyflowCPP's checkpointing-based partial rollback mechanism outperforms flat nesting by as much as 100% and closed nesting by as much as 50%. Open nesting based transactions outperform flat nesting by as much as 140% and closed nesting by as much as 90%. These trends are consistent with past DTM studies on nesting [91, 89], but our relative improvements (i.e. checkpointing and open nesting when compared to closed nesting) are higher, which can be attributed to a more efficient design and the lower overheads of various mechanisms in C++ [49].

1.4 Thesis Organization

The rest of the thesis is organized as follows: Chapter 2 overviews past and related works in DTM, and contrast them with the thesis's problem space. Chapter 3 details the programming model of HyflowCPP. Chapter 4 describes HyflowCPP's architecture. Chapter 5 discusses the TFA algorithm and its extension to different transactional nesting and checkpointing models. We report our experimental results in Chapter 6. Finally, we conclude the thesis in Chapter 7.

Chapter 2

Related Work

In this chapter, we survey the past and related work focusing on the problem of distributed transactional memory (DTM). For our discussion purpose, we classify the past and current work in DTM based on different transactional properties:

1. Serializability
2. Non-Serializability
3. Replication
4. Strong Atomicity
5. Transaction Models

In each section we describe the various research works and related transactional property definitions. In last section we also describe recently developed various unconventional systems.

2.1 Serializable DTM implementations

In previous chapter 1, we have intuitively described serializability property. Here we provide a formal definition of serializability.

Definition 2.1.1 (Serializability). *A set of transactions executes serially if each transaction executes its write operation before the next transaction executes its read operation. That is, the transactions are in no way interleaved. A serial execution of transactions preserves database consistency because each individual transaction preserves database consistency. If* *an interleaved execution of transactions produces the same effect as a serial execution of those same transactions, then the execution is called serializable. Since a serial execution preserves consistency, a serializable execution also preserves consistency. [15]*

In lock-based concurrency control implementation, serializability requires that locks to be acquired in certain range of execution. While in non-lock concurrency control, no lock is acquired, but if the system detects a concurrent transaction in progress it rollbacks.

One of the first work in DTM by Manassiev [61] supported the serializability. It introduced a novel page-level distributed concurrency control algorithm, called Distributed Multiversioning (DMV). DMV allows each node to keep a single local copy of objects to read or write. At commit time, page differences are broadcasted to all other replicas, and a transaction commits successfully upon receiving acknowledgments from all nodes. A central timestamp is employed, which allows only a single update transaction to commit at a time. Unfortunately, this requirement of transaction to acquire a cluster-wide unique token, which globally serializes the commit phases of transactions, imposes considerable overhead and seriously hampers performance as later described by Kotselidis et. al. [57].

Cluster-STM [20] work based on PGAS [17] programming model supported serializability. In Cluster-STM, the dataset is partitioned across the nodes and each object is assigned a home node. Home node maintains the authoritative version of object and synchronizes the accesses of conflicting remote transactions. Being based on PGAS [17] programming model Cluster-STM does not distinguish transaction that execute in the same node from the transaction that execute on a different node and pays a heavy performance penalty for not exploiting shared memory for intra-node communication.

DiSTM [57] work published in 2008 uses a distributed mutual exclusion mechanism to coordinate the commit of transactions. This mechanism ensures that no two conflicting transactions try to commit simultaneously. To provide distributed mutual exclusion this protocol grants lease to nodes on datasets, based on their data access patterns, for each transaction commit. It allows transaction to escape performance penalty incurred by serialization cost in commit phase. However, it still becomes a bottleneck in contention intensive workloads. In addition, DiSTM suffers the scalability issues due to the single coordinating node performing lease establishment mechanism as the number of nodes increases. Due to this bottleneck, DiSTM provides a dedicated node to perform lease establishment mechanism.

In 2009 Dependable Distributed Software Transactional Memory (D2STM) [26] followed which utilized the atomic broadcast [30] and bloom filter [18] certification to achieve good performance in a replicated cluster. Replication of objects allows it to execute all read only transactions locally without incurring in any network communication overhead. For write transactions, D2STM first validates it locally and aborts if required, on basis of locally available information. After validating locally, replication manager encodes the transaction read-set in a bloom filter and atomically broadcasts it along with the transaction write-set. Even though atomic broadcast allowed the D2STM to support serializability, it incurs high performance cost with increase in the number of nodes in cluster. Additionaly, use of bloom filter requires the prior knowledge of transaction read-set to fine tune for reduced false positives. Same research group later came up with AGGRessively Optimistic concurrency control scheme (AGGRO) [70] to address dependability issue in DTM utilizing the replication. AGGRO propagates dependencies across uncommitted transactions in a serialization order compliant with the optimistic message delivery order provided by the optimistic atomic broadcast (OAB) [71] service. Even though OAB improved performance, it also made it prone to saturation issue with the OAB group communication subsystem.

In 2009 another work, namely, Sinfonia [12] service came out which utilized the mini-transactions. The idea behind the mini-transactions was to send the transaction itself as a piggyback in first phase of two-phase commit. All the transactions for which conditional value and update object exist on same node can be converted into a mini-transaction. Utilizing the mini-transaction Aguilera et. al. were able to reduce the network communication to a large extent and achieve good performance. Drawback of this approach is that we need to know the data accessed by transaction beforehand.

Cloud-TM [74] work published in 2010 enumerated the features, which can be useful to make DTM successful in providing concurrency solution over network cloud. They suggested making DTM easily graspable by hiding complexities and making it capable to cope up with workload heterogeneity. They also make a point to maximizing locality and automatic resource provisioning for a high performance and adaptability of system. They asked DTM community to support durability to survive in failure prone cloud environment.

In 2011, based on D2STM and AGGRO work Romano et. al. came up with a generic framework for Replicated Software Transactional Memories (GenRSTM) [23]. Goals of this framework was to simplify the development and testing of new replication protocols and DTM algorithms, provide high decoupling between the architecture building blocks, and support multiple implementations of the architecture building block. This framework enabled system administrators to seek optimal performance as a function of the workload/deployment scenario by reconfiguring the replicated STM middle-ware platform, in a transparent fashion for the user level application. It simplified the development and evaluation of alternative replication protocols.

In same year Srinivas et. al. from Oak Ridge National Laboratory published a technical report [83] on language based software transactional memory (STM) for distributed memory systems. In Chapel [24], a general-purpose parallel language, they provided atomic semantics and pluggable compiler support for multiple DTM implementations. They also provided a prototype distributed STM implementation Global Transactional Memory 2 (GTM2) an enhancement over GTM [84] work published in 2009 based on remote procedure call (RPC) to provide DTM support. In GTM2 simple RPC was improved with read versioning, deferred update, and eager acquire scheme.

Recently in 2012, Granola [27] work from MIT provided the support for the serializability. It divides the transactions in three different categories: Single-repository transactions executing using objects within a repository, coordinated distributed transactions executing using objects from more than one repositories, independent distributed transactions executing atomically across a set of repositories and committing independently. Coordinated distributed transactions follow the traditional two-phase commit voting protocol and provide the current state of art performance. Meanwhile, single-repository transactions and independent distributed transactions use time-stamp synchronization without locking.

Many works in DTM have also supported stronger consistency properties like opacity. We have already described opacity in chapter 1. Here we provide a formal definition.

Definition 2.1.2 (**Opacity**). *A transactional memory system follows opacity criteria if: (1) all operations performed by every committed transaction appear as if they happened at some single, indivisible point during the transaction lifetime, (2) no operation performed by any aborted transaction is ever visible to other transactions (including live ones), and (3) every transaction always observes a consistent state of the system. [37]*

In 2011, HyflowJava framework [78] for DTM was released for non-replication based systems. Later in 2012, a DTM framework in Scala language [88] was released which showed improvement over previous framework. Both of these frameworks, supported opacity and utilized Transaction Forwarding Algorithm (TFA) for their prototype implementation. TFA uses an asynchronous clock-based validation technique to ensure DTM concurrency control. HyflowCPP framework also uses TFA as its base transactional algorithm. We will describe TFA algorithm later in detail in Chapter 5.

2.2 Non-Serializable DTM implementations

Many researchers have contented the serializability as a very strong criterion and used the weaker criteria to provide the high performance for DTM. Here, we describe some famous non-serializable consistency criteria and related work in DTM.

Snapshot isolation is very famous consistency criteria often used in the database systems. Some researchers have developed the DTM systems based on this criterion. Snapshot isolation (SI) can be formally described as following:

Definition 2.2.1 (**Snapshot Isolation**). *SI is defined by two properties: Snapshot-read requires that a transaction T reads data from a snapshot, which contains all updates, committed before T starts (plus its own updates). Snapshot-write requires that no two concurrent transactions may write the same object; that is, if two concurrent transactions want to write the same data item only one of them will be allowed to commit. [14, 60]*

Snapshot-read is typically implemented through a multi-version system, where read operations access previously committed versions. Multi-versioning is often used in databases to provide the concurrent access. It can be defined as below:

Definition 2.2.2 (**Multi-Versioning**). *In multi-version concurrency control (MVCC), each write on an object x creates a new copy of it. All copies of same object are given proper version based on history value. MVCC provides time-consistent views for the system. It allows the delayed read operations to execute successfully by providing the object version relevant to transaction's time-stamp.*

Conflict detection for snapshot-write can be implemented through locking or validation. In SI read and write skew anomalies can occur, which happens when two transactions concurrently read an overlapping data set make disjoint updates, and finally concurrently commit. Neither of transaction see update performed by the other.

One of the first paper on weaker consistency model SI was DecentSTM [16] in 2010. DecentSTM algorithm keeps limited list of committed versions of all shared data and lazily obtains a consistent memory snapshot during execution. By choosing a version upon read, a transaction determines on which versions it depends. In fact, with unlimited version history, a read only transaction would never have to abort, because it could always read a previous version that does not conflict with the data read so far. For coincidental commits, DTM uses a voting based randomized consensus protocol. Using snapshot isolation do provide the higher performance in DecentSTM, but it also adds up additional memory overhead of maintaining version-ed objects.

In 2011 Nuno et. al. in DiasSTM [31] came up with approach of static analysis for transactional code to provide serializable correctness to the snapshot consistency model based database systems. They suggested methods to avoid read-write anomalies by automatically modifying the transaction code.

Genuine Multi-version Update-Serializable Partial Data Replication (GMU) [72] work published in 2012 provided high performance using the consistency criterion Extended Update Serializability (EUS). Update serializability is a weaker consistency criterion and can be defined as follows:

Definition 2.2.3 (**Update Serializability**). *A schedule s over a set T of transactions is update serializable (USR) iff each schedule s' obtained from s after deleting all but one read-only transaction is serializable. If there are no read-only transactions , then no transactions need be deleted [39].*

Extended Update Serializability (EUS) [40] allows concurrent read-only transactions to observe snapshots generated from different linear extensions of the history of update transactions.

At its heart GMU uses a distributed multiversion concurrency control scheme, a vector clock based synchronization algorithm, to track down data and causal dependency relations. It uses the partial replication to reduce the amount of network communication.

2.3 Replication

Replication has been heavily used by DTM community to extract the transaction localization and to boost performance. For replication-based transactional systems, consistency is generally defined using 1-copy serialization, which can be defined as described below:

Definition 2.3.1 (1-Copy Serializability). *A concurrent execution of transactions in a replicated database is 1-copy-serializable if it is equivalent to an ordering obtained when the transactions are performed sequentially in a single centralized database. [21]*

As described earlier, first paper in DTM by Manassiev [61] based on DMV algorithm, used data replication. For providing 1-copy serialization DMV used the global serialization time-stamp token, which proved costly approach to support 1-copy serialization in replication based systems. Later Dependable Distributed Software Transactional Memory (D2STM) system [26] and AGGRO scheme [70] improved performance by replacing global time-stamp with atomic broadcast messages for all object replicas.

Sinfonia [12] service also supported replication. However, it did not design the concurrency algorithm taking the replication in consideration. Instead, it used the primary-copy replication to recover in case of failure.

DecentSTM [16] used the full replication with random consensus concurrency protocol and provided the snapshot consistency. It showed improvement using replication and Snapshot consistency, but lacked any innovation in concurrency protocol. Later GMU [72] presented an innovative concurrency protocol based on partial replication supporting Extended updated serializability, and showed good performance improvement.

2.4 Strong Atomicity

We have already introduced the strong atomicity informally in chapter 1. Formally, strong atomicity can be described as below:

Definition 2.4.1 (Strong Atomicity). *Strong atomicity is a transactional semantics that guarantees the atomicity between transactional and non-transactional code.*

Cluster-STM [20] supported the strong atomicity by implementing a programming restriction. It asked programmer to access each memory location always within a transaction or outside transaction, but it did not provide any automatic support. DecentSTM [16] programming model guaranteed strong atomicity and helped programmers to stop careless atomicity mistakes. HyflowScala [88] and HyflowCPP also have en-build the strong atomicity using programming model.

2.5 Partial Abort Models

For performance improvement, nesting techniques are widely used in database systems. In 1981, Moss [65] first time described the nesting concept for distributed databases. He extended two-phase commit protocol [94] to support the nesting and proposed algorithms for distributed transaction management, object state restoration, and distributed deadlock detection. Later in 1983 Gracia [34] et. al. extensively analyzed it in open nesting context using undo-logs transactions.

Transaction nesting was first time introduced to Software Transactional Memory (STM) in 2006 by Moss and Hosking [66].They provided the semantics of transactional operations in terms of system state made of a tuple of a transaction ID, a memory location, a read/write flag, and the value read or written. Later Moss [64] further described the open nesting as method to overcome false conflicts and improve concurrency.

In same year Moravan et al. implemented nesting in logTM [62] and demonstrated the 100% speed-up for few benchmarks. In 2009, Agrawal et. al. [11] introduced the concept of transaction ownership by combining the closed and open nesting. Herlihy and Koskinen propose transactional boosting [43] for implementing highly concurrent transactional data structures, which internally implemented the open nesting. Later Koskinen and Herlihy [56] suggested the checkpointing as an alternative to nesting. Recently, HyflowJava based work on closed nesting [91] and open nesting [89] described the tradeoff of using nesting in DTM.

2.6 Unconventional Database Systems

In recent times, there has been lot of work on designing unconventional database system. With explosion in amount of data processed and stored in data warehouses, system administrators are understanding the limitation of current database systems. Many works [85, 41] have argued that current DBMS perform a poor job of CPU utilization and might require whole redesign. Works like HStore [51] and Google Spanner [25] have achieved high performance using new architectures based on replication and multi-versioning.

2.7 Summary

HyflowCPP framework is implemented in C++ at API-level and focuses on non-replicated peer-to-peer distributed systems. Current currency control algorithm, Transaction Forwarding Algorithm (TFA), provides the strong memory consistency. Using distributed clock, TFA is able to overcome any global serialization overhead. Single version objects without any replication helps TFA to reduce the amount of network messaging and enables it to scale without any network bottleneck.

Chapter 3

Programming Interface

In this chapter, we introduce the programming interface provided by HyflowCPP to execute the distributed atomic transactions. First, we describe the basics of interface, and then explain it by developing the List benchmark. HyflowCPP allows programmer to configure the benchmark and execution settings using a configuration file. Programmer can also set-up the configuration using environment variables.

In a distributed setting, objects are dispersed over different nodes, therefore normal object reference can not be used. Programmer is required to use a unique key to address a particular object anywhere in network. We provide a base class named HyflowObject for every distributed object. Every distributed object must inherit this class. HyflowObject provides the getId() method which returns as unique key to access the object from anywhere in the network.

HyflowCPP performs object serialization using boost serialization library. Programmer is required to follow it for proper packing and unpacking of object over the network. Distributed objects created by programmer are required to inherit HyflowObject class. Each inherited object field is registered with boost serialization function, so it can be serialized or deserialized over network.

For development of DTM applications HyflowCPP provides two transactional interfaces:

1. Transaction support using Macros
2. Transaction support using Atomic class

HyflowCPP atomic construct

```
HYFLOW_ATOMIC_START{
// Example of simple compare
// and swap operation
    value = Read(Address);
    if( value==myValue )
        Write(Address, myValue)
}HYFLOW_ATOMIC_END;
```

Standard STM atomic construct

```
atomic{
// Example of simple compare
// and swap operation
    value = Read(Address);
    if( value==myValue )
        Write(Address, myValue)
}
```

Figure 3.1: Atomic Construct for HyflowCPP vs. Standard STM implementations

3.1 Transaction Support using Macros

HyflowCPP supports standard atomic semantics using macros HYFLOW_ATOMIC_START and HYFLOW_ATOMIC_END. Figure 3.1 shows how to utilize these macros to execute any given part of code atomically, and compares it with standard STM atomic semantics.

Any object in the network can be opened in either *Read* or *Write* mode. Once programmer requests the object, HyflowCPP fetches the object from its current location and copies to transactions read or write set depending of access type. For accessing any object programmer should use macro HYFLOW_FETCH(ID, IS_READ). First argument to this macro is object ID and second argument is access type (true/false), true for read, otherwise false.

For reading or writing the fetched objects, we provide two macros HYFLOW_ON_READ(ID) and HYFLOW_ON_WRITE(ID). HYFLOW_ON_READ returns the programmer the constant reference pointer to HyflowObject, which can be used for read-only operations. For manipulating the object programmer is required to call HYFLOW_ON_WRITE, which returns a normal reference pointer to object. Programmer can also use the constant object reference pointer itself in place of unique object key to retrieve the object in write mode.

HYFLOW_PUBLISH_OBJECT(OBJ) allows programmer to publish any locally created object on the network. However, such objects must inherit the HyflowObject class as a base type as discussed earlier. Similarly, HYFLOW_PUBLISH_DELETE(OBJ) allows programmer to delete any object from the network.

Now using described macros, we write the List benchmark as described in figure 3.2. In this figure, we illustrate the list node addition and deletion atomically using HyflowCPP Macros.

HyflowCPP also supports the transaction checkpointing using macros. For minimum checkpointing overhead, we support it in outermost transaction call. We allow programmer to checkpoint at any place using HYFLOW_CHECKPOINT_HERE. Programmer is also required to initiate the checkpointing using HYFLOW_CHECKPOINT_INIT at start of transaction. To save any primary data structure object on stack or heap, macro HYFLOW_STORE is provided. For heap objects programmer is responsible for creating object copy and mem-

```
class ListNode::HyflowObject {

void ListNode::addNode(int value) {
  HYFLOW_ATOMIC_START{
     std::string head="HEAD";
     HYFLOW_FETCH(head, false);

     ListNode* headNodeRead =  (ListNode*)HYFLOW_ON_READ(head);
     std::string oldNext = headNodeRead->getNextId();
     ListNode* newNode = new ListNode(value, ListBenchmark::getId());
     newNode->setNextId(oldNext);
     HYFLOW_PUBLISH_OBJECT(newNode);

     ListNode* headNodeWrite = (ListNode*)HYFLOW_ON_WRITE(head);
     headNodeWrite->setNextId(newNode->getId());
  } HYFLOW_ATOMIC_END;
}

void ListNode::deleteNode(int value) {
  HYFLOW_ATOMIC_START{
     ListNode* targetNode = NULL;
     std::string head("HEAD");
     std::string prev = head, next;

     HYFLOW_FETCH(head, true);
     targetNode = (ListNode*)HYFLOW_ON_READ(head);
     next = targetNode->getNextId();

     while(next.compare("NULL") != 0) {
       HYFLOW_FETCH(next, true);
       targetNode = (ListNode*)HYFLOW_ON_READ(next);
       int nodeValue = targetNode->getValue();
       if (nodeValue == value) {
         ListNode* prevNode = (ListNode*)HYFLOW_ON_WRITE(prev);
         ListNode* currentNode = (ListNode*)HYFLOW_ON_WRITE(next);
         prevNode->setNextId(currentNode->getNextId());
         HYFLOW_DELETE_OBJECT(currentNode);
         break;
       }
       prev = next;
       next = targetNode->getNextId();
     }
  } HYFLOW_ATOMIC_END;
}

}
```

Figure 3.2: List benchmark using HyflowCPP Macros

```
void BankAccount::transfer(string Account1, string Account2, Money)
    {
    HYFLOW_ATOMIC_START {
        HYFLOW_CHECKPOINT_INIT;

        withdraw(Account1, Money, __context__);

        HYFLOW_CHECKPOINT_HERE;

        deposit(Account2, Money, __context__);
    }HYFLOW_ATOMIC_END;
}
```

Figure 3.3: Checkpointing in Bank transfer function

ory management, HYFLOW_STORE macro only save the address value for programmer. HYFLOW_STORE(VAR_REF, VAR_VALUE) macro requires two arguments, variable reference, and variable value. This macro automatically restores the saved variable value on transaction resume from selected checkpoint.

Figure 3.3 illustrates how checkpointing can be implemented in a simple bank transfer function. Note that how programmer is required to pass the current context instance to withdraw function. This requirement exists for all the functions, which are called within atomic block and are required to be executed atomically. This additional argument passing will be removed once the compiler support is added to atomic block.

3.2 Transaction Support using Atomic class

Atomic class interface is more evolved and allows programmers to directly code against the HyflowCPP framework. Using the Atomic class interface programmer can directly manipulate transaction context and execute any function atomically. Programmer can assign 'atomically' function pointer of Atomic class instance a desired function pointer value. Later, 'execute' method in Atomic class instance can be called to execute desired function atomically. 'atomically' function pointer requires a strict argument set to support the atomic execution of functions. Programmer is required to use inherited classes of BenchMarkArgs and BenchMarkReturn to pass arguments to atomic function and receive the return values. Two additional arguments HyflowObject and HyflowContext are passed to function for internal DTM book keeping.

Atomic class also provides the function pointers to support advance nesting features like open-nesting. Programmer can specify the *onCommit* and *onAbort* functions for HyflowObject requiring the open nesting support. All these function pointers requires to follow a defined argument set similar to 'atomically' function pointer.

```
class ListNode::HyflowObject {

void deleteNodeAtomically (HyflowObject* self, BenchMarkArgs* args,
 HyflowContext* __context__, BenchMarkReturn* success)
{
    int value = ((ListArgs*)args)->value;
    ListNode* targetNode = NULL;
    std::string head("HEAD");
    std::string prev = head, next;

    HYFLOW_FETCH(head, true);

    targetNode = (ListNode*)HYFLOW_ON_READ(head);
    next = targetNode->getNextId();

    while(next.compare("NULL") != 0) {
        HYFLOW_FETCH(next, true);

        targetNode = (ListNode*)HYFLOW_ON_READ(next);

        int nodeValue = targetNode->getValue();
        if (nodeValue == value) {

            ListNode* prevNode = (ListNode*)HYFLOW_ON_WRITE(prev);

            ListNode* currentNode = (ListNode*)HYFLOW_ON_WRITE(next);

            prevNode->setNextId(currentNode->getNextId());

            HYFLOW_DELETE_OBJECT(currentNode);

            break;
        }
        prev = next;
        next = targetNode->getNextId();
    }
}

void ListNode::deleteNode(int value) {
    Atomic atomicDelete;
    ListArgs args(value);

    atomicDelete.atomically = ListNode::deleteNodeAtomically;
    atomicDelete.execute(NULL, &args, NULL);
}

}
```

Figure 3.4: List benchmark using HyflowCPP Atomic Class

In figure 3.4, we rewrite the same List benchmark functions using the Atomic class. 'deleteNode' function illustrates how to create Atomic class instance and execute a function atomically. In 'deleteNodeAtomically' function, we demonstrate how using different macros distributed objects can be manipulated.

Only limitation of Atomic class is that, they do not supported transaction checkpointing. Checkpoints are required to be created in outermost transactions and Atomic class executes the function pointer as an inner transaction, which limits the checkpointing support in Atomic class.

Programmer can also write the benchmark directly without using any of Atomic class or Macros. This will require the programmer to directory manipulate the DirectoryManager for remote object access and ContextManager for transaction atomicity. It can be some time useful for debugging some internal framework issues. Otherwise, it is not a recommended method.

Chapter 4

System Architecture

HyflowCPP is a distributed software transaction memory system at API level. It is implemented in C++ using object oriented programming paradigm. It is designed in a modular manner to provide pluggable support for different concurrency control algorithms, directory lookup protocols, contention management policies, and network communication protocols. All components are developed independent of each other and communicate through well-defined interfaces. Any individual component can be easily replaced or modified by writing an implementation compliant to the given component interface.

Figure 4.1 shows the architecture of a transactional node in HyflowCPP. It is composed of six modules: Transaction Interface Module, Transaction Validation Module, Object Access Module, Object serialization Module, Network Module, and Message Processing Module. We have already discussed about Transaction Interface Module in detail in chapter 3. Now we will describe rest of modules in detail in following sections.

4.1 Transaction Validation Module

The goal of Transaction Validation Module is to provide transactional consistency and achieve system wide progress. All the concurrency control logic is performed in this module by extending base class HyflowContext. Currently by default, HyflowContext is extended as DTLContext which implements the TFA algorithm, which we discuss in detail in chapter 5. This module validates memory locations and retries the transaction on commit failure. This module can be configured based on the transaction model used like checkpointing, closed nesting, and open nesting.

To support the DTM protocols this module also provides the *lock-table* for object level or word level locking. This table is implemented using high performance concurrent hash-maps of Thread Building Block(TBB) [97] library. This module also provides the *context-map*

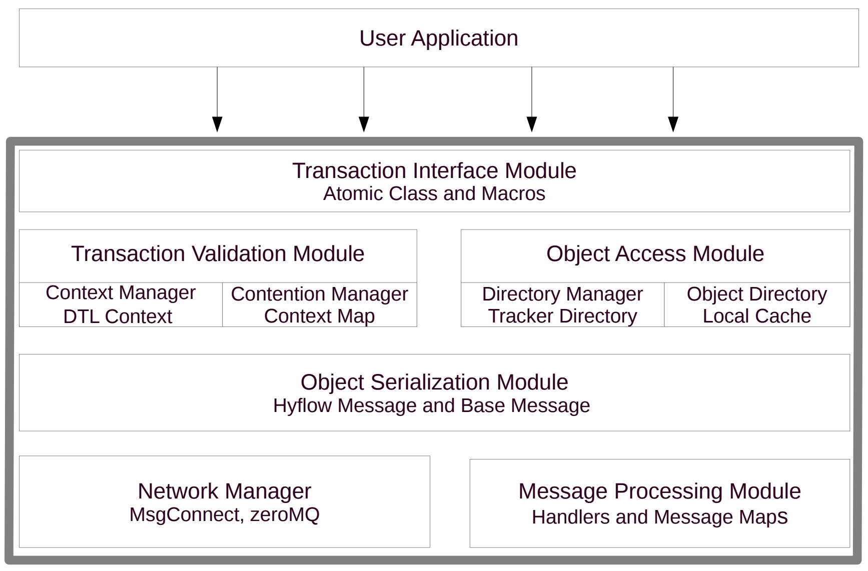


Figure 4.1: HyflowCPP Architecture

to access the context instance using transaction Id. It enables the contention managers to collect meta-data and make updates to different transactional contexts.

This module interfaces with transaction interface module to create transactional contexts for user applications and to provide access to distributed objects through object access module. It also uses the message interface to perform context specific messaging for commit and validation requests with remote nodes.

4.2 Object Access Module

Object Access Module serves multiple purposes in HyflowCPP framework. It provides the copy of distributed objects, object owner information, and performs object version validation and object directory updates. Objects are located using its unique object Id. This module encapsulates a directory lookup protocol to access distributed objects. Currently, by default it implements the efficient tracker directory cache coherence protocol. Tracker directory moves the object across nodes and maintains the current owner information on specific tracker node. To access any object, this module first finds out the current owner information using object directory, then it sends the object request to owner node. Owner node replies current copy of object or a null value, in case object was deleted by some other transaction.

Tracker directory updates the owner information in tracker node object directory as object moves from one node to another node. Also on object creation, or deletion object access module updates the object directory with owner information.

Similar to transaction validation module, object access module contains two object maps to support any object access protocol. It provides the *local cache* and *object directory*. The purpose of local cache is to maintain the authoritative copy of objects owned by current node. Meanwhile, object directory is utilized by nodes to keep the meta-data information, like in case of Tracker Directory, the object owner information.

Object Access Module interfaces with two other modules: *object validation module* and *message interface module*. To provide *strong atomicity* HyflowCPP directs all the accesses to objects through transaction context. Therefore, all object requests to object access module go through Transaction Validation Module. Object deletion or publication requests made by transaction validation module are served in object access module. In addition to that, Object Access Module handles the object validation request. For serialization and de-serialization of all remote objects, object access module interacts with the object serialization module.

4.3 Object Serialization Module

Message serialization and de-serialization is a big challenge in distributed computing using C++. To free user from this cumbersome process, HyflowCPP provides a messaging interface and allows developer to add pluggable validation and distribution protocols without worrying about serialization and de-serialization of messages. HyflowCPP provides HyflowMessage and BaseMessage classes for this purpose. BaseMessage acts as parent class for any message in HyflowCPP. Framework developer can create any new type of message to perform any protocol specific task just by extending BaseMessage. HyflowMessage class acts as wrapper class for BaseMessage and all its extensions. HyflowMessage contains the information about the BaseMessage encapsulated in it to provide required information for proper serialization and de-serialize.

HyflowMessage provides a standard interface to network library. All the HyflowMessage instances are converted in a binary blob, before forwarding to network library, to communicate over network. In this way, network implementation is totally independent of *transaction validation module* and *object access module*. For serialization and de-serialization of HyflowMessage, we use the *Boost serialization* [52]. It provides the support to serialize any type of complex message or object.

Object serialization module is central part of HyflowCPP framework. It is accessed by all other modules except transactional interface module. Object validation module and object access module use this interface to send transactional messages and object requests, while network manager and message handling interface utilize it to process incoming message and provide responses to upper layer modules.

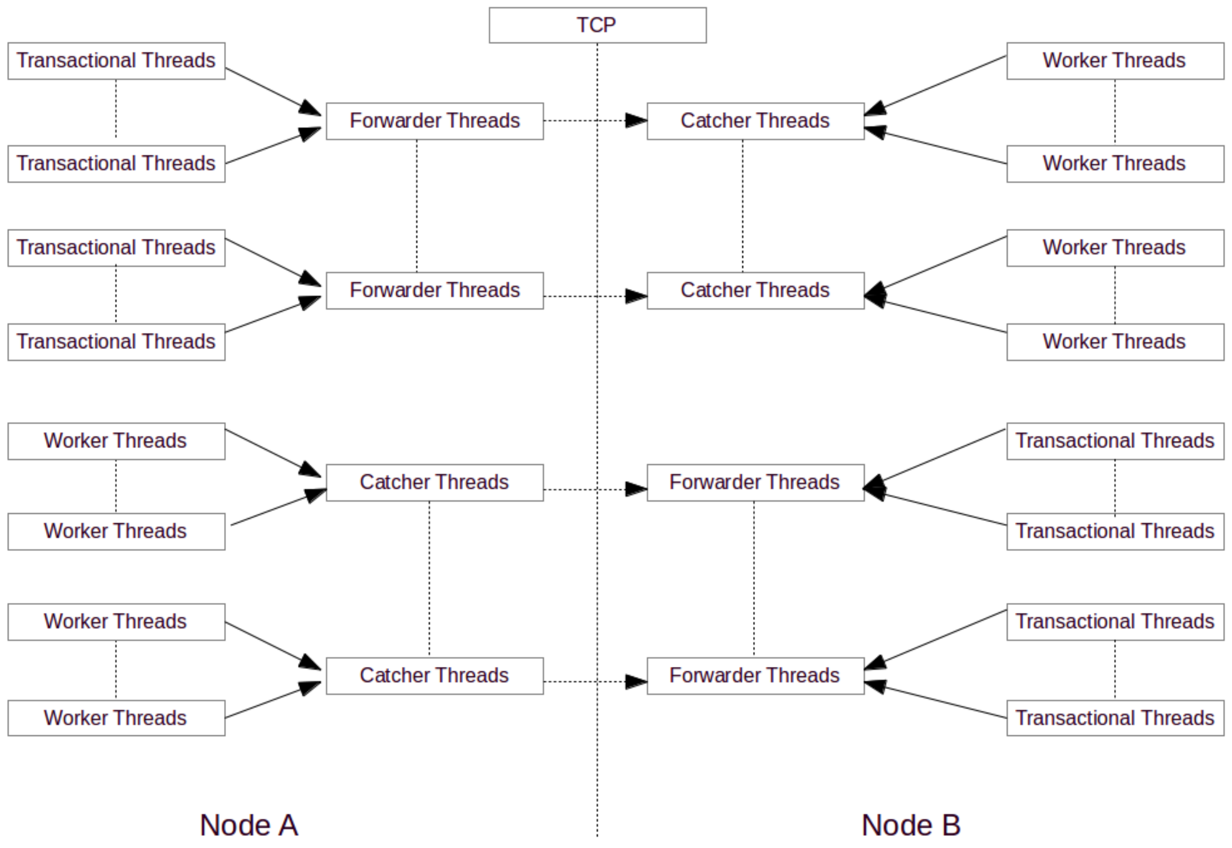


Figure 4.2: ZeroMQ Network Architecture

4.4 Network Module

Network module plays a very vital role in performance of a DTM framework. A poor implementation and bad scalability of network module can lead to loss in performance even for an efficient DTM algorithm. In HyflowCPP, we provide pluggable support for any network library through a well-defined network interface. Currently in HyflowCPP, we support two networking libraries: MsgConnect [67] and ZeroMQ [48].

MsgConnect is an open source library for Linux platforms. It provides a high-level messaging interface for programmers. It frees programmers from socket level message handling and provides a reliable way of communication using TCP protocol [33]. Unfortunately, we found some scalability issues in open source implementation. Still, for basic prototyping it can be a reliable networking library.

To design a fast and scalable networking solution, we use industry standard ZeroMQ library. ZeroMQ is socket level library, but provides very efficient solutions for in-process communication between threads, which makes it suitable for any multi-threaded networking requirement. In figure 4.2, we describe our networking architecture designed using ZeroMQ socket library. This architecture design allows configuring the networking threads to fine tune for desired workload.

As described in figure 4.2, transactional nodes A and B communicate with each other using *forwarder* and *catcher* networking threads. These dedicated threads are responsible for connecting with different nodes using ZeroMQ router socket. ZeroMQ router sockets are very useful to communicate with multiple nodes simultaneously. Forwarder threads are responsible for receiving message request from transactional threads and forward it to catcher thread of desired nodes. On other side, catcher threads are responsible for receiving message workload from forwarder thread and assign it to available *worker* thread. Worker threads process the message, and send back reply to catcher thread if required. Catcher thread in turn returns message back to forwarder thread, which conveys it to original requester transactional thread.

ZeroMQ provides up to 5 times better performance in comparison to MsgConnect. Using multi-part messaging of ZeroMQ, we are able to reduce the amount of polling between threads to bare minimum at two sockets. With experiments we found that, one forwarder is enough for six transactional threads. In contrast, even one worker per catcher can be sufficient for message processing in some situations. To fine-tune these values, we define two ratios zeroMQTFR (i. e., zeroMQ transactional thread to forwarder thread ratio) and zeroMQWFR (i. e., ZeroMQ worker thread to forwarder thread ratio). It is worth noting that forwarder to catcher ratio is always one, therefore zeroMQWFR also defines the Catcher threads to worker thread ratio. By fine tuning these values user can extract a very high messaging throughput.

4.5 Message Processing Module

Message Processing Module provides message-handling capability to any network library. The major task to this interface is to find a proper handler for any message and support asynchronous messaging. When a framework developer creates a new type of message, he/she also creates a proper message handler of this message type. All message handlers are registered by Message handler interface, at network initiation time. Later, when network library receives a request message, using the message handler, it creates a proper response, and replies back as required.

Another important task performed by this module is to support the asynchronous messaging. This module defines a class HyflowMessageFuture, which allows a transactional thread to send a message asynchronously. A transactional thread can send a message with a HyflowMessageFuture object and proceed with other tasks. Later, transactional thread can come back to message specific HyflowMessageFuture and wait for message response.

This module directly interfaces with network module and object serialization module. All messages and their handling are defined in Object Serialization Module. Message and their handler mapping is defined in the Message Processing Module. Network module utilizes this module to find the appropriate handler and provide the asynchronous message response notification to requesting transactional threads.

Chapter 5

Transaction Forwarding Algorithm

In this chapter, we describe Transaction Forwarding Algorithm (TFA) in detail for different transactional models. TFA is a lock-based algorithm with lazy acquisition scheme. It uses an innovative asynchronous distributed clock mechanism to validate transactions. TFA supports opacity property [38] and guarantees strong progress [38]. Opacity is a stronger property in comparison to serialization and informally can be described as an extension of the classical database serializability property with the additional requirement that even non-committed transactions are prevented from accessing inconsistent states. Strong progressiveness promises that all non-conflicting transactions will commit and at least one of all conflicting transactions will commit.

Now we present the TFA algorithm for different transactional models and define its main procedures for transaction validation.

5.1 Flat Nesting

TFA uses a synchronization variant similar to Lamport [58] mechanism to keep the clocks synchronized. In TFA, each distributed node has a local clock *lc*. Each node increases its local clock atomically on a write transaction commit. All the communication between nodes piggyback the sender node's local clock. On receiving the messages of remote node, each node compares the piggybacked remote node clock with its local clock. A node updates its local clock to remote node clock if it is higher than node's local clock. Otherwise, node ignores it. In this way, all the nodes are kept in synchronization and are able to establish the happens before relationship between object reads.

Asynchronous distributed clocking enables the TFA to detect any early conflicts in the transaction. Early conflicts in the transaction are detected through a process called Transaction Forwarding, which is performed at the time of object access (read/write).

```
Context::TransactionForwarding(senderNodeClock) {
    if senderNodeClock > transaction.timeStamp {
        forAll obj in readSet {
            if (currentVersion(obj)>transaction.timeStamp) {
                transaction.rollback();
                return ;
            }
        }
        transaction.timeStamp = senderNodeClock ;
    }
}
```

Figure 5.1: Transaction Forwarding in Flat Nesting

```
Context::Commit() {
    forAll obj in writeSet {
        lock = obj.acquireLock()
        if !lock
            rollback
    }

    forAll obj in readSet {
        valid = obj.readValidate();
        if !valid
            rollback
    }

    if writeTransaction
        nodeClock++;

    commitWriteSet();

    forAll obj in transaction.writeSet {
        obj.commitValue()
        obj.setVersion(transaction.timeStamp)
        obj.releaseLock()
        if obj.remote {
            updateOwner(obj)
        }
    }

    forAll obj in transaction.publishSet
        publish(obj)

    forAll obj in transaction.deleteSet
        delete(obj)
}
```

Figure 5.2: Commit in Flat Nesting

On start of any transaction, node's local clock is attached to its context c as a time stamp wv. When any object is accessed by a transaction, the sender nodes clock rc is compared against the transactions time stamp wv. If sender nodes clock rc is greater than the transactions time stamp wv, we verify whether we can move the transaction time stamp wv to wv', where $wv' = rc$. This validation is done by validating all read-set objects version against transaction time stamp wv, if they are still less than wv, we can safely forward transaction from wv to wv'. Figure 5.1 illustrates the Transaction Forwarding procedure.

In figure 5.2, we illustrate the transaction commit process. Transaction commit in TFA is performed similar to two-phase commit [94] protocol. At commit time, the transactional node tries to acquire locks on the write-set objects in appropriate order to avoid deadlocks. Remote object lock requests are sent to object owner. If any of locks can not be acquired, transaction rollbacks. If transaction succeeds in acquiring the object locks, it tries to re-validate the read-set objects. Again, if read-set object validation fails the transaction performs rollback. Once write-set locks are acquired and read-set objects are validated, it is safe to commit the transaction. In commit process for write transactions, local clock is increased atomically. All write-set object versions are updated to transaction version. For local objects, the updated copies of objects are committed and for the remote objects, the change of object ownership is performed. Publish-set and delete-set object entries are updated in object access module. After completion of commit process, all the write-set object locks are released.

5.2 Closed Nesting

In closed nesting, each transaction attempts to do individual commits, but for inner transactions the commits are not visible outside of the enclosing transaction. Inner transactions are allowed to abort independently of their parents. In this way, by permitting partial aborts for the inner transactions, closed nesting helps to improve performance. In this section, we present the modified version of TFA for closed nesting: *Transaction Forwarding Algorithm for closed nesting (TFACN).*

To support closed nesting, each context maintains a reference to the parent context. For outermost context, the parent context reference is set to *null*. In transaction forwarding step, we perform transaction forwarding on whole context tree branch rather than looking only at the current inner transaction. In figure 5.3, we illustrate the transaction forwarding step for TFACN.

In commit phase, TFACN directly merges the context objects to parent context objects for inner transactions. For outermost transaction commit procedure is same as in flat nesting as illustrated in figure 5.2.

```
Context::TransactionForwarding(senderNodeClock) {
    if senderNodeClock > transaction.timeStamp {
        contextList = context.fetchContextBranch()
        forAll context in contextList {
            forAll obj in context.readSet {
                if (currentVersion(obj)>transaction.timeStamp) {
                    transaction.rollback();
                    return ;
                }
            }
        }
        transaction.timeStamp = senderNodeClock ;
    }
}
```

Figure 5.3: Transaction Forwarding in Closed Nesting

5.3 Checkpointing

Transaction checkpointing saves the transaction execution state at various points. Later on commit failure, it enables a transaction to resume from previously saved valid checkpoint. Checkpointing allows a transaction to abort only required part of transaction for which actually the conflict happens and therefore boosts the performance. Checkpointing can be considered as an extension of closed nesting, where partial aborts can be applied at any point in transaction execution state. In this section, we present the modified the TFA algorithm for checkpointing: *Transaction Forwarding Algorithm with checkpointing (TFACP).*

To support checkpointing in TFACP we partition read-set, write-set, publish-set and delete-set objects on basis of first access checkpoint. Partitioning allows TFACP to identify the conflicting objects, and to calculate dependency for determining the valid checkpoint. To save the transaction execution stack state we use the *setContext()* and *getContext()* functions. Heap objects are maintained in context read-set, write-set and publish-set.

HyflowCPP provides a helper class *CheckPointProvider* to create, iterator, and maintain the transaction checkpoints. *HYFLOW_STORE(OBJECT_REF, OBJECT_VALUE)* macro is provided to store any stack or heap object values just before creating checkpoint. These values are automatically are restored on continuing from checkpoint after partial abort.

Figure 5.4 illustrates the commit procedure in TFACP. Commit process in TFACP is also modified similar to transaction forwarding. On commit failure instead of restarting transaction, we resume from available valid checkpoint. Note that here we are required to release any acquired lock before resuming.

```
commit::tryResume() {
    if transaction.checkPointAvailable {
        removeInvalidatedObjects();
        releaselocks();
        transaction.resume();
    }else {
        transaction.rollback();
        return ;
    }
}

Context::Commit() {
    forAll obj in writeSet {
        lock = obj.acquireLock()
        if !lock
            tryResume()
    }

    forAll obj in readSet {
        valid = obj.readValidate()
        if !valid
            tryResume()
    }
    if writeTransaction
        nodeClock++

    commitWriteSet()

    forAll obj in transaction.writeSet
        obj.commitValue()
        obj.setVersion(transaction.timeStamp)
        obj.releaseLock()
        if obj.remote then
            updateOwner(obj)

    forAll obj in transaction.publishSet
        publish(obj)

    forAll obj in transaction.deleteSet
        delete(obj)
}
```

Figure 5.4: Commit in Checkpointing

In figure 5.5, we illustrate the transaction forwarding process for TFACP. Similar to flat nesting we perform transaction forwarding by object validation. However, on validation failure, we do not rollback instead resume the transaction from valid checkpoint if available. Before resuming, we remove all the invalidated heap objects from the context.

```
Context::TransactionForwarding(senderNodeClock) {
    if senderNodeClock > transaction.timeStamp {
        forAll obj in readSet {
            if (currentVersion(obj)>transaction.timeStamp) {
                if transaction.checkPointAvailable {
                    removeInvalidatedObjects();
                    transaction.resume();
                }else {
                    transaction.rollback();
                    return ;
                }
            }
        }
        transaction.timeStamp = senderNodeClock ;
    }
}
```

Figure 5.5: Transaction Forwarding in Checkpointing

5.4 Open Nesting

Open nesting provides the performance improvement by reducing the false memory level conflict. In contrast to closed nesting in open nesting the inner transactions commit on completion are visible globally and corresponding undo action is merged into parent transaction. In case of parent transaction abort the undo action are used to recover from the inner transaction commit. To maintain the memory consistency the inner transaction specific higher level locks are maintained.

To support open nesting in TFA (TFAON), a concept of abstract locks is introduced, which provides the higher level locks for inner transactions. Unfortunately, abstract lock mechanism suffers from the livelocks issue and uses a random back-off mechanism to overcome this issue. TFAON allows programmer to define the *onAbort* and *onCommit* functions to be performed in case of abort or commit on parent transaction. TFAON acquires abstract locks in inner transactions and releases them in parent transaction on commit or abort.

As described in figure 5.7 abstract locks can be created by provide unique string name for transaction. Abstract locks are registered in transaction using the *onLockAcess* function. In figure 5.6, we illustrate the open nesting commit process. At commit time, the registered abstract locks are acquired. Later, abstract locks are either merged to parent context or released (for outermost transaction). In case of aborts, rollback is performed by using *onAbort* function. Once compensating action for related inner transaction is completed, abstract locks are released. Compensation actions run with higher priority by not performing random back- off.

```
Context::Commit() {
    forAll obj in writeSet {
        lock = obj.acquireLock()
        if !lock
            rollbackInnerTxn()
    }

    forAll obj in readSet {
        valid = obj.readValidate()
        if !valid
            rollbockInnerTxn()
    }

    for abstractLock in abstractLockSet {
        lock = abstractLock.acquireLock()
        if !lock
            rollback();
    }

    if writeTransaction
        nodeClock++

    commitWriteSet()

    forAll obj in transaction.writeSet {
        obj.commitValue()
        obj.setVersion(transaction.timeStamp)
        obj.releaseLock()
        if obj.remote {
            updateOwner(obj)
        }
    }

    forAll obj in transaction.publishSet
        publish(obj)

    forAll obj in transaction.deleteSet
        delete(obj)

    if topTransaction {
        releaseAbstractLocks()
    }else {
        mergeAbstractLockToParent()
    }
}
```

Figure 5.6: Commit in Open Nesting

```
Benchmark::transaction() {
    Atomic atomicTxn;
    TxnArgs txnArgs;

    atomicTxn.atomic = Benchmark::transactionAtomic;
    atomicTxn.onAbort = Benchmark::onAbortTxn;
    atomicTxn.execute(NULL, &txnArgs, NULL);
}

Benchmark::transactionAtomic(HyflowObject* self,
    BenchMarkArgs* args, HyflowContext* __context__,
    BenchMarkReturn* success) {
    string txnLock = txnSpecficLockName();

    readOperations();
    writeOperations();

    if (nesting == HYFLOW_OPEN_NESTING ) {
        string txnlock;
        __context__.onLockAccess(Benchmark, txnLock, false);
    }
}
```

Figure 5.7: Abstract lock acquisition in Open Nesting

Chapter 6

Experimental Results & Evaluation

In this chapter, the performance of HyflowCPP is compared against the other DTM frameworks using micro- and macro-benchmarks. Being the first ever DTM framework implemented in C++, we compare this work with other JVM based competitors like HyflowScala, HyflowJava, GenRSTM and DecentSTM. We evaluate the performance on multiple micro benchmarks including Bank, List, Binary Search Tree, Skip-list, Hash-table and macro benchmarks including Loan, Vacation, and TPCC.

6.1 Test Environment

Our all experiments are conducted on a 48 core machine which consist of four AMD Opteron^{TM} processors (6164 HE), each with 12 cores running at 1700 MHz, and 16 GB of memory. All experiment are conducted in Linux based environment using Ubuntu Linux Server 10.04 LTS 64-bit. We have treated each core on machine as a distributed node where each core communicates with other core through a network loop-back TCP connection, emulating the behavior of 48 distributed nodes. All process instances of benchmarks are bound to specific core to stop unnecessary context-switch between different cores. We use the following framework dependency libraries: Boost version 1.40, ZeroMQ version 1.30, and Intel Thread Build Block version 4.0.

6.2 Flat Nesting

Flat nesting does not provide any performance incentive over other partial abort models like closed or open nesting. Still, it is very useful to understand benchmark's characteristics and compare the raw performance between different DTM frameworks. We have evaluated the performance with different read ratios (0.2 , 0.5 and 0.8) for $4, 8, 16, 32, 48$ nodes. For concentrating on the distributed behavior in most of our experiments, we have limited the number of transactional threads on each node to ’1’. In order to understand the impact of network latency and CPU utilization, we have added up to 24 transactional threads per core for benchmarks like Bank and Hash-table. The concurrent nature of these benchmarks makes it easier to analyze the impact of more transactional threads per node on network latency, and on overall throughput. We perform these network latency and CPU utilization specific comparisons only with HyflowScala, because networking implementations of other competitors are very rudimentary. Other implementations lack proper workload balancing, synchronization and packet handling, which leads to higher queuing delays, response time and multiple TCP connection re-initiations between nodes.

For performance comparison, we compare the HyflowCPP with various competitors like HyflowJava [78], HyflowScala [88], DecentSTM [16] and GenRSTM [23]. We have compared the all the benchmarks already available with the competitors. For some HyflowCPP benchmarks, we have not found any implementation in competitors. We have not spent time in developing benchmarks in competitor frameworks. Instead, we have devoted our time in more important research work.

6.2.1 Micro Benchmarks

Micro benchmarks that we considered for performance evaluation are Bank, Linked-list, Binary Search Tree, Skip-list, and Hash-table. In each data structure, we convert all coarse grain lock accesses of lock based implementation with HYFLOW_ATOMIC_START and HYFLOW_ATOMIC_END transactional block. For every flat nesting experiment, we executed 2000 transactions for 3 times, and calculated transactional throughput of experiments by taking average of those 3 runs.

6.2.1.1 Bank

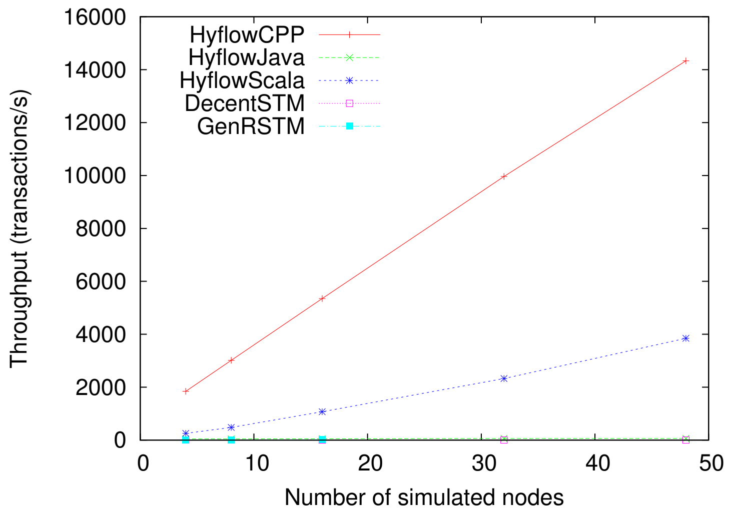
The Bank benchmark simulates a distributed banking system. Each distributed node is initiated with certain number of bank accounts. Each account is created with some initial amount. The Bank benchmark supports two operations: total balance and transfer. At the end of experiment a sanity check is performed. The point to be noted here is that the Bank benchmark is distributed in nature and it can perform multiple Transfer operations in parallel on different accounts.

In this experiment, we create 10000 accounts distributed over different nodes. In figure 6.1, we can observe the performance of the HyflowCPP in comparison to various competitors for different read percent: 20%, 50% and 80%. Bank being distributed in nature allows HyflowCPP performance to linearly scale with increase in the numbers of nodes. As we create 10000 accounts over the network, the increase in number of nodes does not increase the contention.

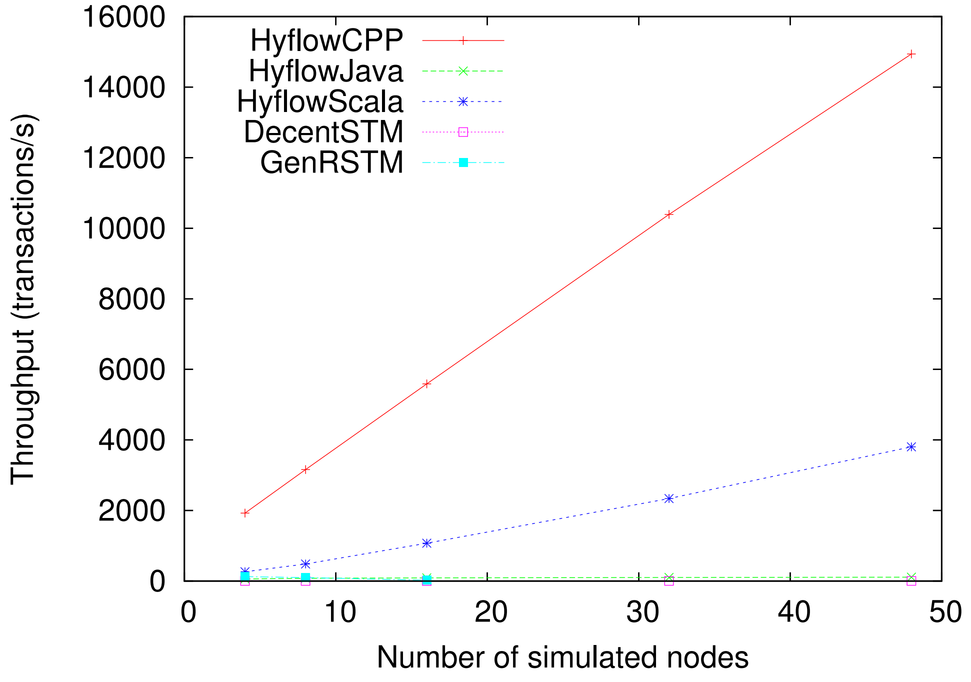
HyflowCPP performs up to 3 to 5 times better in comparison to its nearest competitor HyflowScala for single transactional thread per node setup. This high throughput gain can be attributed to lower network latency in HyflowCPP. For single transactional thread per core setup, the effect of network latency dominates over throughput. Once a transactional thread sends a remote message, for instance to access a distributed object, it waits for a reply message and does not use any CPU time. In case of higher network latency, a transactional thread will wait for longer time and lesser CPU time will be used. For single thread per core experiments, we observed that the average CPU time utilization for HyflowScala was around 20%-30%, whereas HyflowCPP utilized the 50%-60% of CPU time, which affirms our argument regarding lower CPU utilization due to the higher network latency.

To understand the impact of network latency and CPU utilization further, we increased the number of thread per node in HyflowCPP, and HyflowScala to the extent to which CPU utilization increases more than 98%. In such cases the effect of network latency is minimum on throughput, as the available CPU time due to wait of any network message can be utilized by some other transactional thread. In figure 6.1(d), we compare the performance of HyflowCPP with HyflowScala for up to 24 threads per node. We observe gains of only 2 to 4 times, because the impact of network latency reduces with increase in CPU utilization of HyflowScala.

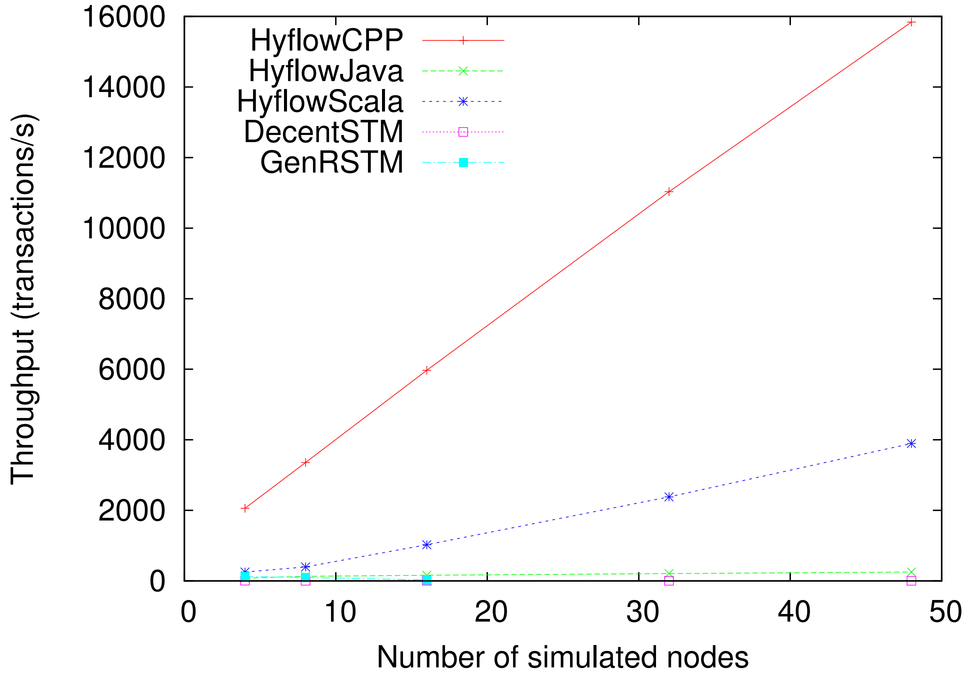
Poor performance of other implementations like HyflowJava, GenRSTM, and DecentSTM is rooted in their bad implementation. The HyflowJava and HyflowCPP use the same TFA algorithm for concurrency control. However, networking support in HyflowJava is very rudimentary and one of the major reason for bad performance. Similarly, the DecentSTM and GenRSTM implementations are also not optimized for network messaging. From algorithmic point of view, DecentSTM and GenRSTM should be able to perform better in case of high reads as replication allows to access objects locally.



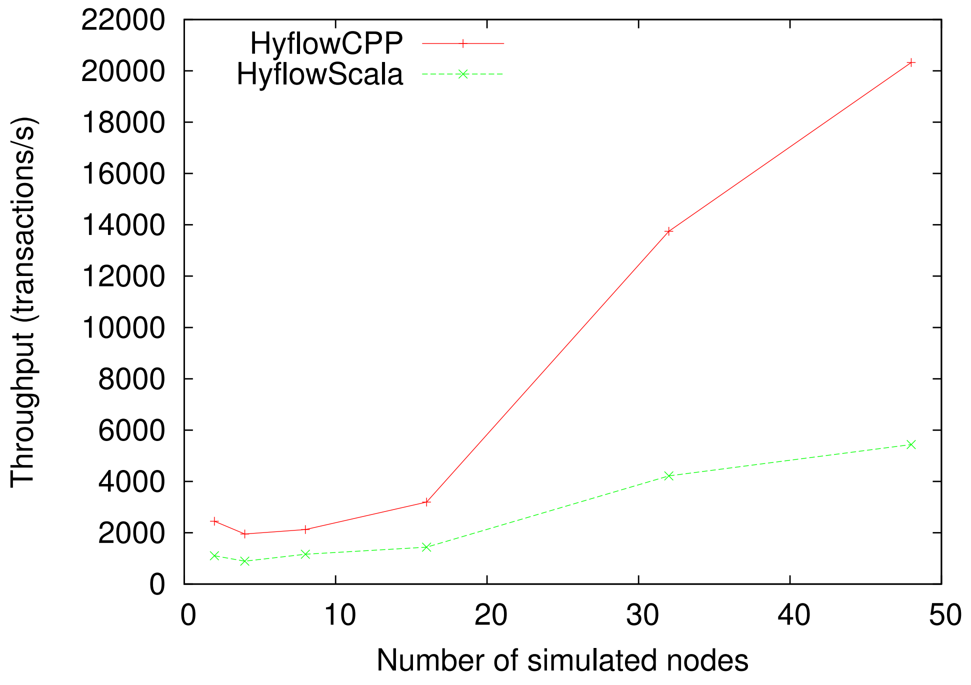
(a) Bank 20% reads



(b) Bank 50% reads



(c) Bank 80% reads



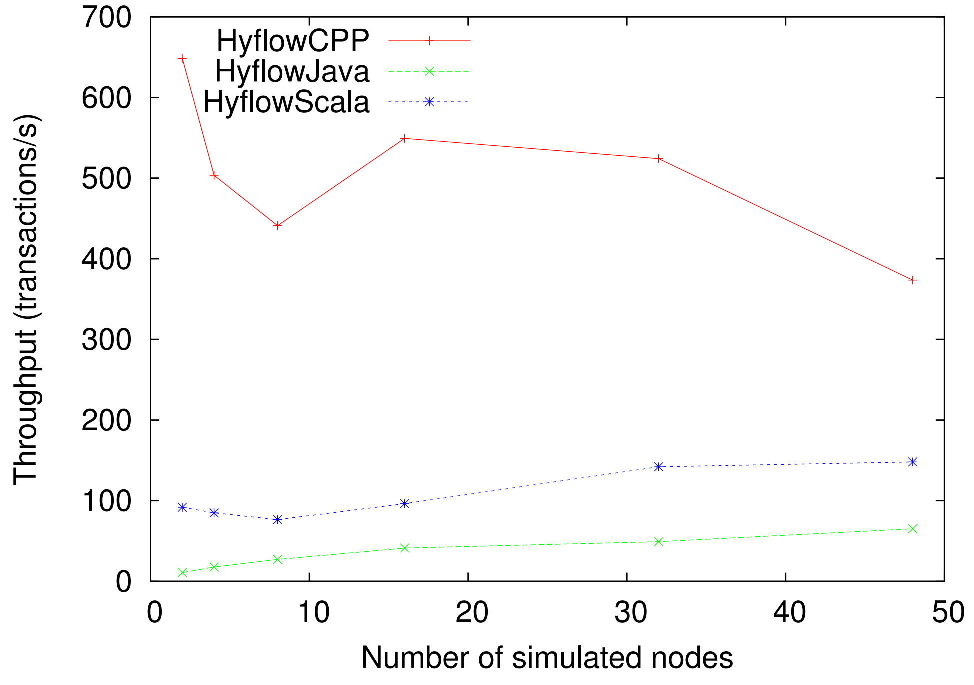
(d) Bank: HyflowCPP comparison with HyflowScala for up to 24 thread per node

Figure 6.1: Flat Nesting: Transactional throughput for Bank

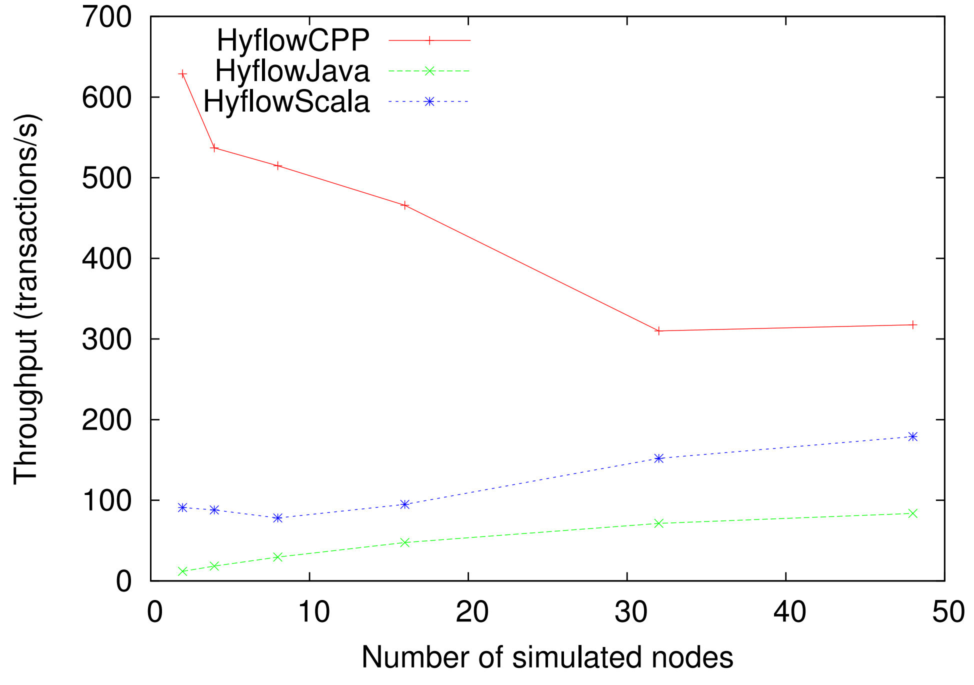
6.2.1.2 Linked List

In this benchmark, we have designed an ordered, singly linked linear list data structure. We implement the list as set and do not allow the duplicate objects in it, which helps to control the maximum number of objects in the list. It supports two write operations: add and remove, and one read operation: contains. While traversing the list, all the read objects are added to transaction read-set, which makes list prone to high contention. Any write operation on a traversed node will lead to transaction abort.

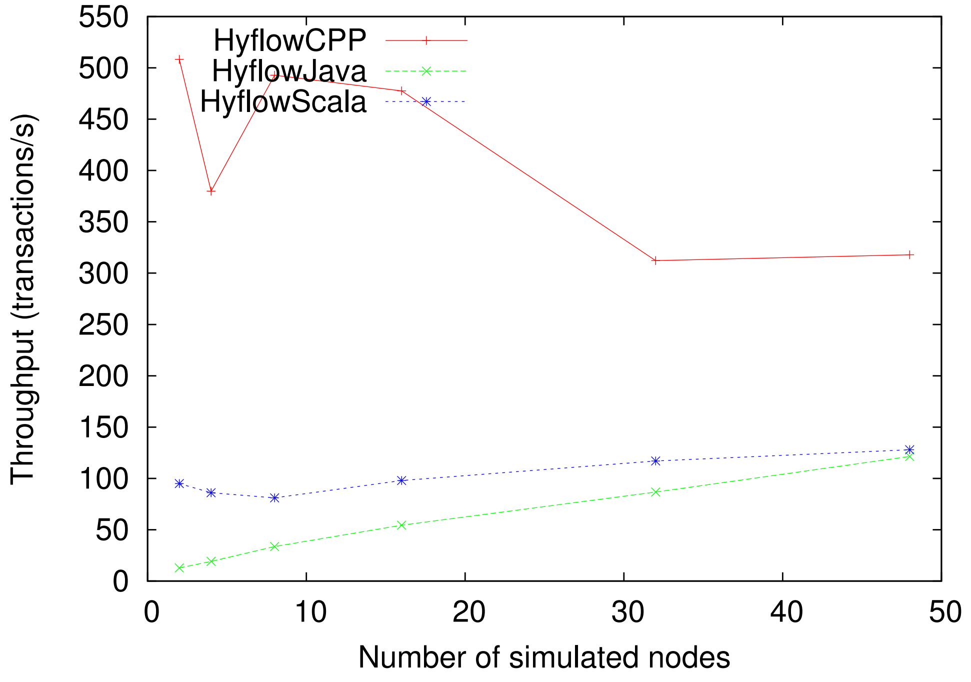
For our experiments, we have allowed up to 50 objects in linked list, and have warmed up benchmark by adding 50% objects before starting the experiments. In figure 6.2, we observe that HyflowCPP performance is up to 3 to 5 times better in comparison to nearest competitor HyflowScala. HyflowJava performs even worse than HyflowScala. We also find that transactional throughput increases with increase in read percent. We are unable to find any consistent pattern in throughput with increase in number of nodes. This is because of fact that contention is very high in list and aborts happen in random order. On an average, the list throughput decreases as number of nodes are increased, due to higher contention. Throughput value for list is lower is comparison to the benchmarks like Bank and Hash-table due to high abort rate and large read-set size.



(a) Linked list 20% reads



(b) Linked list 50% reads



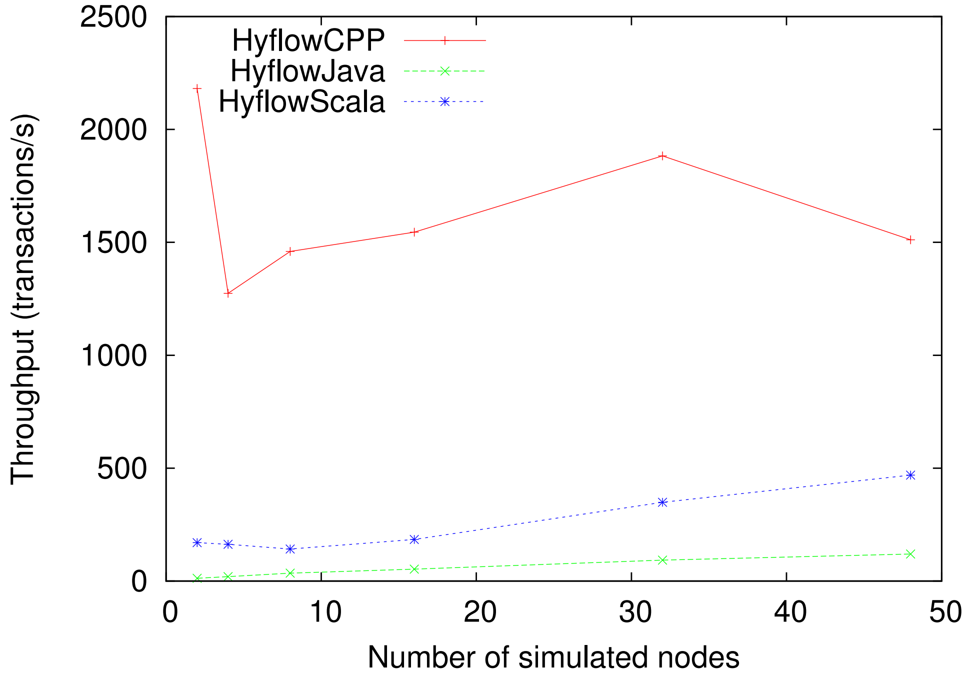
(c) Linked list 80% reads

Figure 6.2: Flat Nesting: Transactional throughput for Linked list

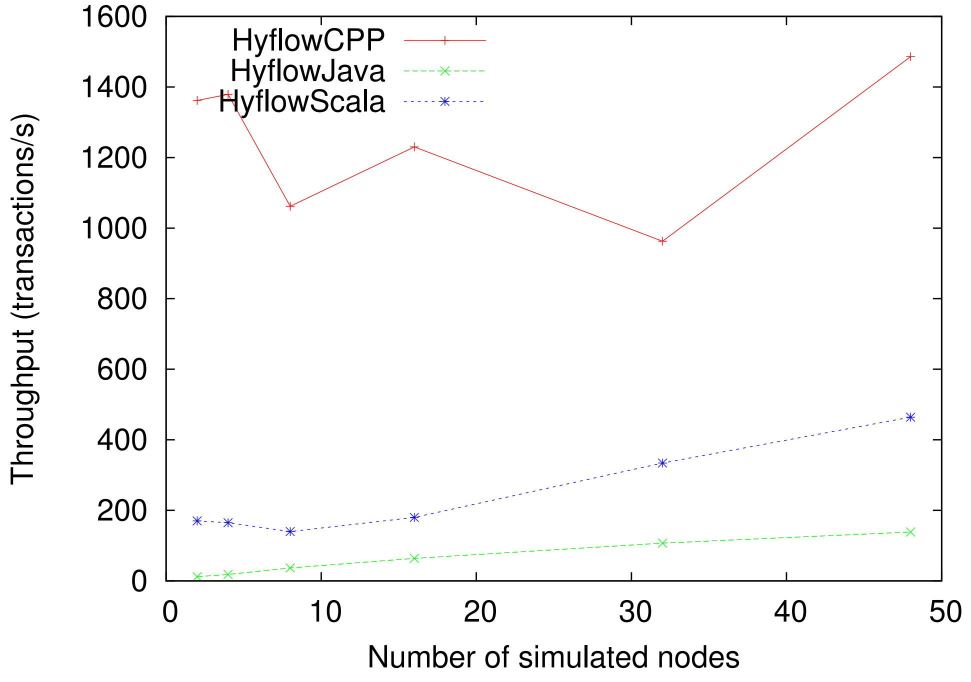
6.2.1.3 Skip List

In this benchmark, we have designed standard Skip-list data structure. Similar to Linked-list, in Skip-list we do not allow the duplicate values. Skip-list benchmark supports two write operations: add and remove, and one read operation: contains. In Skip-list, contention is generally lower than the Linked-list benchmark as it requires fewer nodes to traverse. We logarithmically distribute the number of layers in any given Skip-list node, which means Skip-list node count decrease as we move from Skip-list nodes having fewer numbers of layers to higher numbers of layers. Due to different layers, Skip-list read set size in generally smaller in comparison to Linked-list benchmark.

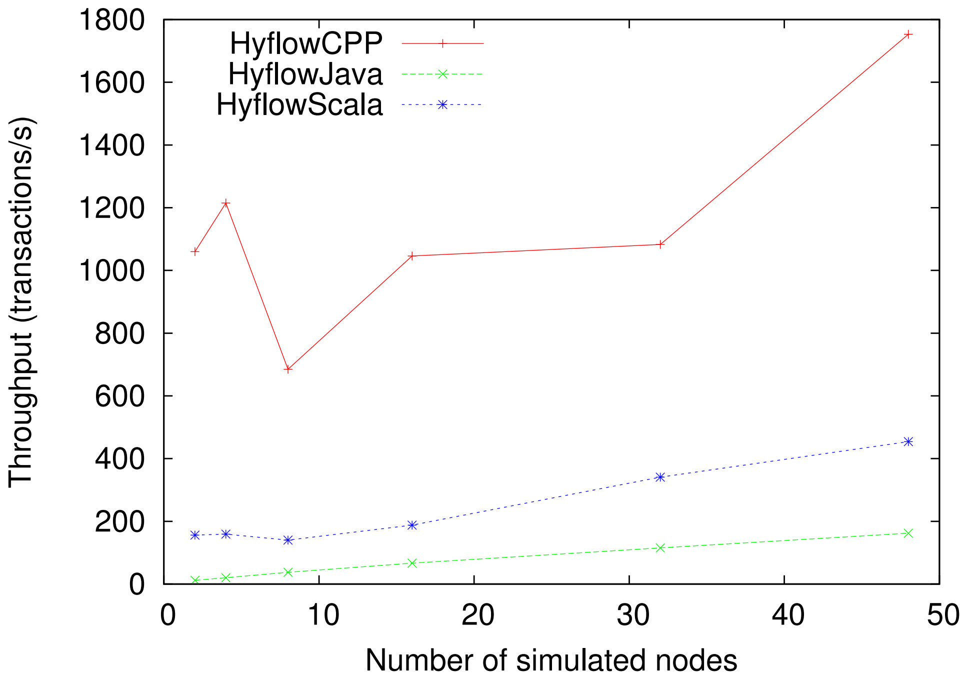
In our experiments, we have created the Skip-list with up to 50 objects and 5 layers. We have warmed up the Skip-list benchmark by adding 50% objects before starting the experiment. In figure 6.3, we can observe that HyflowCPP performance is up to 3 to 5 times better in comparison to HyflowScala. HyflowJava performs even worse in comparison to HyflowScala. With increase in number of nodes, the HyflowCPP performance for 20% reads decreases in general and for 50% and 80% it increases. Overall, due to smaller read set size the Skip-list throughput is better than Linked-list benchmark.



(a) Skiplist 20% reads



(b) Skiplist 50% reads



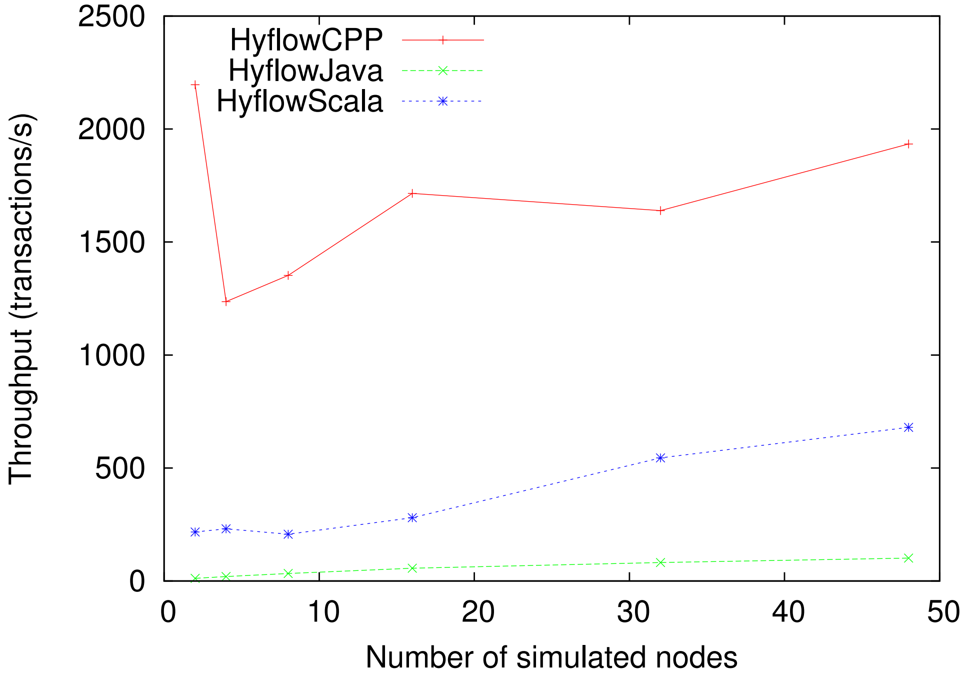
(c) Skiplist 80% reads

Figure 6.3: Flat Nesting: Transactional throughput for Skip-list

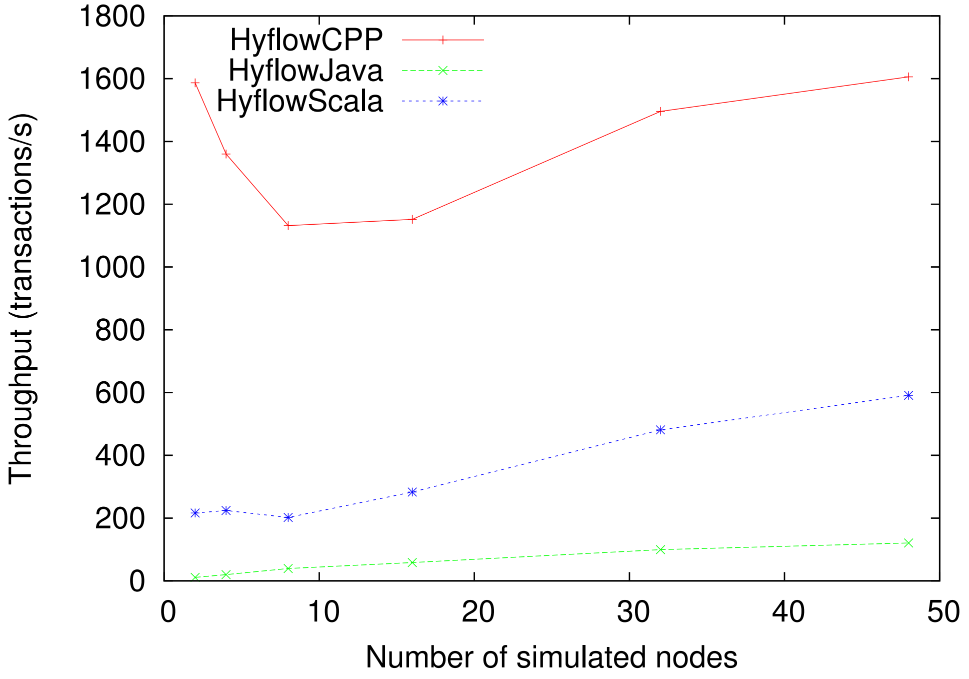
6.2.1.4 Binary Search Tree

In this benchmark, we have implemented the standard Binary Search Tree (BST). Similar to Linked-list and Skip-list, we do not allow the duplicate values in BST. It supports two write operations: add and remove, and one read operation: contains. BST provides the higher concurrency when compared to Linked-list, as it allows the concurrent read and write operations to go in parallel for different sub-trees.

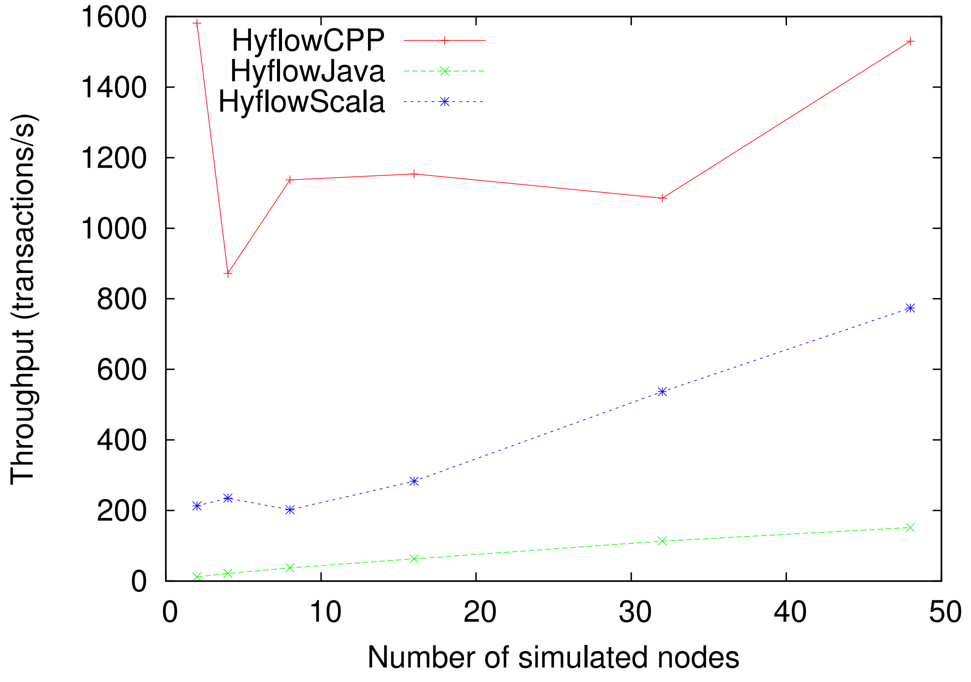
In our experiments, we have allowed up to 50 objects in BST and have warmed up the benchmark with 50% object before starting our experiments. In figure 6.4, we can observe HyflowCPP performs up to 3 to 5 times better in comparison to HyflowScala for different read ratios. HyflowJava performance remains poor as in previous cases. Due to higher level of concurrency, we find that throughput increases with increase in number of nodes for all read ratios. Higher concurrency available in BST can be easily understood through its performance, which is almost double in comparison to Linked-list.



(a) BST 20% reads



(b) BST 50% reads



(c) BST 80% reads

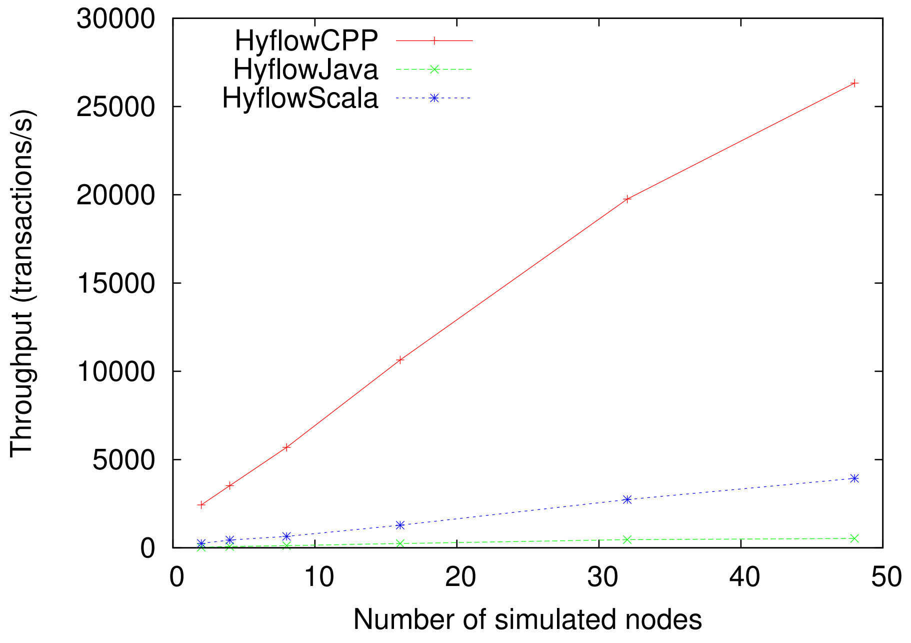
Figure 6.4: Flat Nesting: Transactional throughput for Binary Search Tree

6.2.1.5 Hash Table

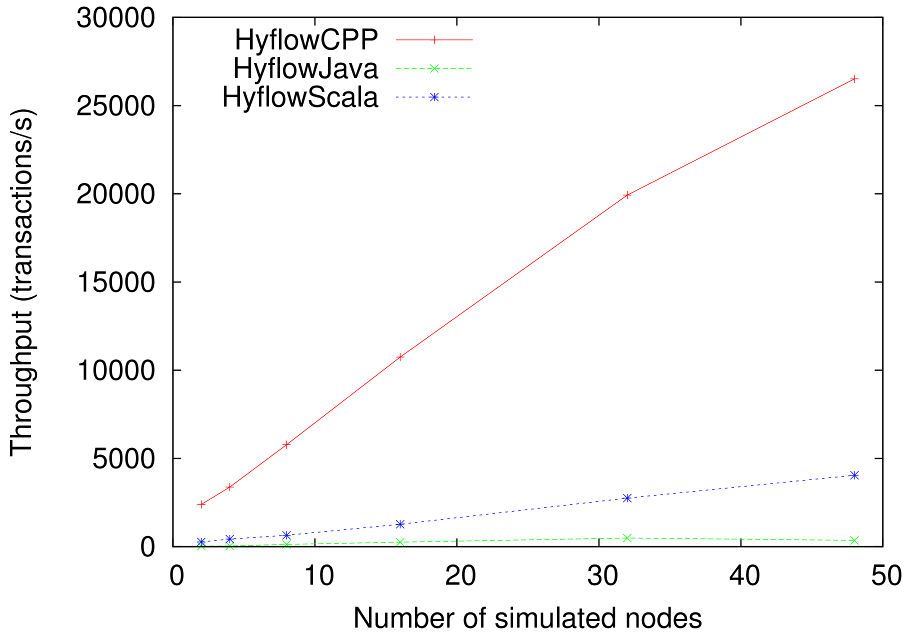
In this benchmark, we have implemented a bucket-based standard Hash-table. We do not allows duplicate keys in the Hash-table. It supports two write operations: add and remove, and one read operation: find. Hash-table is distributed in nature and it allows the concurrent operations on different buckets. Small read-write sets and distributed nature generally leads to very high throughput for Hash-table.

For our experiments, we created 2000 Hash-buckets distributed over different nodes. In figure 6.5, we can observe the performance of HyflowCPP in comparison to various competitors for different read percent: 20%, 50% and 80%. Hash-table being distributed in nature allows HyflowCPP performance to linearly scale with increase in the number of nodes. As we create 2000 Hash-buckets over the network, increase in number of nodes does not increase contention. Due to smaller read-write set size, the Hash-table throughput is highest among all the benchmarks.

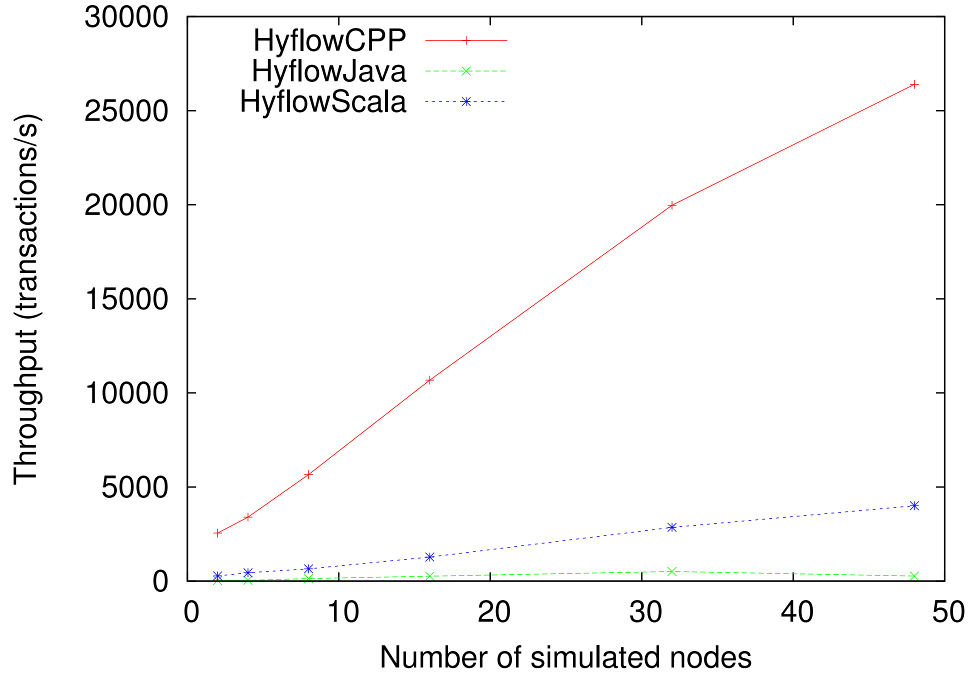
HyflowCPP performs up to 4 to 6 times better in comparison to its nearest competitor HyflowScala. HyflowJava performs even worse in comparison to HyflowScala. In Hash-table, we notice that contention happens only when different keys try to access same bucket, otherwise they can execute in parallel. Due to this reason, we do not see much difference in throughput for different read ratios.



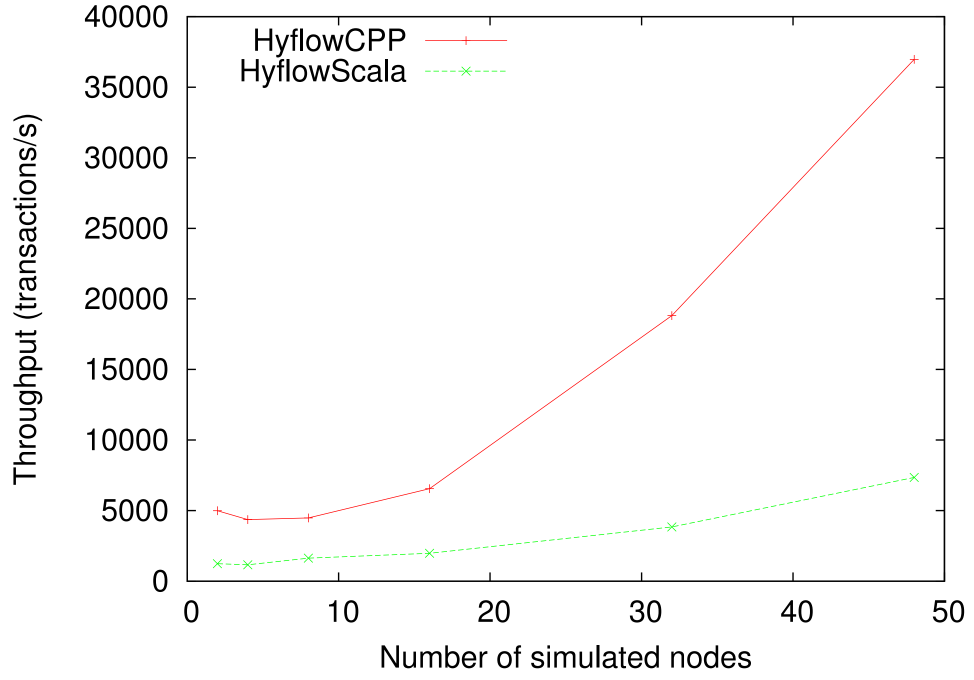
(a) Hash-table 20% reads



(b) Hash-table 50% reads



(c) Hash-table 80% reads



(d) Hash-table: HyflowCPP comparison with HyflowScala for up to 24 thread per node

Figure 6.5: Flat Nesting: Transactional throughput for Hash table

Similar to Bank benchmark, in Hash-table we have performed an experiment in which we add more number of threads per node to reduce the impact of network latency. We add more threads in HyflowCPP and HyflowScala until their CPU utilization reaches more than 98%. In such cases, the CPU time is fully utilized by running other transactional threads, while one of them waits for a network message to arrive. In figure 6.5(d), we compare the performance of HyflowScala and HyflowCPP for up to 24 threads per core. We find that Hash-table results are comparable to Bank benchmark. We achieve only 2 to 5 times performance improvement in comparison to HyflowScala as the impact of network latency and CPU utilization decreases.

6.2.2 Macro Benchmarks

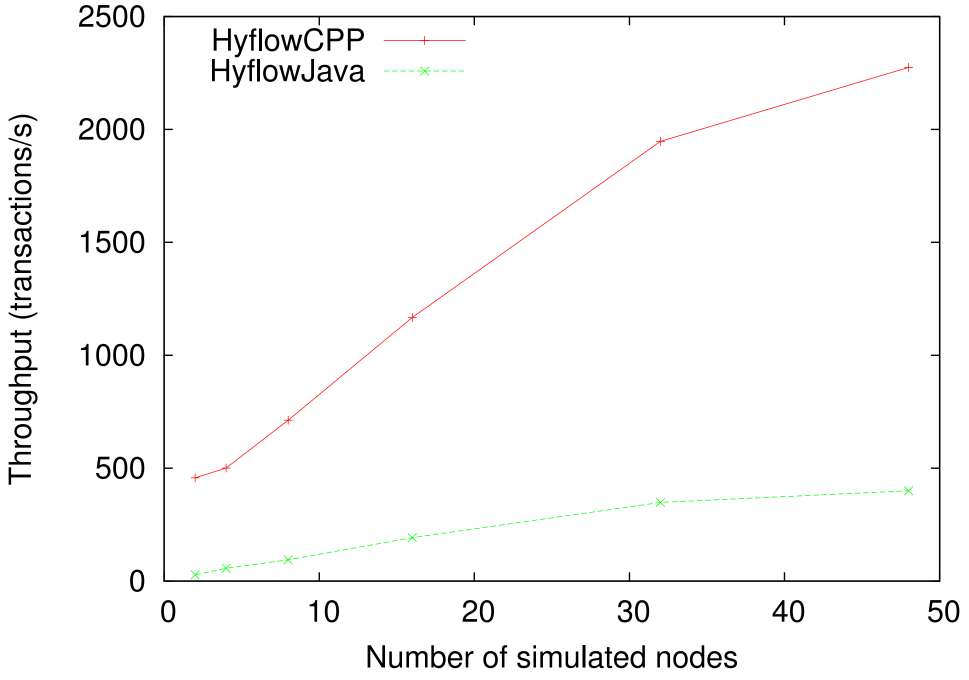
Macro Benchmarks are useful to verify the framework capability in real life applications. Macro benchmarks perform many operations leading to high messaging and transaction execution time. Due to the high messaging phenomenon, these macro benchmarks are helpful to analyze possible bottlenecks in form of messaging, synchronization, or memory usages in the framework. In this section, we evaluate the HyflowCPP performance on three macro benchmarks: Vacation, Loan, and TPCC.

6.2.2.1 Vacation

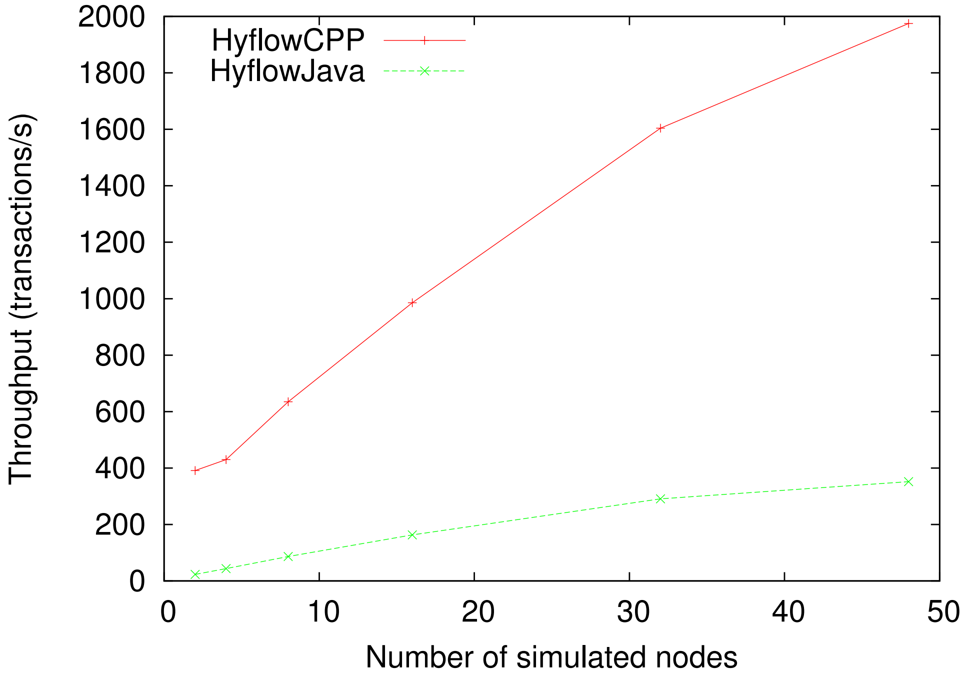
The Vacation benchmark emulates a client/server based itinerary planning and reservation system. It is a distributed version of STAMP Vacation benchmark application. It allows user to make online reservations for cars, rooms, and flights. It also enables administrators to add or delete the users, and update the currently available offers. Each reservation request or offer update request contains 10 queries, writing on 10 distributed objects.

Vacation executes the reservation request (a low contention operation) as a read operation, and delete customer and update offers (high contention operations) as write operations. For our experiments in the Vacation benchmark, we create total 10000 objects: 1000 customers, 3000 cars, 3000 flights, and 3000 rooms. We generate random reservation query of 10 objects and perform reservations of available objects.

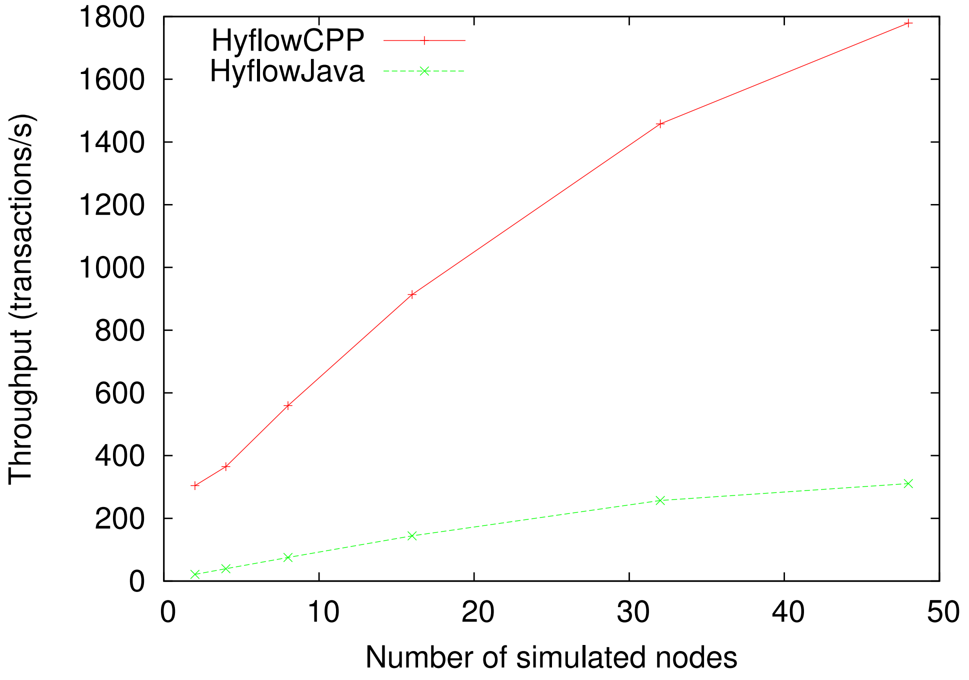
In figure 6.6, we can see that the throughput of the Vacation benchmark scales linearly with number of nodes. The high number of objects prevents increase in contention with increase in number of node. For the Vacation benchmark, read operation are more costly as they require to access more number of objects. We have compared the HyflowCPP performance with HyflowJava for different reads. HyflowCPP performs up to 4-5 times better than HyflowJava for Vacation benchmark.



(a) Vacation 20% reads



(b) Vacation 50% reads



(c) Vacation 80% reads

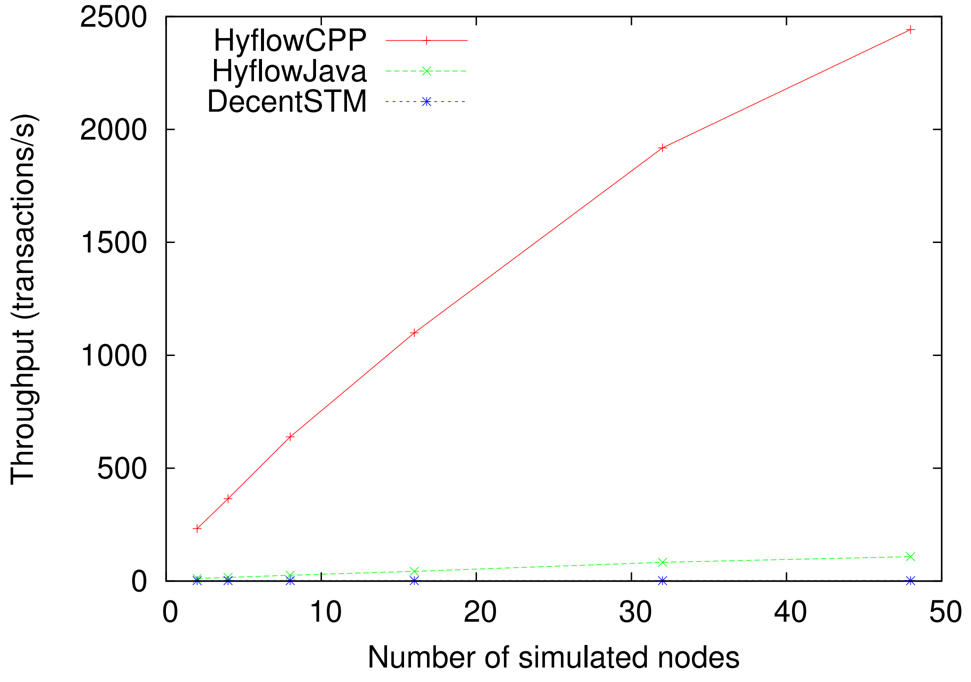
Figure 6.6: Flat Nesting: Transactional throughput for Vacation

6.2.2.2 Loan

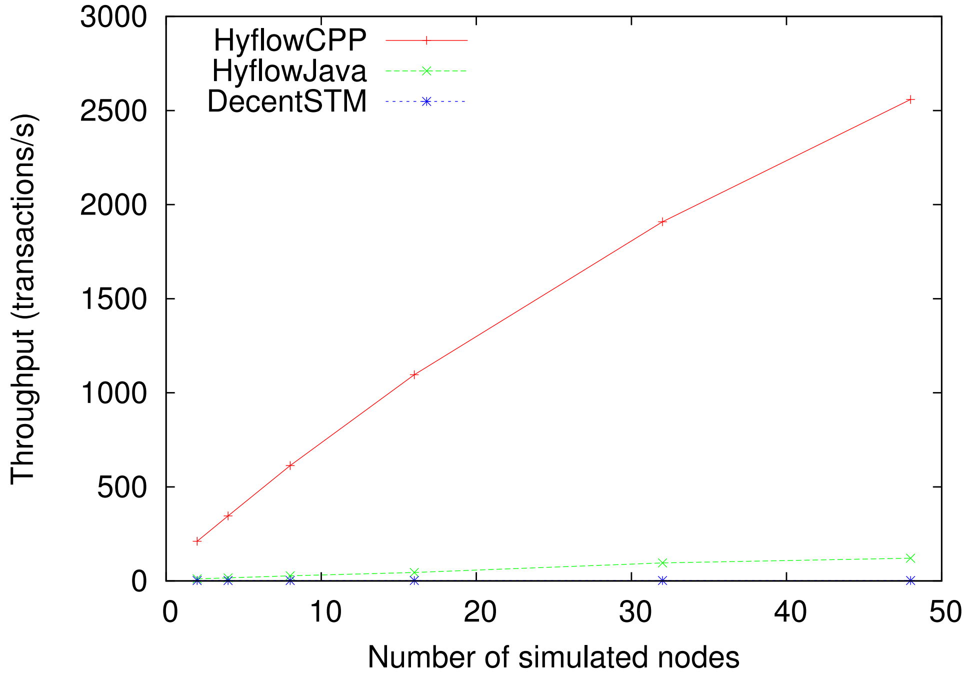
The Loan benchmark simulates the banking situation where loans are provided to user based on the different set of bank accounts. Each account is accessed in a nested fashion and a certain amount of money is borrowed based on the current amount. Loan benchmark also allows the user to get total balance available in the all accounts.

In our experiments, we run the total balance as read operation and loan as write operation. We create total 10000 account over network and access 6 accounts to provide loan or calculate the total balance. Loan operation calls itself in nested fashion either until all the account are used to borrow a random part of total loan money or total money is gathered.

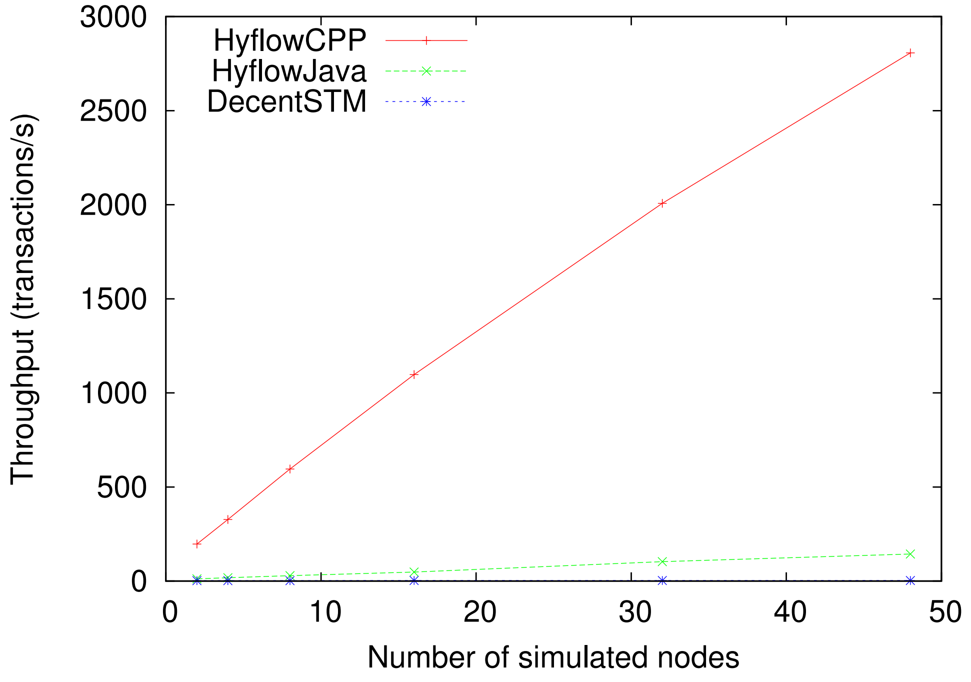
In figure 6.7, we compare HyflowCPP performance with two other frameworks HyflowJava and DecentSTM. We can see that it performs up to 5 to 10 times better than other frameworks. Throughput for 80% read is higher than 20% reads as write dominated experiments suffer from higher contention. At very low contention level, around 2 to 4 nodes, Loan benchmark's performance is very similar for different read ratios as read and write operations access same number of objects.



(a) Loan 20% reads



(b) Loan 50% reads



(c) Loan 80% reads

Figure 6.7: Flat Nesting: Transactional throughput for Loan

6.2.2.3 TPCC

TPC-C benchmark is a very popular yardstick for comparing OLTP performance on various hardware and software configurations. TPC-C simulates a computing environment where a group of users executes transactions against a database. TPC-C benchmark runs five operations in predefined ratio: New order (45%), Payment (43%), Delivery (4%), Order Status (4%), and Stock-level (4%). Stock-level operation is most heavy operation, which requires accessing 300 to 400 objects in a single transaction.

In our experiments, we do not vary the read ratios as in previous benchmarks, because TPCC have predefined operation ratio. TPCC access various objects in same repositories, for which a remote procedure call (RPC) based algorithm can be more efficient, because RPC request can be send to remote node to obtain the result, instead of bringing all the objects to requester node. In figure 6.8, we can observe that the performance of TPCC at start drops with increase in remote nodes as localization of repository objects reduces. Once almost all the objects are accessed remotely, throughput starts to increase with increase in number of nodes. Due to very large read and write set size TPCC benchmark have lowest throughput among all the tested benchmarks.

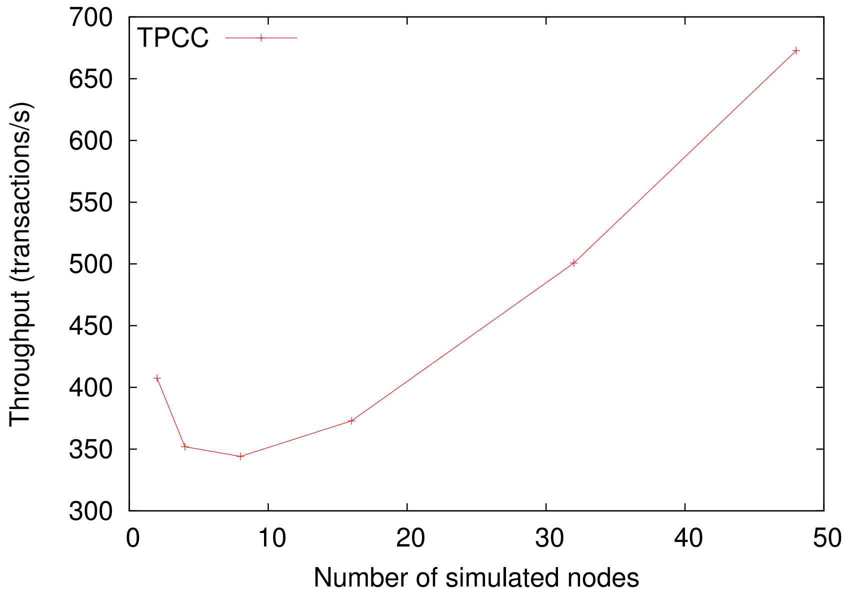


Figure 6.8: Flat Nesting: Transactional throughput for TPCC

6.3 Checkpointing and Closed Nesting

Checkpointing and closed nesting both provide methods to partial rollback in case of abort. We perform various experiments to understand the benefit of checkpointing and closed nesting over flat nesting. We perform our experiments on Bank, List, Skip-list, and Hash-table benchmarks. We alternate various parameters to understand the conditions in which these models can be useful.

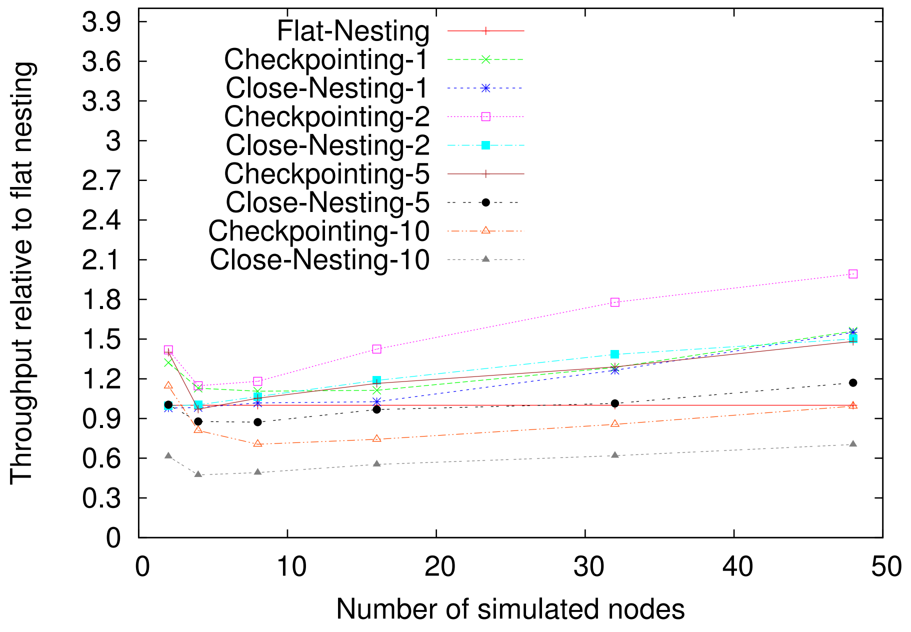
We have performed our experiments for different read-ratios: 20%, 50%, and 80% and different node counts: 2, 4, 8 and 16. These two parameters allowed us to vary the contention and abort count. We used less number of objects in our experiment so that enough aborts occur. Without significant number of aborts, measuring the impact of partial abort mechanism is not possible. We also varied number of objects based on number of inner transactions to maintain the same contention level across the experiments. We used different number of inner transaction to understand the impact of transactional length and points of partial aborts in transaction.

We can divide our experiments mainly in two groups. In first group, we maintained the same number of objects for all the experiments and changed the other parameters like read ratio, number of nodes, and number of inner transaction for different nesting models. In second group, we performed experiments same as in first group, except now we increased the number of objects in each experiment with increase in number of inner transaction. It enabled us to maintain the similar contention levels for different inner transactions. We also performed a third experiment on checkpointing to examine the impact of checkpoint granularity on transactional throughput. In our results, we represent checkpointing and closed nesting throughput relative to flat nesting.

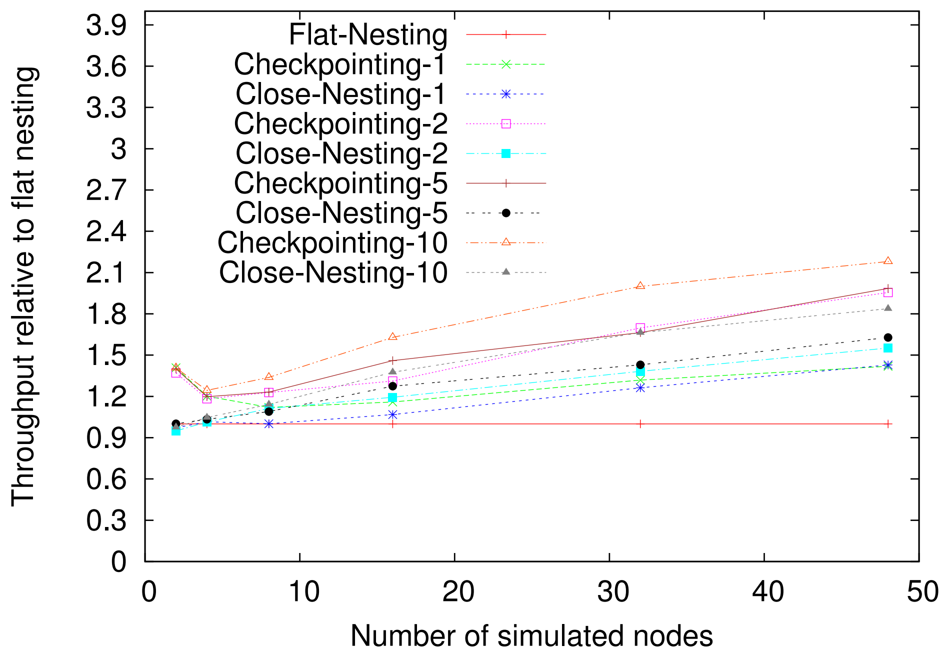
6.3.1 Bank

In figure 6.9(a), we present the results for 20% reads in first group of experiments. In these experiments, we maintain object count constant and increase node count and number of inner transactions. We present throughput for different inner transactions $1, 2, 5, 10$ for closed nesting and checkpointing relative to flat nesting. We can observe checkpointing performs better in comparison to flat nesting up to 90%, whereas closed nesting perform up to 60% better. We find that as we increase the inner transaction count, performance of both nesting models deteriorates due to increase in contention. For 10 inner transactions, actually checkpointing and closed nesting perform worse than flat nesting.

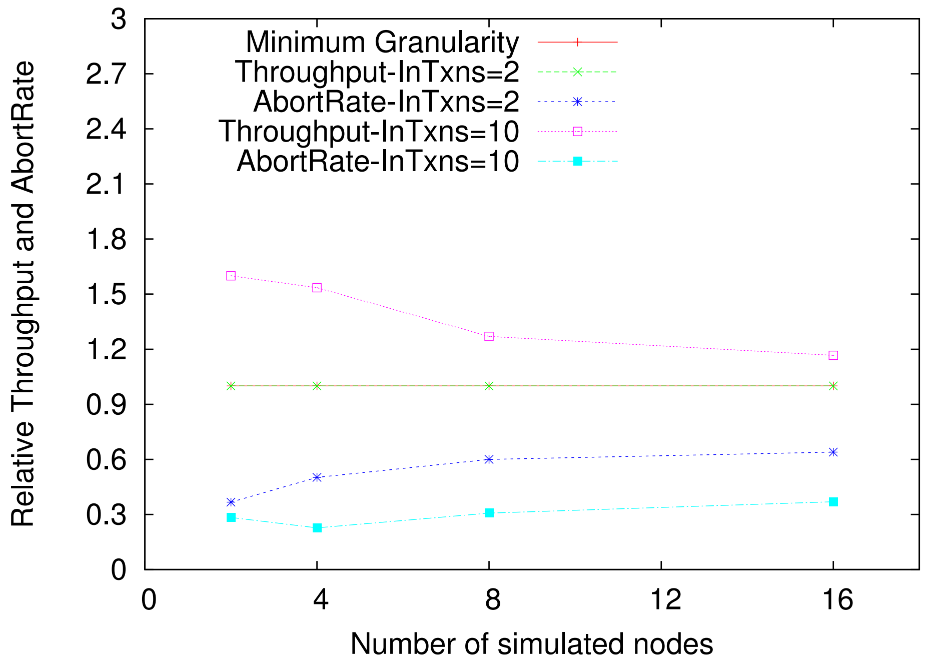
In figure 6.9(b), we present the results for 20% reads in second group of experiments. In these experiments, we increase object count with increase in number of inner transactions. We keep objects to inner transactions ratio constant, which allows us to maintain contention level with increase in number of inner transactions. We can observe that with increase in inner transactions, relative performance of closed nesting and checkpointing does not decrease, instead it is able to maintain pattern of improvements similar to lower number of inner transactions.



(a) Bank: Relative throughputs with constant object count for different number of inner transactions



(b) Bank: Relative throughputs with increasing object count for different number of inner transactions



(c) Bank: Throughput and Abort rate w.r.t minimum granularity

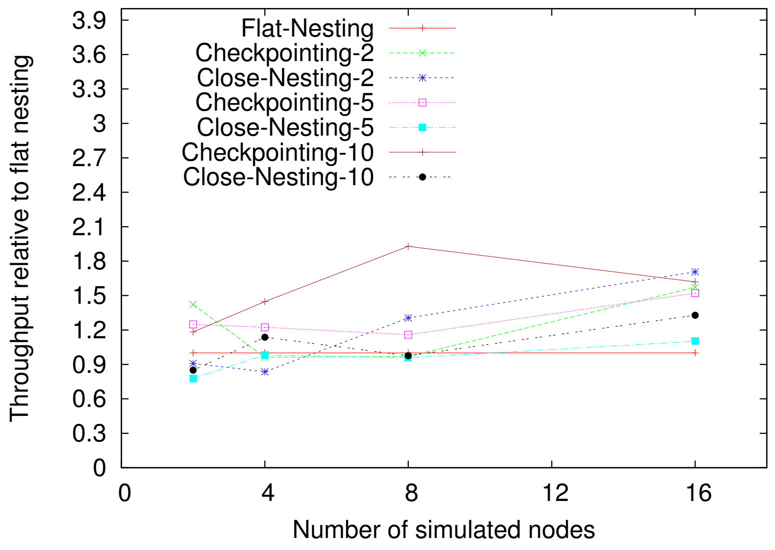
Figure 6.9: Closed Nesting and Checkpointing: Relative Transactional throughput for Bank

In figure 6.9(c), we present the relative change in throughput and abort rate for minimum to maximum granularity with inner transactions 2 and 10. We can observe that increase in checkpoint granularity decreases the abort rate, but throughput gains are relative smaller. Also with increase in contention level, these gains start to diminish. Lower throughput gains occur at higher contention level as lower number of partial rollbacks incur in transaction commit.

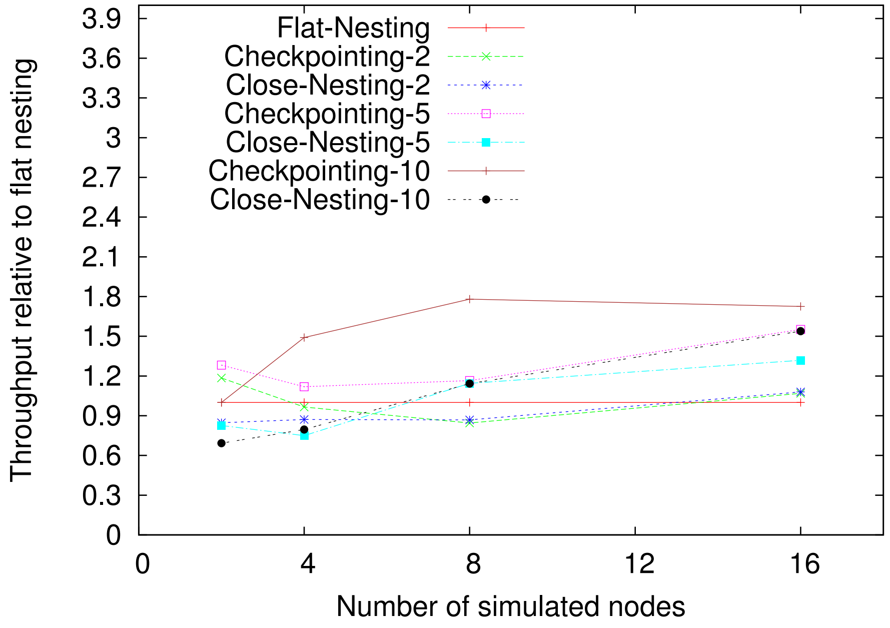
6.3.2 Linked List

In figure 6.10(a), we present the results for 20% reads in first group of experiments. In these experiments, we maintain object count constant and increase node count and number of inner transactions. We present throughput for different inner transactions $2, 5, 10$ for closed nesting and checkpointing relative to flat nesting. We can observe that with lower number of inner transactions checkpointing and closed nesting perform worse in comparison to flat nesting. However, at higher inner transaction count both nesting models performance improves. Here, we find that checkpointing and closed nesting are helpful only in case of higher inner transactions for List benchmark.

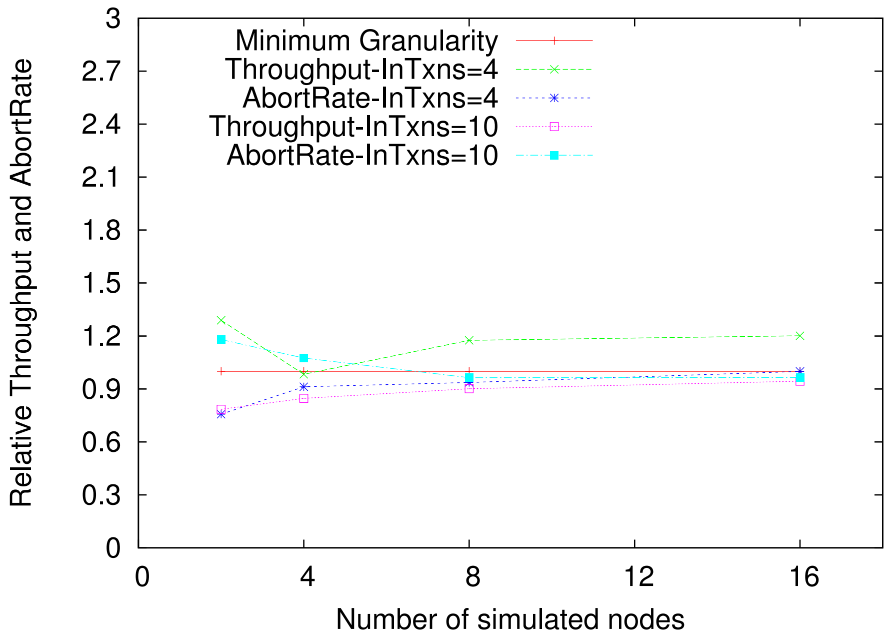
In figure 6.10(b), we present the results for 20% reads in second group of experiments. In these experiments, we increase object count with increase in number of inner transactions. For such experiments, we obtain similar pattern as in figure 6.10(a). Similar pattern can be easily understood as increasing the list size does not influence contention level. In List, only one transaction, which writes on the nearest node to head, commits. Read set size does not impact on contention level in List.



(a) Linked List: Relative throughputs with constant object count for different number of inner transactions



(b) Linked List: Relative throughputs with increasing object count for different number of inner transactions



(c) Linked List with decreasing granularity

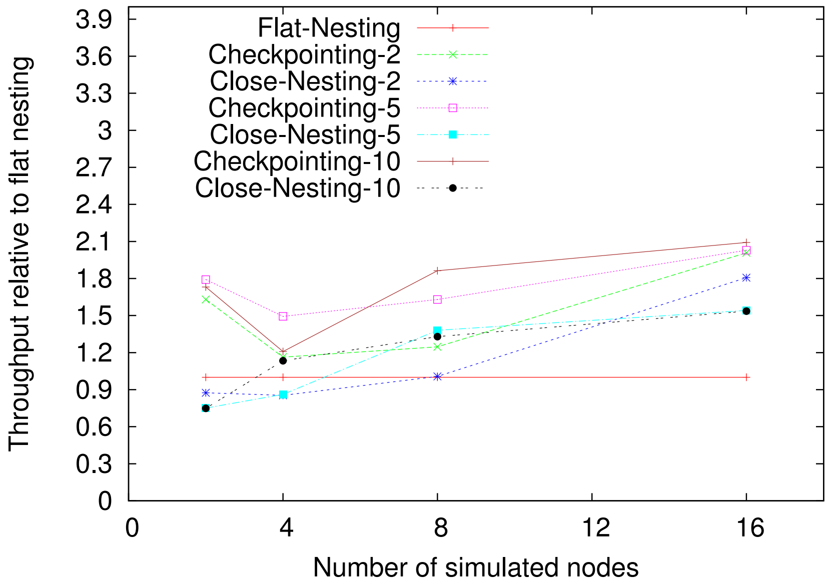
Figure 6.10: Closed Nesting and Checkpointing: Relative Transactional throughput for Linked List

In figure 6.10(c), we present the relative change in throughput and abort rate for minimum to maximum granularity with inner transactions 4 and 10. we can observe that at very high contention abort rate and throughput are not effected by transaction granularity.

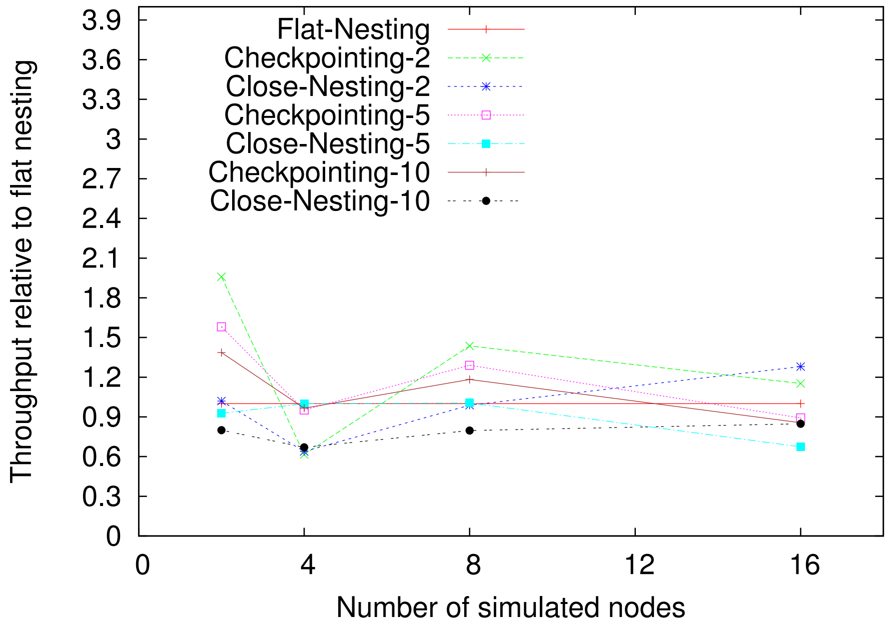
6.3.3 Skip List

In figure 6.11(a), we present the results for 20% reads in first group of experiments. In these experiments, we maintain object count constant and increase node count and number of inner transactions. We present throughput for different inner transactions $2, 5, 10$ for closed nesting and checkpointing relative to flat nesting. We can observe that checkpointing and closed nesting performs worse in comparison to flat nesting for lower numbers of inner transactions. With increase in inner transactions, more partial rollback points are created and overall performance improves.

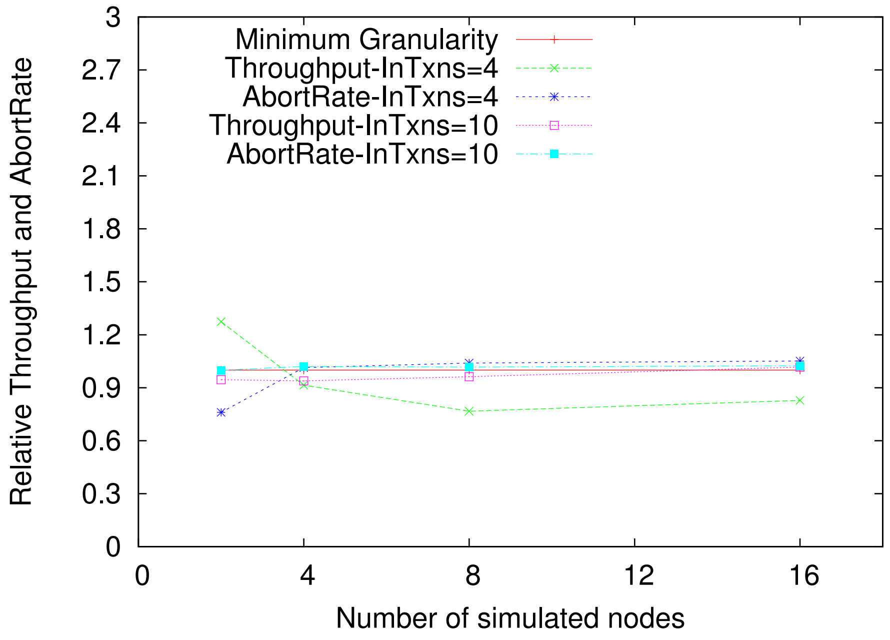
In figure 6.11(b), we present the results for 20% reads in second group of experiments. In these experiments, we increase object count with increase in number of inner transactions. We can observe that overall improvement using closed nesting and checkpointing actually deteriorates in comparison to first group of experiments. This loss in performance can be explained by increase in contention due to bigger read set size, because with increase in Skip-list size the number of nodes required to access the target node increases in Skip-list.



(a) Skip-list: Relative throughputs with constant object count for different number of inner transactions



(b) Skip-list: Relative throughputs with increasing object count for different number of inner transactions



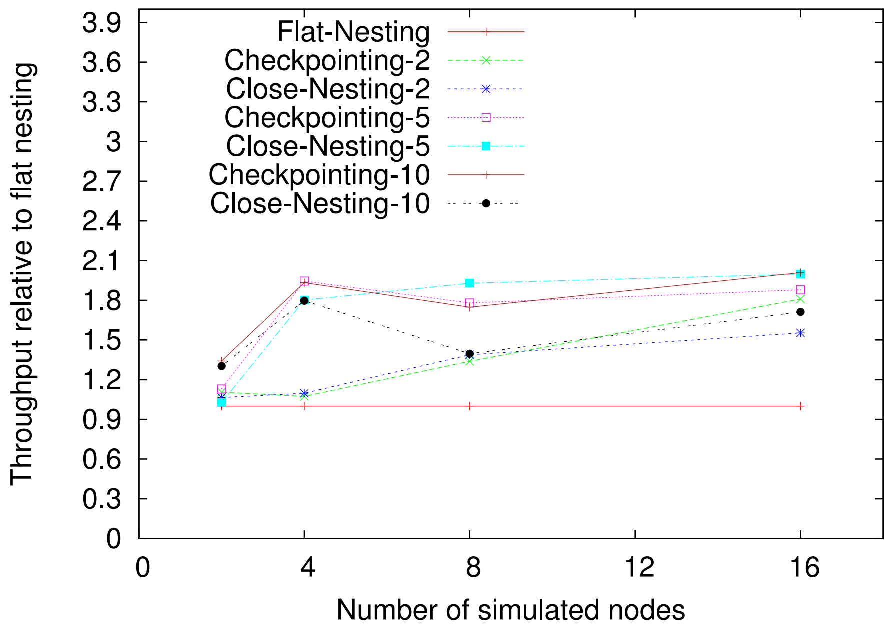
(c) Skip-list: Relative throughputs with increasing object count for different number of inner transactions

Figure 6.11: Closed Nesting and Checkpointing: Relative Transactional throughput for Skip-list

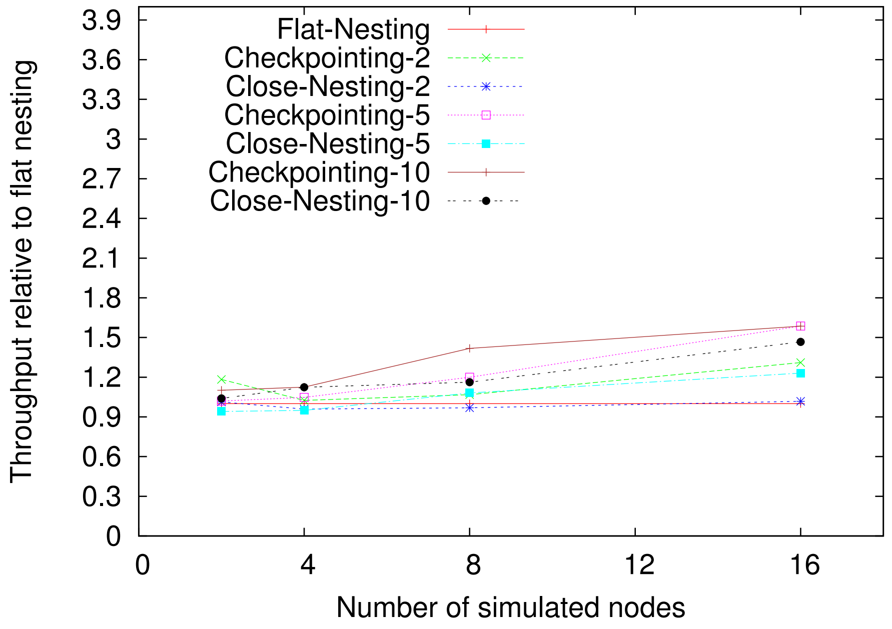
In figure 6.11(c), we present the relative change in throughput and abort rate for minimum to maximum granularity with inner transactions 4 and 10. We can observe the improvement in throughput for lower number of inner transaction with drop in abort rate. However, in case of higher number of inner transactions i.e., higher contention, varying the checkpointing granularity is not helpful in performance improvement.

6.3.4 Hash Table

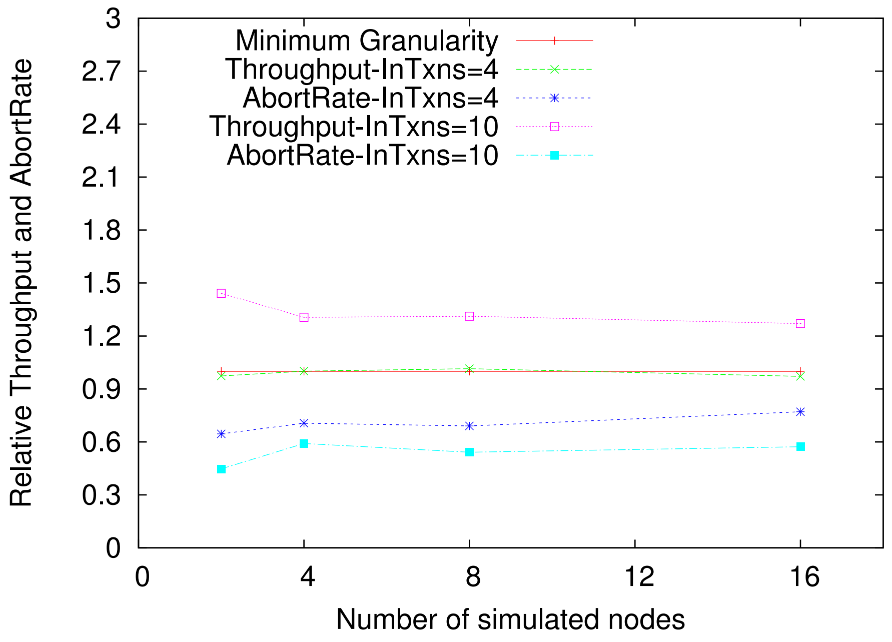
In figure 6.12(a), we present the results for 20% reads in first group of experiments. In these experiments, we maintain object count constant and increase node count and number of inner transactions. We present throughput for different inner transactions $2, 5, 10$ for closed nesting and checkpointing relative to flat nesting. We can observe checkpointing performs better in comparison to flat nesting up to 100%, whereas closed nesting perform up to 90% better. We can observe that as we increase the inner transaction count, both nesting models performance improves with increase in partial rollback points. This behavior is contrary to Bank where increase in inner transaction deteriorates the performance. In Hash-table only one object is accessed in each transaction, therefore increase in inner transactions does not increase the contention by much. Instead, increase in partial rollback points helps to improve performance.



(a) Hash-table: Relative throughputs with constant object count for different number of inner transactions



(b) Hash-table: Relative throughputs with increasing object count for different number of inner transactions



(c) Hash-table: Throughput and Abort rate w.r.t minimum granularity

Figure 6.12: Closed Nesting and Checkpointing: Relative Transactional throughput for Hash-table

In figure 6.12(b), we present the results for 20% reads in second group of experiments. In these experiments, we increase object count with increase in number of inner transactions. We keep objects to inner transactions ratio constant, which allows us to prevent increase in contention level with increase in number of inner transactions. We can observe that with increase in inner transactions relative performance of closed nesting and checkpointing does not decrease due to maintained contention level.

In figure 6.12(c), we present the relative change in throughput and abort rate for minimum to maximum granularity with inner transactions 4 and 10. We can observe that increase in checkpoint granularity decreases the abort rate 30% to 70%, but increase in throughput is only 20% to 50%. Relative low improvement in throughput occurs, as not all partial rollbacks incur in transaction commit.

6.4 Open Nesting

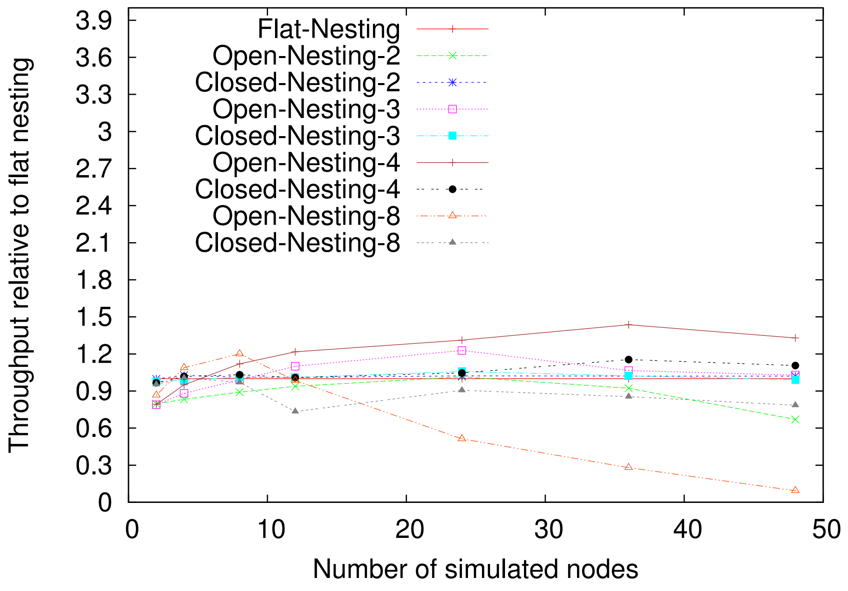
The aim of our experiments for open nesting is to analyze its performance under different the parameters and identify the workload conditions which are best for open nested transactions. We compare the improvements achieved by open nesting with flat and closed nesting. We perform our experiments on configurable benchmarks Hash-table, Skip-list, and BST for nodes: $2, 4, 8, 16, 24, 36, and 48$.

We test our open nesting implementation by varying various parameters like read ratio, number of transactional nodes, and number of inner transactions. We perform all these experiments for flat nesting, closed nesting, and open nesting. In each experiment, we collect various meta-data like committed/aborted transactions, committed/aborted sub-transactions (closed and open nesting), committed/aborted compensating/commit actions (open nesting only) and waiting time after aborted (sub-)transactions (for back-off).

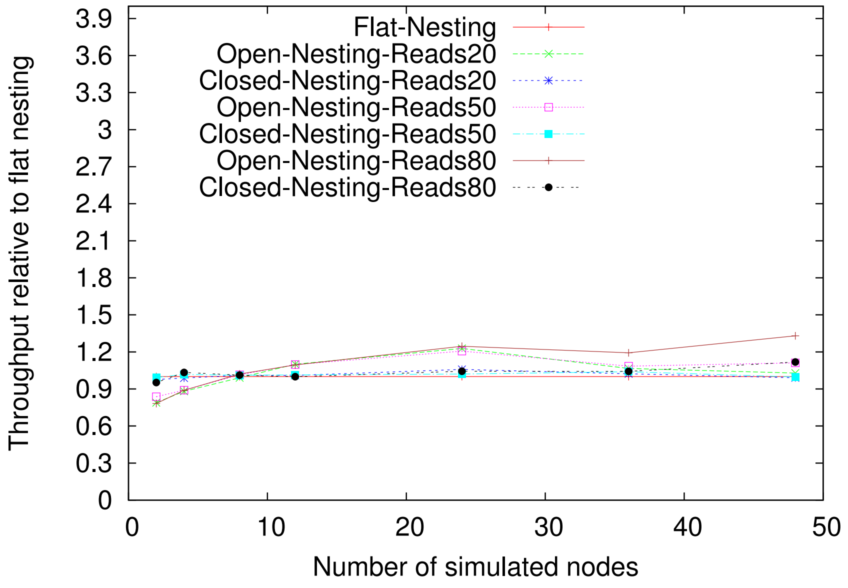
We represent our results for each benchmark in four different plots. In first plot we present the relative throughput improvement achieved by open nesting and closed nesting relative to flat nesting for different number of inner transaction per transaction. Second plot describes the impact of different read to write ratios on open and closed nesting with respect to flat nesting. In third plot, we compare flat, closed, and open nesting performance for higher object count. In last plot, we present the total time spent by open nested transaction in various actions like commit, abort, abort compensation, and back-off.

6.4.1 Hash Table

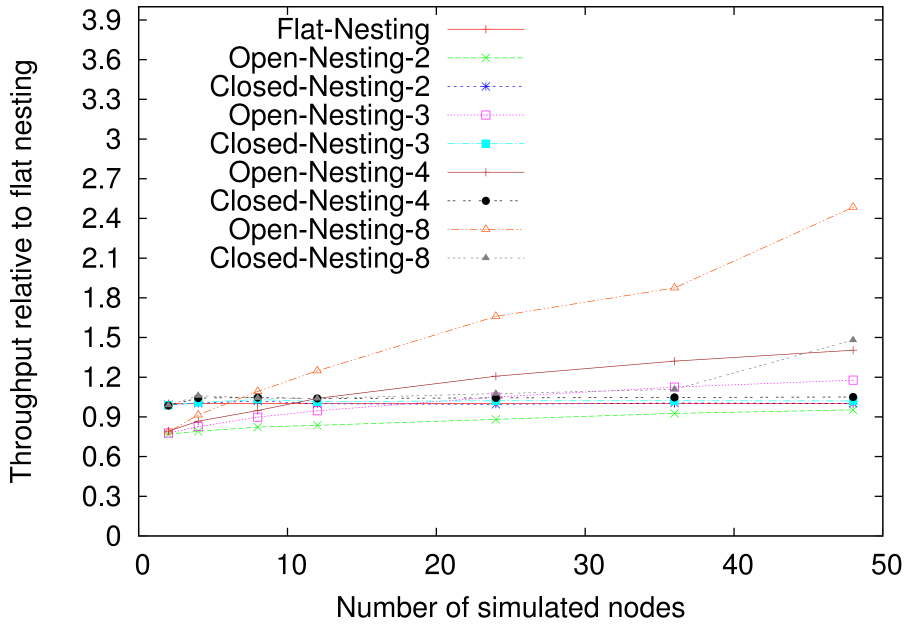
In figure 6.13(a), we present the open and closed nesting throughput relative to flat nesting for different number of inner transactions, and 300 bucket objects. We can observe that as we increase the number of inner transactions open nesting performance improves as amount of contention increases. However, at very higher contention for 8 inner transaction performance actually falls due to increased abstract lock contention.



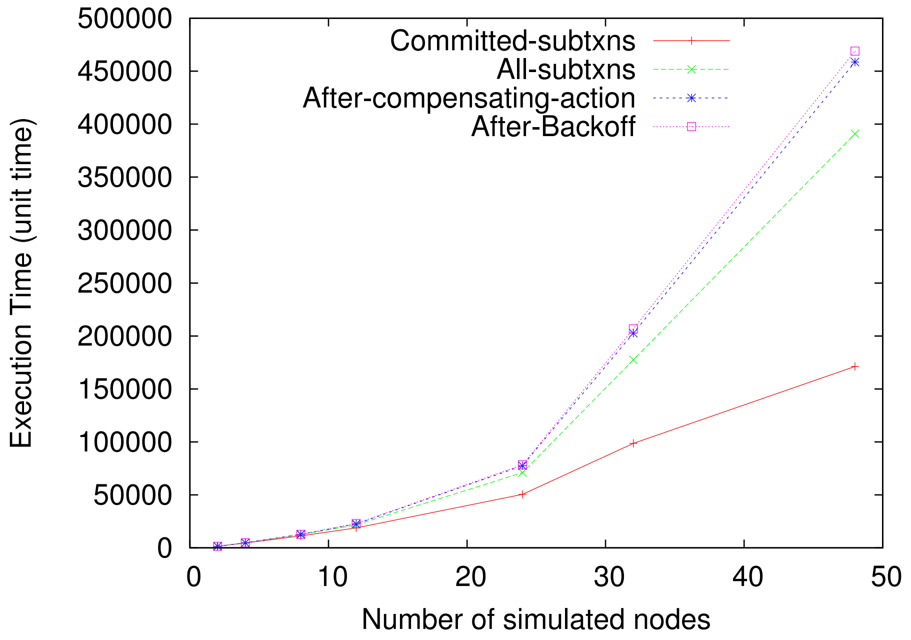
(a) Hash-table: Open and closed nesting throughput relative to flat nesting with 300 buckets for different number of inner transactions



(b) Hash-table: Open and closed nesting throughput relative to flat nesting for read ratios



(c) Hash-table: Open and closed nesting throughput relative to flat nesting with 1500 buckets for different number of inner transactions



(d) Hash-table: Cumulative execution time split for open nested transaction

Figure 6.13: Open Nesting: Relative Transactional throughput and execution time split for Hash-table

In figure 6.13(b), we present the open and closed nesting throughput relative to flat nesting for different read ratios for 3 inner transactions. We find increase in throughput with increase in read ratio due to decreased contention. It is worth noting that we have observed very high abort rate in this experiment from 200% to 4000%.

In figure 6.13(c), we present the open and closed nesting throughput relative to flat nesting for different number of inner transaction, and 1500 bucket objects. We can observe the performance improvement in open nesting as fundamental contention decreases.

In figure 6.13(d), we examine the cumulative time split of open nested transactions for 4 inner transactions. We can observe that with increase in number of nodes the commit time increase slowly, but increase in total sub-transaction time and abort compensation time is high. Back-off time also increases, but it constitutes very small part of total time spent.

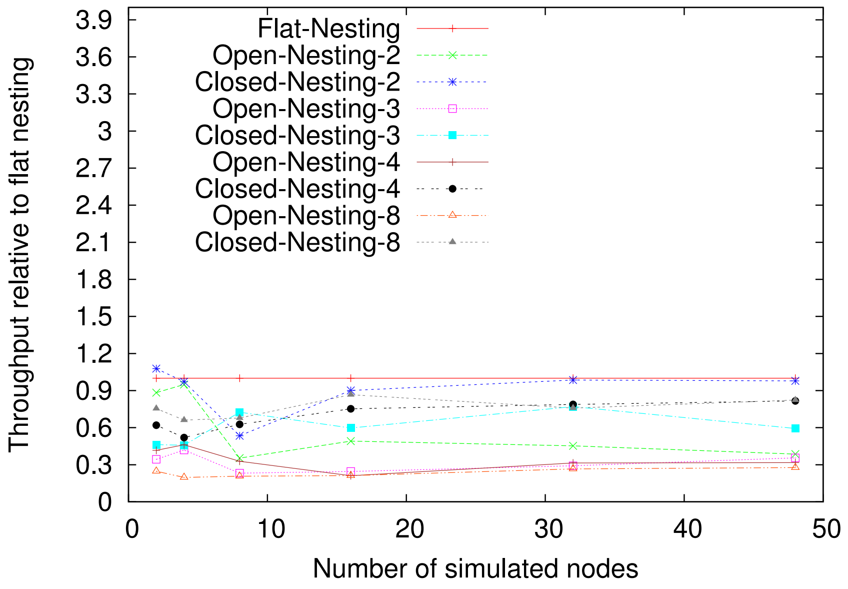
6.4.2 Skip List

In figure 6.14(a), we present the open and closed nesting throughput relative to flat nesting for different number of inner transactions, and 100 objects. We can observe that as we increase the number of inner transactions open nesting performance decreases. It can be explained based on read-set object caching. For flat nesting and closed nesting, objects accessed by previous inner transaction are cached in read-set. For higher inner transaction count, almost all objects are access locally by inner transactions executing after first 2-3 inner transactions. Meanwhile, open nested transactions are required to fetch the objects over network, which increases the execution time significantly. It is also worth mentioning that open nesting has additional messaging overhead for maintaining distributed abstract-locks.

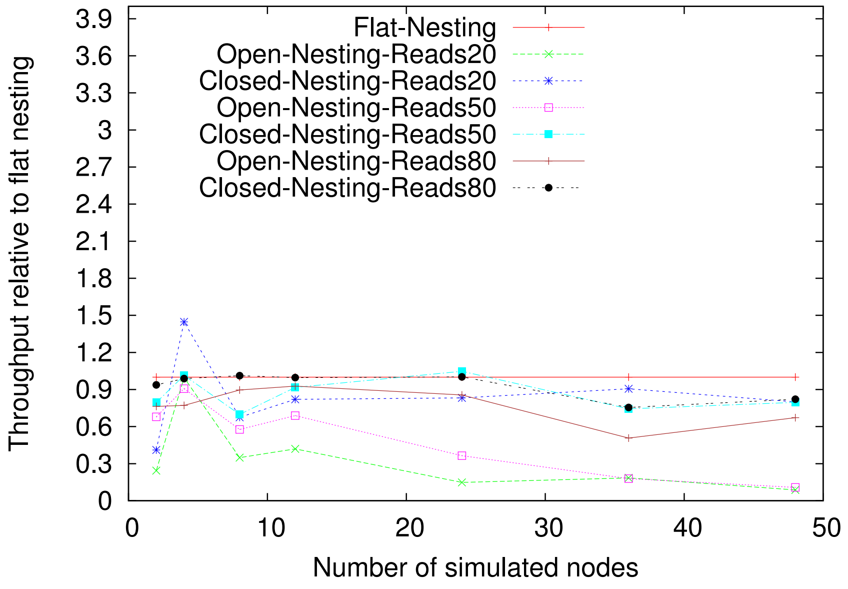
In figure 6.14(b), we present the open and closed nesting throughput relative to flat nesting for different read ratios for 3 inner transactions. We find increase in throughput with increase in read ratio due to decreased contention. It is worth noting we have observed very high abort rate in this experiment from 200% to 9000%.

In figure 6.14(c), we present the open and closed nesting throughput relative to flat nesting for different number of inner transaction, and 1000 objects. We can observe the performance gain in open nesting performance for lower number of inner transactions, where the object caching effect is less dominating. With increase in inner transactions open nesting performance degrades. Performance gain for higher numbers of objects in open nesting can be explained by reduction in fundamental contention.

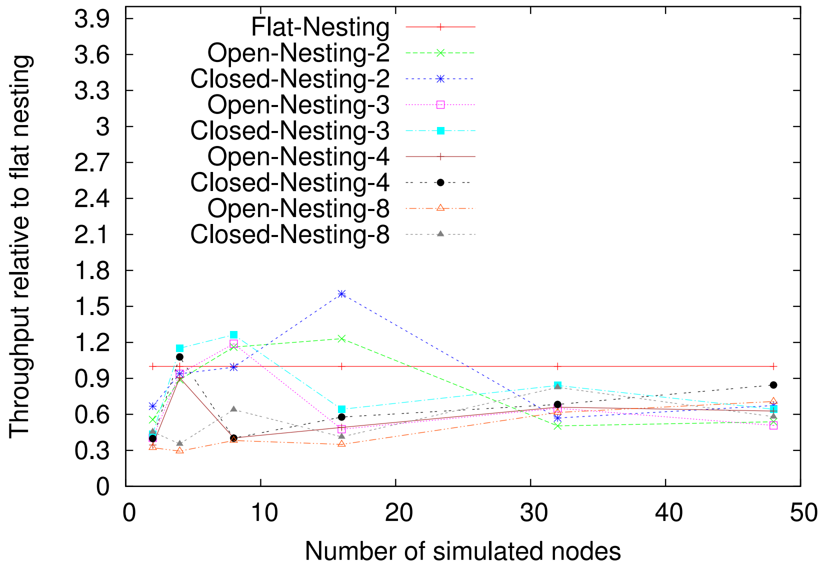
In figure 6.14(d), we examine the cumulative time split of open nested transactions. We can observe that with increase in number of nodes the commit time, abort time and compensation time in nodes do not change much, but increase in back-off time is nearly linear. It indicates towards continuous aborts while execution and explain the low gains achieved in Skip-list.



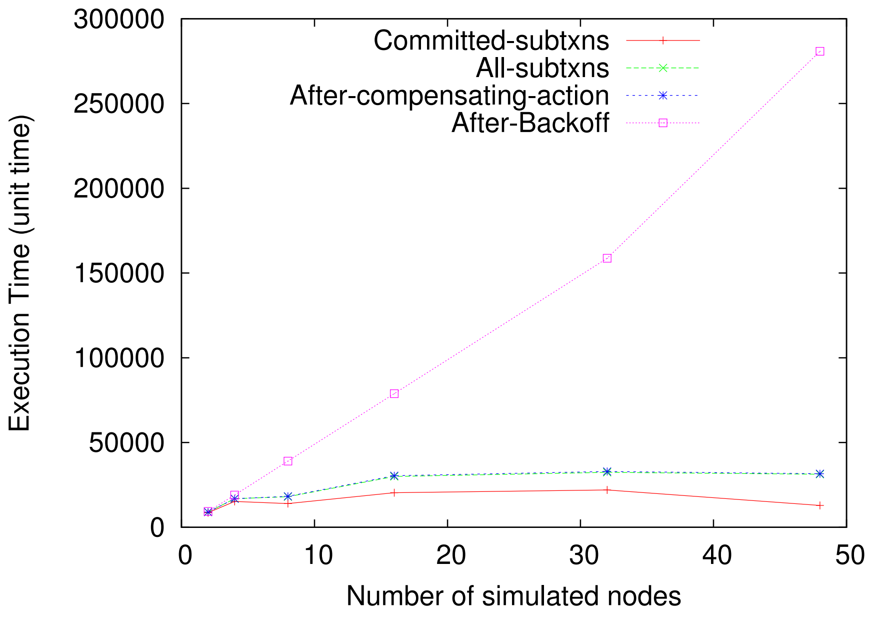
(a) Skip-list: Open and closed nesting throughput relative to flat nesting with 100 objects for different number of inner transactions



(b) Skip-list: Open and closed nesting throughput relative to flat nesting for read ratios



(c) Skip-list: Open and closed nesting throughput relative to flat nesting with 1000 objects for different number of inner transactions



(d) Skip-list: Cumulative execution time split for open nested transaction

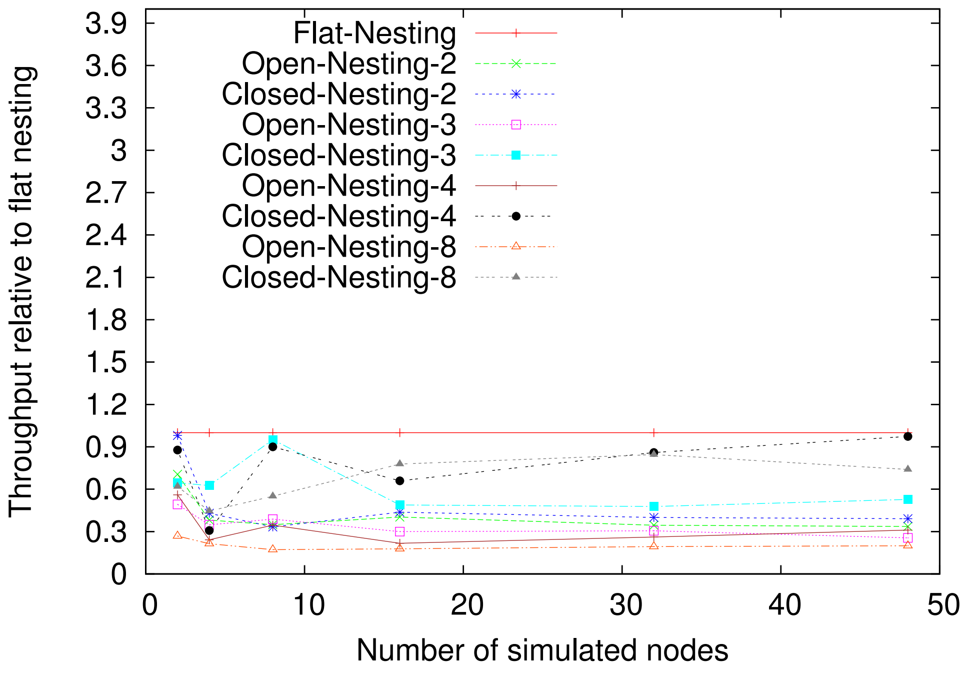
Figure 6.14: Open Nesting: Relative Transactional throughput and execution time split for Skip-list

6.4.3 Binary Search Tree

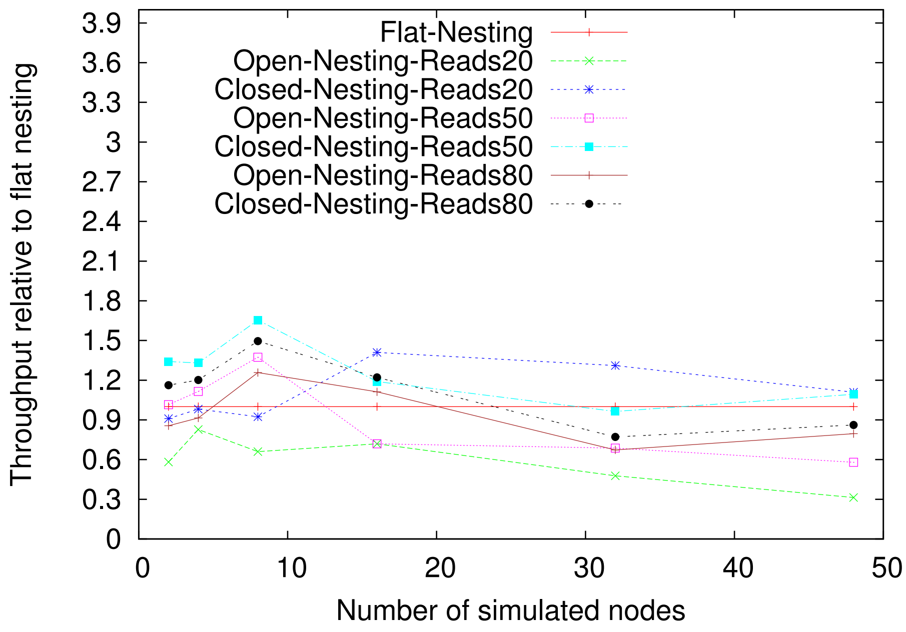
In figure 6.15(a), we present the open and closed nesting throughput relative to flat nesting for different number of inner transactions, and 100 objects. We can observe that as we increase the number of inner transactions open nesting performance decreases. It can be explained based on read-set object caching similar to Skip-list, which decreases the execution time for Flat and Closed nesting significantly.

In figure 6.15(b), we present the open and closed nesting throughput relative to flat nesting for different read ratios for 3 inner transactions. We find increase in throughput with increase in read ratio due to decreased contention. It is worth noting we have observed very high abort rate in this experiment from 200% to 6000%.

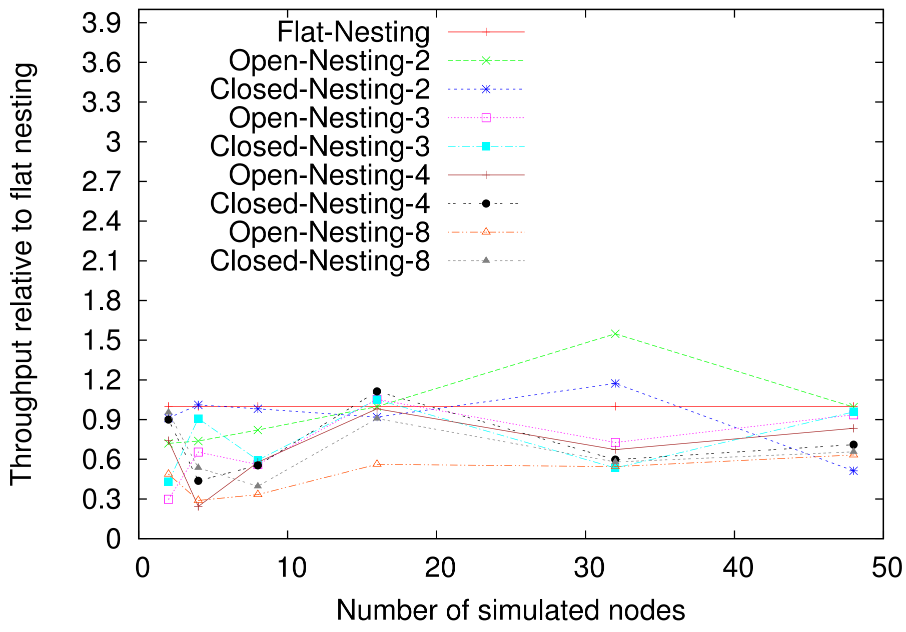
In figure 6.15(c), we present the open and closed nesting throughput relative to flat nesting for different number of inner transaction, and 1000 objects. Similar to Skip-list, we can observe the performance gain in open nesting for lower number of inner transactions, where the object caching effect is less dominating. With increase in inner transactions open nesting performance degrades. Performance gain for higher numbers of objects in open nesting can be explained by reduction in fundamental contention.



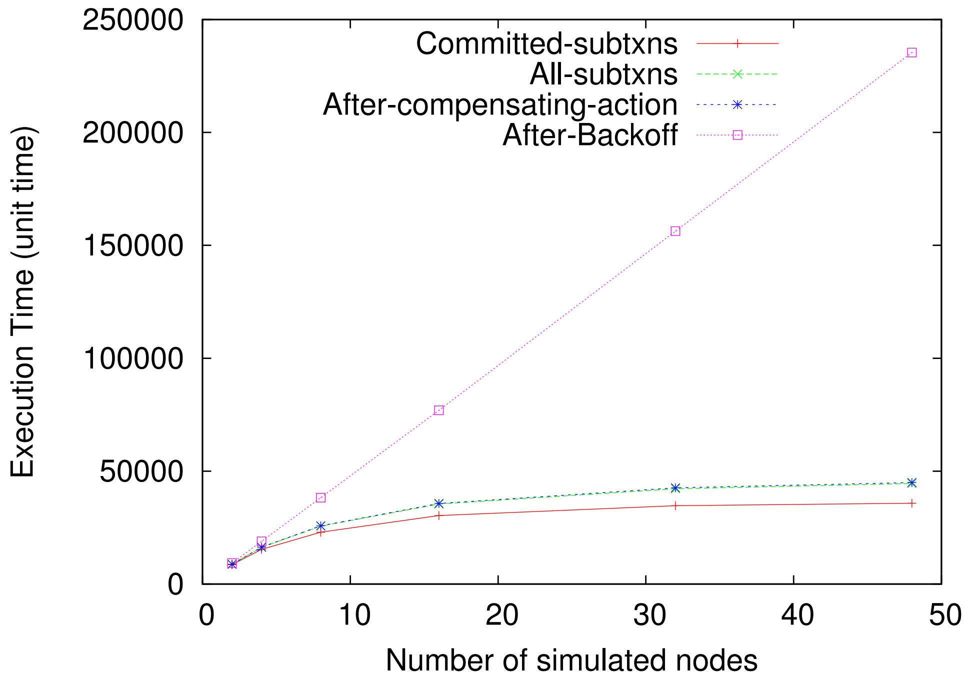
(a) BST: Open and closed nesting throughput relative to flat nesting with 100 objects for different number of inner transactions



(b) BST: Open and closed nesting throughput relative to flat nesting for read ratios



(c) BST: Open and closed nesting throughput relative to flat nesting with 1000 objects for different number of inner transactions



(d) BST: Cumulative execution time split for open nested transaction

Figure 6.15: Open Nesting: Relative Transactional throughput and execution time split for BST

In figure 6.15(d), we examine the time split of open nested transactions. Similar to Skip-list, we can observe that with increase in number of nodes, the commit time, abort time and compensation time do not change much, but increase in back-off time is nearly linear. It indicates towards continuous aborts while execution and explain the low gains achieved in BST.

6.5 Summary

In summary, we can see HyflowCPP provides a very high throughput in comparison to its competitors like GenRSTM, DecentSTM, HyflowJava, and HyflowScala for flat nesting experiments. We achieved maximum performance improvement up to 6 times in comparison to HyflowScala. Other competitors like GenRSTM, DecentSTM, and HyflowJava perform even worse in comparison to HyflowScala. These high improvements are achieved in cases where network latency dominates over throughput and CPU is underutilized by JVM based frameworks like HyflowScala. However, in cases of high CPU utilization we achieved up to 2 to 5 times performance improvement in comparison to HyflowScala.

Our close nesting and checkpointing, achieved up to 90% and 100% performance gains respectively. For first time we achieved higher performance gain using checkpointing in comparison to closed nesting. Our experiments showed inefficacy of checkpointing granularity in high contention levels. Open nesting experiments demonstrated as much as 140% improvement over flat nesting and 90% over closed nesting. We achieved lower performance gain in open nesting for large read-set size benchmarks like Skip-list and Binary Search Tree.

Chapter 7

Conclusion and Future Work

In this thesis, we presented a DTM framework for C++ called HyflowCPP. HyflowCPP provides a generic programming interface-based DTM abstraction, with pluggable support for different concurrency control algorithms, directory lookup protocols, contention management policies, and network communication protocols. The framework also supports transactional features including strong atomicity, flat nesting, closed nesting, open nesting, and checkpointing for improved performance and code composability.

We implemented the Transaction Forwarding Algorithm for concurrency control in HyflowCPP. We conducted experimental studies using a set of micro-benchmarks and macro-benchmarks and comparing with competitor (JVM-based) DTM frameworks including GenRSTM, DecentSTM, HyflowJava, and HyflowScala. We observed that, HyflowCPP's transactional throughput scales robustly with increase in number of objects, read-write ratio, and number of nodes. Our results for flat nesting show that HyflowCPP outperforms its nearest competitor, HyflowScala, by as much as 6 times in certain cases. Other competitors such as GenRSTM, DecentSTM, and HyflowJava perform much worse in comparison to HyflowCPP. We attribute HyflowCPP's high performance to its efficient ZeroMQ-based networking support and optimized C++ implementation.

Our experiments on checkpointing – a uniquely distinguishing feature of HyflowCPP – revealed up to 100% performance improvement over flat nesting and up to 50% improvement over closed nesting. The maximum performance gains were observed for the Bank macro-benchmark and the Hash-table micro-benchmark. We observed that the performance gain improves with increase in the number of inner transactions (i.e., increase in the number of checkpoints) at significant levels of contention. The performance gain diminishes as the contention level increases. With configurable checkpointing support, we demonstrated that the maximum granularity for checkpointing is not useful in cases where abort rate is extremely high. To the best of our knowledge, this is the first ever DTM implementation where checkpointing provides higher performance gain in comparison to closed nesting.

Our closed nesting results endorsed previous studies. Our closed nesting implementation outperformed flat nesting by as much as 90

Our open nesting experiments also confirmed the trends of previous studies. Additionally, they showed higher improvements. Open nesting was found to achieve up to 140% performance improvement over flat nesting and over 90% performance improvement over closed nesting. The maximum performance gain was observed for the Hash-table benchmark. However, open nestings performance gains were found to decrease with increase in read-set sizes at high inner transaction count for benchmarks such as Skip-list and Binary search tree. This phenomenon can be explained by object caching in flat nesting and closed nesting. Object caching allows the flat nesting and closed nesting to utilize the cached object copy opened in the previous inner transactions.

HyflowCPP is publicly available at hyflow.org for programmers who are interested in developing distributed applications using our DTM framework.

7.1 Future Work

Several directions exist for future work. Immediate directions include extending HyflowCPP to support conditional transactional synchronization and irrevocable transactions [95]. Conditional synchronization will enable programmers to properly coordinate thread accesses for shared objects, without unnecessarily enforcing serialization of thread execution. It can very useful to develop complex event driven application efficiently. Similarly, irrevocable transactions support will allow programmers to perform actions whose side effects either cannot be rolled back or are expensive to roll back, which is one of the limitation of current TM systems.

HyflowCPP can also be extended to support replication-based concurrency protocols such as the quorum-based DTM protocol in [101], which enables fault-tolerant DTM. Current network module of HyflowCPP is optimized for peer-to-peer communication design. For efficient communication in a cluster environment, a new network module with multi-cast support can be plugged in HyflowCPP. For concurrency control algorithm such as tree-quorum HyflowCPP already provides pluggable support.

Near term (or longer-term) future directions include integrating HyflowCPP into production application systems and empirically characterizing performance. Example such production systems include Phonix++ [7], CloudStore [2], Boost.MapReduce [1], and Sector/Sphere [8]. In such products, the lock-based synchronization can be replaced with atomic section support provided by HyflowCPP. Such implementations can be highly useful to understand the performance gains and limitation of DTM in production environment.

HyflowCPP's programming interface can also be made more programmer-transparent, especially for advanced nesting features, through compiler support. A configurable LLVM [6] or GCC [4] compiler module can be created to support atomic section based concurrency. Such module can automatically parse the different nested modules and argument dependency without any explicit user feedback. Compiler support will swiftly decrease the learning curve for new DTM programmers and reduce the incorrect usages based errors in code.

Bibliography

[1] Boost library for map-reduce. http://www.craighenderson.co.uk/mapreduce/. Accessed: 20/01/2013.

[2] Cloudstore:mapreduce support in c++. http://en.wikipedia.org/wiki/CloudStore. Accessed: 20/01/2013.

[3] Complete context control. http://www.gnu.org/software/libc/manual/html_node/System-V-contexts.html#System-V-contexts. Accessed: 20/01/2013.

[4] Gcc, the gnu compiler collection. http://gcc.gnu.org/. Accessed: 20/01/2013.

[5] Language popularity. http://www.langpop.com/. Accessed: 20/01/2013.

[6] The llvm compiler infrastructure. http://llvm.org/. Accessed: 20/01/2013.

[7] The phoenix system for mapreduce programming. http://mapreduce.stanford.edu/. Accessed: 20/01/2013.

[8] Sector/sphere:high performance distributed file system and parallel data processing engine. http://sector.sourceforge.net/. Accessed: 20/01/2013.

[9] M. Abadi, T. Harris, and M. Mehrara. Transactional memory with strong atomicity using off-the-shelf memory protection hardware. *ACM Sigplan Notices*, 44(4):185–196, 2009.

[10] S. Adve and K. Gharachorloo. Shared memory consistency models: a tutorial. *Computer*, 29(12):66 –76, dec 1996.

[11] K. Agrawal, I. Lee, J. Sukha, et al. Safe open-nested transactions through ownership. *ACM Sigplan Notices*, 44(4):151–162, 2009.

[12] M. K. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. *ACM Trans. Comput. Syst.*, 27(3):5:1–5:48, Nov. 2009.

[13] G. Alonso. Partial database replication and group communication primitives. In *Proc. European Research Seminar on Advances in Distributed Systems*. Citeseer, 1997.

[14] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ansi sql isolation levels. In *Proceedings of the 1995 ACM SIGMOD international conference on Management of data*, SIGMOD '95, pages 1–10, New York, NY, USA, 1995. ACM.

[15] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. *IEEE Transactions on Software Engineering*, 5(3):203–216, 1979. Copyright - Copyright Institute of Electrical and Electronics Engineers, Inc. (IEEE) May 1979; Language of summary - English; Pages - 203-216; ProQuest ID - 195573704; Last updated - 2011-07-20; CODEN - IESEDJ; Place of publication - New York; Corporate institution author - Bernstein, P A; Shipman, D W; Wong, W S; DOI - 1143677; 49143; 17010; IESEDJ; ISO; 00096734; 79-11756.

[16] A. Bieniusa and T. Fuhrmann. Consistency in hindsight: A fully decentralized stm algorithm. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –12, april 2010.

[17] Bill Carlson et. al. Partitioned global address space (pgas). `upc.gwu.edu/tutorials/tutorials_sc2003.pdf`, 2003.

[18] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13:422–426, July 1970.

[19] C. Blundell, E. Lewis, and M. Martin. Subtleties of transactional memory atomicity semantics. *Computer Architecture Letters*, 5(2):17–17, 2006.

[20] R. L. Bocchino, V. S. Adve, and B. 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.

[21] M. Bornea, O. Hodson, S. Elnikety, and A. Fekete. One-copy serializability with snapshot isolation under the hood. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 625–636. IEEE, 2011.

[22] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[23] N. Carvalho, P. Romano, and L. Rodrigues. A generic framework for replicated software transactional memories. In *Network Computing and Applications (NCA), 2011 10th IEEE International Symposium on*, pages 271 –274, aug. 2011.

[24] B. L. Chamberlain, D. Callahan, and H. P. Zima. Parallel programmability and the chapel language. *The International Journal of High Performance Computing Applications*, 21(3):291, 2007. Copyright - Copyright SAGE PUBLICATIONS, INC. Aug 2007; Language of summary - English; ProQuest ID - 220792714; Last updated - 2010-06-09; Place of publication - London; Corporate institution author - Chamberlain, B L; Callahan, D; Zima, H P; DOI - 1308892151; 36439401; 37378; ISAP; INODISAP0000061533.

[25] J. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, et al. Spanner: Googles globally-distributed database. *To appear in Proceedings of OSDI*, page 1, 2012.

[26] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2stm: Dependable distributed software transactional memory. In *Dependable Computing, 2009. PRDC '09. 15th IEEE Pacific Rim International Symposium on*, pages 307 –313, nov. 2009.

[27] J. Cowling and B. Liskov. Granola: low-overhead distributed transaction coordination. In *Proceedings of the 2012 USENIX conference on Annual Technical Conference*, pages 21–21. USENIX Association, 2012.

[28] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS*, pages 336–346, 2006.

[29] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: amazon's highly available key-value store. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 205–220, New York, NY, USA, 2007. ACM.

[30] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, Dec. 2004.

[31] R. Dias, J. Lourenço, and N. Preguiça. Efficient and correct transactional memory programs combining snapshot isolation and static analysis. In *Proceedings of the 3rd USENIX conference on Hot topics in parallelism (HotPar11), HotPar*, volume 11, 2011.

[32] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen. The essence of compiling with continuations. In *ACM SIGPLAN Notices*, volume 28, pages 237–247. ACM, 1993.

[33] B. Forouzan. *TCP/IP protocol suite*. McGraw-Hill, Inc., 2002.

[34] H. Garcia-Molina. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems (TODS)*, 8(2):186–213, 1983.

[35] R. Goldring. A discussion of relational database replication technology. *InfoDB*, 8:2–2, 1994.

[36] J. Gray. The transaction concept: virtues and limitations (invited paper). In *VLDB '1981: Proceedings of the seventh international conference on Very Large Data Bases*, pages 144–154. VLDB Endowment, 1981.

[37] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, PPoPP '08, pages 175–184, New York, NY, USA, 2008. ACM.

[38] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. In *ACM SIGPLAN Notices*, volume 44, pages 404–415. ACM, 2009.

[39] R. Hansdah and L. Patnaik. Update serializability in locking. In G. Ausiello and P. Atzeni, editors, *ICDT '86*, volume 243 of *Lecture Notes in Computer Science*, pages 171–185. Springer Berlin Heidelberg, 1986.

[40] R. Hansdah and L. Patnaik. Update serializability in locking. In *Proc. of International Conference on Database Theory*, volume 243, ser, 1986. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, pp. 171185.

[41] S. Harizopoulos, D. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 981–992. ACM, 2008.

[42] M. Herlihy. Apologizing versus asking permission: optimistic concurrency control for abstract data types. *ACM Trans. Database Syst.*, 15(1):96–124, Mar. 1990.

[43] M. Herlihy and E. Koskinen. Transactional boosting: a methodology for highly-concurrent transactional objects. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 207–216. ACM, 2008.

[44] M. Herlihy, V. Luchangco, M. Moir, and I. William N. Scherer. Software transactional memory for dynamic-sized data structures. In *PODC*, pages 92–101, Jul 2003.

[45] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*, 1993.

[46] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. *Distributed Computing*, 20(3):195–208, 2007.

[47] M. Hill and M. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33 –38, july 2008.

[48] P. Hintjens. Ømq-the guide. *Online: http://zguide. zeromq. org/page: all, Accessed on*, 23, 2011.

[49] R. Hundt. Loop recognition in c++/java/go/scala. *Proc. Scala Days*, 2011.

[50] R. Jimenez-Peris, M. Patino-Martinez, B. Kemme, and G. Alonso. Improving the scalability of fault-tolerant database clusters. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on*, pages 477 – 484, 2002.

[51] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. Jones, S. Madden, M. Stonebraker, Y. Zhang, et al. H-store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008.

[52] B. Karlsson. *Beyond the C++ Standard Library: An Introduction to Boost*. Addison-Wesley Professional, 2005.

[53] J. Kim and B. Ravindran. On transactional scheduling in distributed transactional memory systems. In S. Dolev, J. Cobb, M. Fischer, and M. Yung, editors, *Stabilization, Safety, and Security of Distributed Systems*, volume 6366 of *Lecture Notes in Computer Science*, pages 347–361. Springer Berlin / Heidelberg, 2010.

[54] L. B. Kish. End of moore's law: thermal (noise) death of integration in micro and nano electronics. *Physics Letters A*, 305(34):144 – 149, 2002.

[55] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. *Software Engineering, IEEE Transactions on*, SE-13(1):23 – 31, jan. 1987.

[56] E. Koskinen and M. Herlihy. Checkpoints and continuations instead of nested transactions. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 160–168. ACM, 2008.

[57] C. Kotselidis, M. Ansari, K. Jarvis, M. Lujn, C. Kirkham, and I. Watson. Distm: A software transactional memory framework for clusters. In *In Proc. of the International Conference on Parallel Processing (ICPP*, pages 51–58, 2008.

[58] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[59] W. Lin. Basic timeetamp, multiple version timestamp, and tvo-phase locking. 1983.

[60] Y. Lin, B. Kemme, R. Jiménez-Peris, M. Patiño Martínez, and J. E. Armendáriz-Iñigo. Snapshot isolation and integrity constraints in replicated databases. *ACM Trans. Database Syst.*, 34(2):11:1–11:49, July 2009.

[61] K. Manassiev, M. Mihailescu, and C. 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.

[62] M. Moravan, J. Bobba, K. Moore, L. Yen, M. Hill, B. Liblit, M. Swift, and D. Wood. Supporting nested transactional memory in logtm. In *ACM Sigplan Notices*, volume 41, pages 359–370. ACM, 2006.

[63] E. Moss and T. Hosking. Nested transactional memory: Model and preliminary architecture sketches, 2005.

[64] J. Moss. Open nested transactions: Semantics and support. In *Workshop on Memory Performance Issues*, volume 28, 2006.

[65] J. Moss and B. Eliot. *Nested transactions: an approach to reliable distributed computing*. PhD thesis, Citeseer, 1981.

[66] J. Moss and A. Hosking. Nested transactional memory: model and architecture sketches. *Science of Computer Programming*, 63(2):186–201, 2006.

[67] MsgConnect. Eldos corporation. 2012. MsgConnect - cross-platform communication framework for your applications. Accessed November, 2012 at http://www.eldos.com/msgconnect/.

[68] B. Nitzberg and V. Lo. Distributed shared memory: a survey of issues and algorithms. *Computer*, 24(8):52 –60, aug. 1991.

[69] D. Nystrom, M. Nolin, A. Tesanovic, C. Norstrdm, and J. Hansson. Pessimistic concurrency control and versioning to support database pointers in real-time databases. In *Real-Time Systems, 2004. ECRTS 2004. Proceedings. 16th Euromicro Conference on*, pages 261 – 270, june-2 july 2004.

[70] R. Palmieri, F. Quaglia, and P. Romano. Aggro: Boosting stm replication via aggressively optimistic transaction processing. In *Network Computing and Applications (NCA), 2010 9th IEEE International Symposium on*, pages 20 –27, july 2010.

[71] F. Pedone and A. Schiper. Optimistic atomic broadcast: a pragmatic viewpoint. *Theoretical Computer Science*, 291(1):79 – 101, 2003. ¡ce:title¿Distributed Computing¡/ce:title¿.

[72] 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.

[73] P. Romano, N. Carvalho, M. Couceiro, L. Rodrigues, and J. 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. (Invited paper).

[74] P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo. Cloud-tm: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44(2):1–6, Apr. 2010.

[75] M. Saad and B. Ravindran. Snake: control flow distributed software transactional memory. In *Proceedings of the 13th international conference on Stabilization, safety, and security of distributed systems*, SSS'11, Berlin, Heidelberg, 2011. Springer-Verlag.

[76] M. Saad and B. Ravindran. Supporting STM in distributed systems: Mechanisms and a Java framework. In *Proceedings of the 6th ACM SIGPLAN Workshop on Transactional Computing*, June 2011.

[77] M. Saad and B. Ravindran. Transactional forwarding algorithm. *ECE Dept., Virginia Tech, Tech. Rep*, 2011.

[78] M. M. Saad and B. Ravindran. Hyflow: a high performance distributed software transactional memory framework. In *Proceedings of the 20th international symposium on High performance distributed computing*, HPDC '11, pages 265–266, New York, NY, USA, 2011. ACM.

[79] B. Saha, A.-R. Adl-Tabatabai, R. L. Hudson, C. Cao Minh, and B. Hertzberg. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP '06*, pages 187–197, Mar 2006.

[80] R. Schaller. Moore's law: past, present and future. *Spectrum, IEEE*, 34(6):52–59, 1997.

[81] F. B. Schneider. Distributed systems (2nd ed.). chapter Replication management using the state-machine approach, pages 169–197. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[82] N. Shavit and D. Touitou. Software transactional memory. In *PODC*, pages 204–213, 1995.

[83] S. Sridharan, J. Vetter, B. Chamberlain, P. Kogge, and S. Deitz. A scalable implementation of language-based software transactional memory for distributed memory systems. Technical report, Tech. Rep, 2011.

[84] S. Sridharan, J. Vetter, and P. Kogge. Scalable software transactional memory for global address space architectures. Technical report, Technical report, ORNL FT Technical Report Series, 2009.

[85] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. 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.

[86] X.-H. Sun and Y. Chen. Reevaluating amdahls law in the multicore era. *Journal of Parallel and Distributed Computing*, 70(2):183 – 188, 2010.

[87] S. E. Thompson and S. Parthasarathy. Moore's law: the future of si microelectronics. *Materials Today*, 9(6):20 – 25, 2006.

[88] A. Turcu and B. Ravindran. Hyflow2: A high performance distributed transactional memory framework in scala. Technical report, Technical report, Virginia Tech, April 2012. URL http://hyflow. org/hyflow/chrome/site/pub/hyflow2-tech. pdf.

[89] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *Proceedings of the 5th Annual International Systems and Storage Conference*, page 12. ACM, 2012.

[90] A. Turcu and B. Ravindran. On open nesting in distributed transactional memory. In *SYSTOR '12: Proceedings of the 5th Annual International Systems and Storage Conference*, pages 1–12, New York, NY, USA, 2012. ACM.

[91] A. Turcu, B. Ravindran, and M. Saad. On closed nesting in distributed transactional memory. In *Seventh ACM SIGPLAN workshop on Transactional Computing*, 2012.

[92] W. Vogels. Eventually consistent. *Commun. ACM*, 52(1):40–44, Jan. 2009.

[93] W. E. Weihl. Local atomicity properties: modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–282, Apr. 1989.

[94] G. Weikum. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems (TODS)*, 16(1):132–180, 1991.

[95] A. Welc, B. Saha, and A. Adl-Tabatabai. Irrevocable transactions and their applications. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 285–296. ACM, 2008.

[96] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Distributed Computing Systems, 2000. Proceedings. 20th International Conference on*, pages 464 –474, 2000.

[97] T. Willhalm and N. Popovici. Putting intel® threading building blocks to work. In *Proceedings of the 1st international workshop on Multicore software engineering*, pages 3–4. ACM, 2008.

[98] B. Zhang and B. Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS '09: Proceedings of the 13th International Conference on Principles of Distributed Systems*, pages 48–53, Berlin, Heidelberg, 2009. Springer-Verlag.

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

[100] B. Zhang and B. Ravindran. Dynamic analysis of the Relay cache-coherence protocol for distributed transactional memory. In *IPDPS '10: Proceedings of the 2010 24th IEEE International Parallel and Distributed Processing Symposium*, Washington, DC, USA, 2010. IEEE Computer Society.

[101] B. Zhang and B. Ravindran. A quorum-based replication framework for distributed software transactional memory. In *Proceedings of the 15th international conference on Principles of Distributed Systems*, OPODIS'11, pages 18–33, Berlin, Heidelberg, 2011. Springer-Verlag.