HyflowCPP: A Distributed Transactional Memory Framework for C++

Sudhanshu Mishra, Alexandru Turcu, Roberto Palmieri, Binoy Ravindran
ECE Department, Virginia Tech, Blacksburg, VA, 24061
{mishras, talex, robertop, binoy}@vt.edu

***Abstract*—We present the first ever distributed transactional memory (DTM) framework for distributed concurrency control in C++, called *HyflowCPP*. HyflowCPP provides distributed atomic sections, and pluggable support for policies for concurrency control, directory lookup, contention management, and networking. While there exists other DTM frameworks, they mostly target VM-based languages (e.g., Java, Scala). Additionally, HyflowCPP provides uniquely distinguishing TM features including strong atomicity, closed and open nesting, and checkpointing. Our experimental studies revealed that HyflowCPP achieves up to 6x performance improvement over state-of-the-art DTM frameworks.**

I. Introduction

Transactional systems based on Software Transactional Memory (STM) are increasingly becoming a promising technology for designing and implementing transactional applications. STM is a framework for managing concurrent operations on a shared data set. With STM, programmers organize code that read/write in memory elements (e.g., memory chunks or object pointers) as *memory transactions*, and the framework transparently ensures transactional properties such as atomicity, consistency, and isolation [1]. The transactional abstraction thus allows programmers to easily implement concurrent applications by simply enclosing parts of the code accessing shared data in transactions, without being burdened with the mechanisms needed to ensure transactional properties. Moreover, relying on STM means that, programmers do not have to cope with deadlocks, livelocks, lock convoying, etc., which are the typical pitfalls when synchronization mechanisms are manually implemented. As a result, code reliability increases and development time is shortened.

The nature of STM in memory transactions results in transaction execution times that are several orders of magnitude smaller than that in conventional transactional systems [2]. This allows STM systems to simultaneously service a massive number of concurrent requests. In fact, STM's popularity has grown with the widespread adoption of multi-core architectures, simply because these architectures can naturally support an increased level of application parallelism.

Distributed STM is a logical way to exploit STM's advantages in distributed settings i.e., systems with nodes interconnected using message passing links. In particular, pitfalls of manual, lock-based distributed synchronization (e.g., distributed deadlocks, distributed livelocks) are orders of magnitude harder to debug than those in a multiprocessor setting. DTM's (distributed) transactional abstraction is therefore a compelling distributed synchronization abstraction. Additionally, DTM protocols are increasingly becoming a way to target scalability due to their superior performance in comparison with manually implemented distributed locks [3], [4], [5].

Most existing DTM frameworks are prototyped on top of VM-based programming languages (e.g., Java, Scala): ClusterSTM [6], D2STM [7], DiSTM [8], GenRSTM [9], HyflowJava [3], and HyflowScala [10]. Some DTM implementations have been developed for C and C++: DMV [11], ClusterSTM [6], and Sinfonia [12]. However, DMV and ClusterSTM are implemented as proofs-of-concept and Sinfonia is designed as a backup and restore service, rather than as DTM frameworks.

C++ is one of the most popular programming languages for high performance systems. In the last decade a number of protocols for DTM have been proposed, and some of them have been developed. The languages usually adopted for their implementation are based on *virtual machine* (e.g., Java, Scala). The reason is typically related to the current technological trend. In fact, in order to easily integrate such systems with benchmarks, applications and other sub-systems recently implemented, the designers adhere to the employment of these programming languages. Even though the support for a transparent integration with other systems is a critical feature for several products, their performance still certainly represents the core comparison point among different possible solutions.

Lower level programming languages, like C++, enable the programmer to exploit architectural advantages that could bias system performance. This is especially true in transactional applications based on in memory operations like STM or DTM in which the overhead for layering the framework, ensuring information hiding and wrapping methods typically results in more complex execution flow composed by several steps that significantly impact the overall performance. This is the reason many high-performance production systems are developed in C++, instead of VM-based languages, usually, to overcome performance issues inherent in VM-based languages [13].

In addition, memory management in C++ is manual. Even though this results in a more complex implementation of transactional framework, it potentially eliminates garbage collection overheads of VM-based environments. Moreover, considering the absence of I/O interactions in the STM transactions processing, having the direct control of memory management allows specific optimization unfeasible with VM-based languages. Therefore, a C++ framework for DTM is highly desirable as that enables DTM support for a popular language, and provides high-performance distributed concurrency control for C++ applications with high programmability.

Motivated by these observations, we design and implement HyflowCPP, the first ever DTM framework for C++.

HyflowCPP has a modular architecture and provides pluggable support for various DTM policies (e.g., concurrency control, directory lookup, contention management, networking) and a simple atomic section-based DTM interface. The distributed concurrency control currently implemented in the framework is TFA [14]. TFA uses Herlihy and Sun's dataflow execution model [15] (in which objects migrate and transactions are immobile). The adoption of TFA as a baseline protocol for managing concurrent requests allows a fair comparison of HyflowCPP against the same TFA implementation in two VM-based DTM frameworks such as HyflowJava [3] and HyflowScala [16], highlighting the actual benefit of the lower level programming language. Furthermore, HyflowCPP embeds support for using closed [17] and open [18] nested transactions and checkpointing [19]. To the best of our knowledge, no past DTM frameworks in C++ support all these 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. The key result from our experimental studies is that, HyflowCPP outperforms its nearest competitor, HyflowScala, by a factor of 6. Other competitors, including GenRSTM, DecentSTM, and HyflowJava, perform even worse in comparison to HyflowScala.

Additionally, we conducted an evaluation contrasting the two typical approaches for implementing partial rollback of transactions as a way to minimize the time to retry a transaction after its abort, namely checkpointing and closed-nesting. The experiments reveal that, unlike in [20] where closed-nesting is always better than checkpointing due to the cost of saving and restoring memory images, in HyflowCPP, exploiting the efficient mechanism for saving execution states without incurring significant overheads [21], the checkpointing technique ensures better performance. Generally, we show that HyflowCPP's checkpointing 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 [17], [22], 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 limited overheads of various mechanisms in C++ [13].

The rest of the paper is organized as follows: Section II describes TFA, transaction nesting and checkpointing, which represent the background for this work. Section III details the programming model of HyflowCPP and Section IV describes HyflowCPP's architecture. We report our experimental results in Section V, and conclude in Section VI.

II. Background

In this section we provide a brief introduction to TFA, the base protocol implemented in HyflowCPP. We then proceed to describe the different transaction nesting models and transaction checkpointing.

A. TFA Protocol

Transactional Forwarding Algorithm (TFA) [14] is a lock-based DTM algorithm with lazy lock acquisition and buffered writes. All read objects are stored in a local *read-set*, so all reads can later be revalidated. Written objects are also stored in a local *write-set*.

TFA uses a variant of the Lamport clocks mechanism to establish "happens before" relationships across nodes. Each distributed node has a local clock lc and atomically increments its local clock on the commit of every transaction that changes the shared state (write transactions). All messages sent between nodes piggyback the local clock value of the sender node. Upon receiving such a message, each node compares the remote clock value included in the message with its own local clock. If the remote value is greater, the local clock is updated to this greater value.

Each transaction records the local clock value lc at the time it starts (i.e., starting clock, sc). Then, when it communicates with remote nodes (for the purpose of accessing new objects), it compares the clock value of the remote node (rc) to its own start clock (sc). If $rc > sc$, the transaction undergoes a Transactional Forwarding procedure: it validates its read-set, and, should that be successful, updates its starting time to $sc = rc$. If the validation fails the transaction aborts. Validation is performed by comparing the object's latest version with the current transaction's start clock.

B. Nested Transactions and Checkpointing

Transactions are nested when they appear within another transaction's boundary. Transaction nesting makes code composability easy: multiple operations inside a transaction will be executed atomically, regardless of whether the said operations contain transactions or not, and without breaking encapsulation. This is an important advantage of transactional memory when compared to traditional lock-based synchronization.

Three transactional nesting models were proposed in the literature: Flat, Closed [23] and Open [24], [23].

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 parent transaction. Aborting any inner-transaction causes the parent transaction to abort. Clearly, flat nesting does not provide any performance improvement over non-nested transactions.

Closed nesting allows inner-transactions to abort individually. Aborting an inner-transaction does not necessarily lead to also aborting the parent transaction (i.e., partial rollback is possible). However, inner-transactions' commits are not visible outside the parent transaction. An inner-transaction commits its changes only into the private context of its parent transaction, without exposing any intermediate results to other transactions. Only when the parent transaction commits, the shared state is modified.

Open nesting considers the operations performed by sub-transactions at a higher level of abstraction, in an attempt to avoid false conflicts occurring at the memory level. It allows inner-transactions to commit or abort individually, and their commits are globally visible immediately. In case an enclosing transaction aborts, due to any fundamental conflicts (i.e., not false) at the higher levels of abstraction, all the inner-transactions are roll-backed by using compensating actions, which are predefined for each abstract operation.

Checkpointing [25], [26] addresses this issue by allowing execution to return to any previously saved state (*checkpoint*) within the current transaction, regardless of whether the sub-transaction encompassing that checkpoint is still active or not. This allows developing a very fine grained partial rollback mechanism, which can identify the exact operation to rollback execution to, in order to resolve the current conflict. On abort, a checkpoint that can resolve the conflict is located and activated, effectively reverting transaction execution to the state it had at the time the checkpoint was originally taken. The program control flow is managed by saving and restoring the thread's execution state (i.e., CPU registers and activation stack) and employs a mechanism called *continuations* [27].

III. Programming Interface

Since objects are dispersed over nodes in a distributed setting and normal object references cannot be used, we provide a base class called *HyflowObject*, which must be inherited by each object. *HyflowObject* provides a `getId()` method, which returns a unique key to access the object from anywhere in the network. HyflowCPP serializes/de-serializes objects using the Boost serialization library [28]. Each extended HyflowObject field is registered with the Boost serialization function, for serialization/de-serialization over the network. HyflowCPP provides two transactional interfaces: macros and atomic classes.

A. Transaction Support using Macros

HYFLOW_ATOMIC_START, HYFLOW_ATOMIC_END. These macros are provided to define atomic sections. Any object can be opened in the *Read* or *Write* mode. Once an object is requested, HyflowCPP fetches the object from its current location and copies it to the transaction read-set or write-set depending upon the access type. Objects are accessed using the macro HYFLOW_FETCH.
HYFLOW_ON_READ, HYFLOW_ON_WRITE. These macros are used for reading or writing, respectively, the fetched objects. HYFLOW_ON_READ returns a constant reference pointer to HyflowObject. HYFLOW_ON_WRITE returns a non constant object reference pointer. The constant reference pointer can also be used in place of unique object key to retrieve an object in write mode.
HYFLOW_PUBLISH_OBJECT. This macro is used for publishing a locally created object in the distributed directory.
HYFLOW_PUBLISH_DELETE This macro is used to publish a cancellation of an object in the distributed directory.
HYFLOW_CHECKPOINT_INIT. This macro is used for initiating the checkpointing environment.
HYFLOW_CHECKPOINT_HERE. This is used for creating a checkpoint of transactional execution.
HYFLOW_STORE. The macro is used by the programmer to save any primary data structure object on the stack or the heap. For heap objects, the programmer is responsible for creating object copies and managing memory; the macro only saves the address value. The macro's arguments include the variable reference and value, and it automatically restores the saved value when the transaction resumes from the selected checkpoint.

Figure 1 reports the implementation of *add* and *remove* operations on a Linked List using the presented HyflowCPP macros. Figure 2 shows how checkpointing can be implemented in a transaction (part of Bank benchmark). Note that

```
class ListNode::HyflowObject {
...
void ListNode::addNode(int value){
  HYFLOW_ATOMIC_START{
     std::string head="HEAD";
     HYFLOW_FETCH(head, false);
     ListNode* headNRead=(ListNode*)
          HYFLOW_ON_READ(head);
     std::string oldNext=headNRead->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{
     std::string head("HEAD");
     std::string prev=head,next;
     HYFLOW_FETCH(head, true);
     targetN=(ListNode*)HYFLOW_ON_READ(head);
     next=targetN->getNextId();
     while(next.compare("NULL") != 0){
       HYFLOW_FETCH(next,true);
       targetN=(ListNode*)HYFLOW_ON_READ(next);
       int nodeValue=targetN->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=targetN->getNextId();
     }
  } HYFLOW_ATOMIC_END;
}
}
```

Fig. 1. The implementation of add and remove operations in HyflowCPP.

```
void BankAccount::transfer(Acc1,Acc2,Money){
    HYFLOW_ATOMIC_START{
        HYFLOW_CHECKPOINT_INIT;
        withdraw(Acc1,Money,__context__);
        HYFLOW_CHECKPOINT_HERE;
        deposit(Acc2,Money,__context__);
    }HYFLOW_ATOMIC_END;
}
```

Fig. 2. Checkpointing in a Bank transfer function

the current context instance must be passed to the withdraw function. This requirement exists for all functions which are called within the atomic block and are required to be executed atomically.

B. Transaction Support using Atomic class

Using the `Atomic` class interface, the transaction context can be directly manipulated and any function can be executed atomically. The programmer can assign a desired function pointer value to the `atomically` function pointer of the `Atomic` class instance. Later, the `execute` method in the

`Atomic` class instance can be called to execute the desired function atomically. The `Atomic` class also provides function pointers to support nesting features like open nesting. Specifically, the `onCommit` and `onAbort` functions are specified for the *HyflowObject* requiring open nesting support.

IV. System Architecture

Figure 3 shows HyflowCPP's architecture of a transactional node. It is made of six modules. The transaction interface module is summarized in Section III. Here, we will focus on a subset of the rest of the modules.

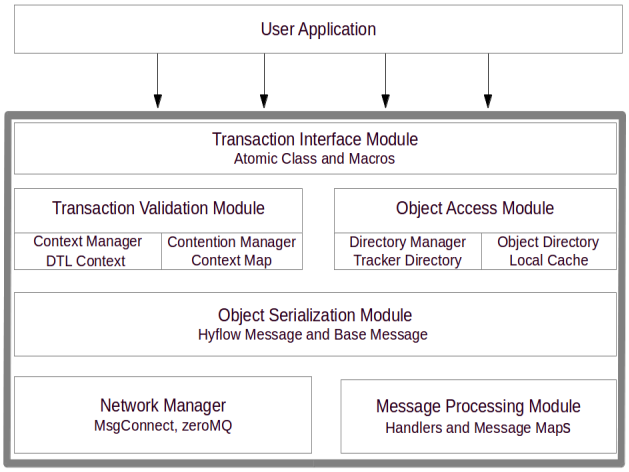


Fig. 3. HyflowCPP's node's architecture

Transaction Validation Module.

The entire concurrency control logic is performed in the *Transaction Validation Module* (TVM) by extending the base class *HyflowContext*. By default, *HyflowContext* is extended as *DTLContext*, which is responsible for providing the implementation of the concurrency control algorithm (i.e., TFA). *DTLContext* is an interface offering all the calls needed for implementing its own concurrency control protocols. Extending this class, the programmer can easily integrate a new contention manager in the FyFlowCpp. TVM validates memory locations and retries the transactional code if needed upon commit or abort. The module can also be configured for supporting checkpointing, closed and open nesting.

This module also provides the data structures needed for managing all the transactional contexts (i.e., the meta-data associated to the on-going transactions). The most relevant structures are: the *lock-table*, used for object-level or word-level locking (implemented using the concurrent hash-map of TBB library [29]); the *context-map*, a concurrent hash table providing access to transactional contexts indexed by transaction ID. The latter structure enables the *contention manager* to manage meta-data and to make updates to different transactional contexts.

Object Access Module

The Object Access Module (OAM) provides facilities to manage distributed shared objects and to interact with the distributed objects directory for changing the object ownership, performing remote validation and updating the directory itself. Objects are located using the unique object ID. The module encapsulates a directory lookup protocol to access distributed objects.

The OAM contains two efficient concurrent hash-maps: *local cache* and *object directory*. The *local cache* maintains authoritative copies of objects owned by the current node, and the *object directory* maintains the meta data information (e.g., object owner information in the case of tracker directory). Similar to the TVM, the programmer can rely on one or both of these structures for implementing the object location service. By default, it implements the efficient tracker directory protocol, which moves an object across nodes and maintains the current owner information on a specific tracker node. This is because TFA requires an object directory for changing the object ownership when transactions commit.

OAM closely interacts with the TVM. In fact, to provide strong atomicity, HyflowCPP directs all accesses to objects through the *context-map* managed by TVM. Therefore, all object requests to the OAM go through the TVM. Object deletion or publication requests made by the TVM are served in the OAM. This module also handles object validation requests.

Object Serialization Module

To free programmers from message serialization and de-serialization, HyflowCPP provides two classes: *HyflowMessage* and *BaseMessage*. *BaseMessage* acts as a parent class for any message in HyflowCPP; it can be extended to create any new message type to perform a protocol-specific operation. *HyflowMessage* provides a standard interface for networking. All *HyflowMessage* objects are converted in a binary blob, before forwarding to the network library, to communicate over the network. In this way, network implementation is independent of the transaction validation module and the object access module. Serialization and de-serialization of messages is done using Boost [28].

Network Module

This module (NM) provides pluggable support for a network library. Providing efficient network interactions is a fundamental for distributed protocol managing transactions executed directly in memory. This is because the transaction execution time is dominated by network time. Relying on an high performance communication layer significant reduces the transaction response time.

HyFlowCPP provides two networking libraries: *MsgConnect* [30] and *ZeroMQ* [31]. In the current release, in order to design a fast and scalable networking solution, we use industry standard *ZeroMQ* library. It is a socket level library, providing very efficient solutions for in-process communication between threads, which makes it suitable for any multi-threaded networking requirement. Figure 4 illustrates our networking architecture designed using the *ZeroMQ* library. Any two transactional nodes A and B communicate with each other using the *forwarder* and *catcher* networking threads. These threads are responsible for connecting with different nodes using the *ZeroMQ* router socket. Forwarder threads receive message requests from transactional threads and forward them to catcher threads of the target nodes, which assign them to available *worker* threads. Worker threads process the message and send back reply to catcher threads

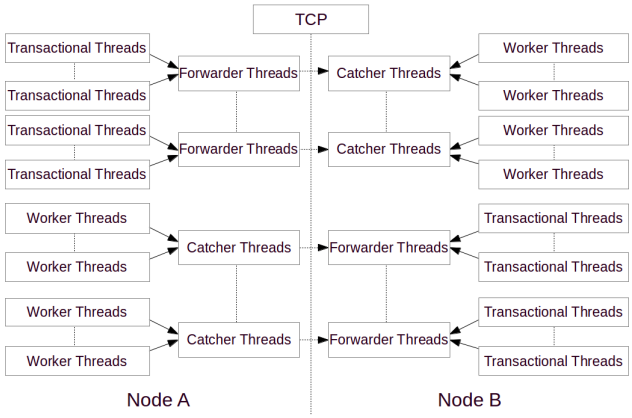


Fig. 4. HyflowCPP's (ZeroMQ-based) network architecture

if required. Catcher threads return the messages back to the forwarder threads, which route them to the original requester transactional threads.

Our studies revealed that *ZeroMQ* provides 4 to 5 times better performance over *MsgConnect*. In addition, using multi-part messaging of *ZeroMQ*, we are able to reduce the amount of polling between threads to a bare minimum at two sockets.

Message Processing Module

This module (MPM) provides message handling capability to the NM. When a programmer creates a new message type, a message handler is also created for that message and registered using the message handler interface. Later, when the NM receives a request message, MPM creates a proper response using the message handler, and replies back as required.

The module also enables asynchronous messaging using a class *HyflowMessageFuture*, which allows a transactional thread to send a message asynchronously, proceed with other tasks, and later wait for the message response.

V. Implementation and Experimental Evaluation

A. Implementation

HyflowCPP has been implemented from scratch adopting technologies and design choices for making the framework general and optimized for in memory transactions. The implementation of flat nesting execution model (i.e., plain transactions without nesting) follows by the TFA protocol rules. Additional mechanisms are needed for implementing nesting supports.

Regarding the closed-nesting supports, for each transactional context, we maintain a reference to its parent transaction context. In the commit phase, we directly merge the context objects with the parent context objects for inner-transactions. The commit procedure of outermost transactions is the same as that in flat nesting.

To support the open open-nesting model, we use abstract locks as higher-level locks for inner-transactions. Further, we allow programmers to define *onAbort* and *onCommit* functions, which are to be performed in case of abort or commit of the parent transaction. Abstract locks are acquired in inner-transactions and released in the parent transaction, on commit or abort. Each inner-transaction commits as a pure flat-transaction, releasing its transactional context, included read-set and write-set.

The implementation of checkpointing model is the one that takes more advantages from the manual memory management allowed by C++ language. We partition the read-set, write-set, publish-set and the delete-set objects on the basis of the first access checkpoint. This allows us to identify the conflicting objects and calculate the dependency to determine the valid checkpoint. To save the transaction execution stack state, we use the *setContext* and *getContext* functions. Heap objects are maintained in the context read-set, write-set, and publish-set. HyflowCPP provides a helper class *CheckPoint-Provider* to create, iterate and maintain transaction checkpoints. *HYFLOW_STORE* macro is provided to store any stack or heap object values just before creating a checkpoint (see Section III). The values are automatically restored upon continuing from a checkpoint after partial abort. On commit failure, instead of restarting the transaction, we resume from an available valid checkpoint. Also, all acquired locks are released before resuming.

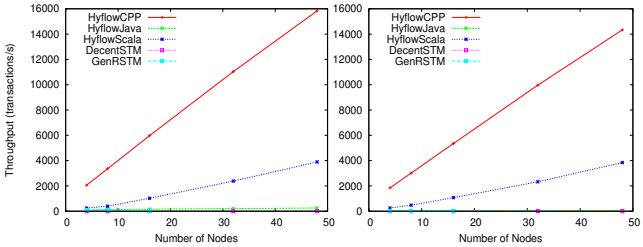
B. Experimental Evaluation

In this section we describe the evaluation results of HyflowCPP compared with the other state of the art JVM-based DTM systems. Being the first ever DTM framework in C++, we cannot compare it with other C++ based DTMs, therefore we evaluated HyflowCPP by comparing with other JVM-based DTM frameworks. We firstly contrasted our HyflowCPP with two state of the art DTM frameworks: GenRSTM [9] and DecentSTM [32]. Subsequently, for the rest of the evaluation, we directly compared HyflowCPP with its JVM-based counterparts: HyflowJava [3] and HyflowScala [10]. We excluded the initial competitors from the comparison because an extensive evaluation has already been presented in [14] where HyflowJava has been shown to outperform both GenRSTM and DecentSTM.

Our experiments were conducted using up to 48 nodes located in a private cluster and interconnected by a Gigabit connection, running two application threads per node. We use the Ubuntu Linux 10.04 server OS. Our evaluation workloads included both micro-benchmarks and macro-benchmarks. Regarding the former, we used the distributed version of: Linked List, Skip List, Binary Search Tree and Hash Table. Regarding the latter we used Bank, Loan and a distributed version of the STAMP's Vacation [33]. For the micro-benchmarks, each transaction consists of several operations on the shared data-structure. For the macro-benchmarks: Bank simulates a banking system; Loan simulates a mortgage lending setting; Vacation simulates an itinerary planning and reservation system. We divided the evaluation in three subsections corresponding to the three transaction execution models supported by HyflowCPP: flat nesting, closed-nesting/checkpointing, open-nesting. Each data-point in the plots is the average of 3 repeated experiments. Due to space constraints we cannot present all the plots. The complete evaluation study can be found in [34].

Flat Nesting

Experiments with flat transactions allow us to understand the raw performance of different DTM frameworks. In Figure 5 the comparison between HyflowCPP and other JVM-based



(a) Bank 80% read (b) Bank 20% read

Fig. 5. Comparison between HyflowCPP and GenRSTM, DecentSTM, HyflowJava, HyflowScala (20% and 80% read workload)

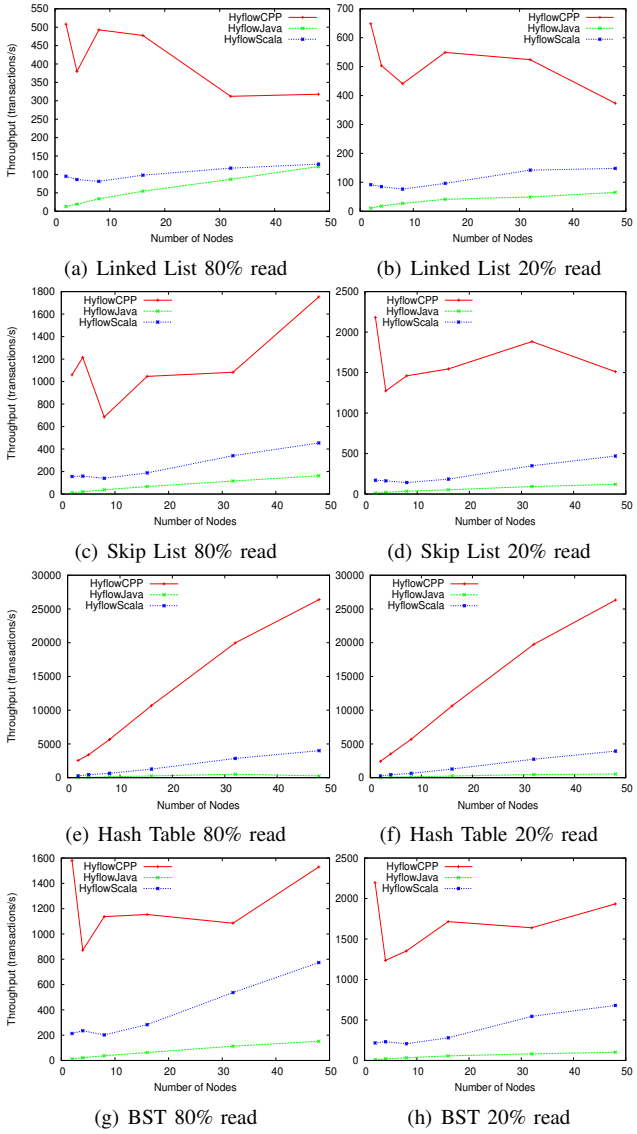
DTM frameworks is presented. The plots clearly show the performance gap between HyflowCPP and other competitors. As mentioned at the begin of this section, in the rest of the evaluation HyflowJava and HyflowScala have been used as competitors.

Micro-benchmarks. For Linked List, Skip List, and BST, 50 objects are deployed; for Hash Table, 2000 distributed buckets were used. Objects and buckets are uniformly distributed over nodes. Each benchmark has been configured producing two different workloads, one read intensive, characterized by 80% of read-only transactions, and the second with 20% of read-only transaction, therefore mostly write intensive.

Figure 6 shows the performance of HyflowCPP against HyflowJava/Scala. Linked List and Skip List are affected by a number of conflicts due to the reduced number of objects available in the system. This scenario mainly stresses the mechanisms needed for implementing the distributed concurrency control. Due to the smaller read-write set size, Hash Table's throughput is highest among all the benchmarks. Linked List's throughput is lower than Hash Table's due to the high abort rate caused by the large read-set. Conversely, the Skip List's performance are better than Linked List due to smaller read set size. BST exhibits high level of concurrency; its performance is up to 2× better in comparison to Linked List.

The plots reveal that HyflowCPP performs up to 3 to 5× better than its nearest competitor HyflowScala. HyflowJava is worse than HyflowScala. We recall that all the DTMs tested implement TFA. Here the HyflowCPP's high throughput can be mainly attributed to the fast network layer, obtained from the higher quality implementation with optimized load balancing, synchronization and packet handling, which yields lower queuing delays, response times, and smaller TCP connection re-initiations. In addition, we observed average CPU time utilization for HyflowScala to be around 20%-30%, whereas HyflowCPP utilized 50%-60%, which supports our argument regarding the minimized CPU idle time due to the optimized network management layer.

Macro-benchmarks. The analysis with macro-benchmarks are useful for understanding DTM performance in a more "real world" settings. Their transactions are composed by several operations leading to high messaging and transaction execution time. Due to their characteristics, such benchmarks are appropriate to analyze possible bottlenecks in terms of messaging, synchronization or memory usages in the framework. Unfortunately, HyflowScala does not provide the implementation of Vacation and Loan, therefore HyflowCPP has been contrasted with the HyflowJava for Vacation and Loan. We configured the benchmarks with a total of 10,000 accounts/objects.



(a) Linked List 80% read (b) Linked List 20% read

(c) Skip List 80% read (d) Skip List 20% read

(e) Hash Table 80% read (f) Hash Table 20% read

(g) BST 80% read (h) BST 20% read

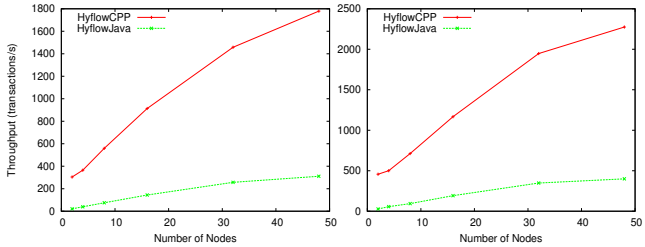
Fig. 6. Performance of micro-benchmarks with (20% and 80% read workload)

Figures 7 and 5 show results for 20% and 80% read ratio. HyflowCPP, in general, performs up to 3 to 5× better than competitors (5 to 10× better for Loan). It is relevant to notice the scalability trend of HyflowCPP, which can be approximated as linear. Conversely, the others performance after 20 or 30 nodes (depending on the benchmark) stall, showing a very limited scalability compared to our proposal.

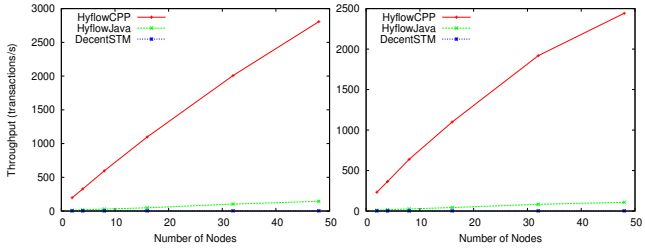
The plots clear reveal the positive impact of the optimization allowed by the manual memory and network management of C++ instead of JVM-based implementation.

Checkpointing and Closed Nesting

Closed nesting and checkpointing are two techniques for implementing partial rollback after an abort is issued. A performance comparison between these two models also appeared in [20].This paper assesses that supporting saving and restoring of large memory chunks is more costly than running closed-nested transactions, electing closed-nesting as reference model for implementing partial abort. The goal of this study is to show how HyflowCPP can subvert the latter decision rein-

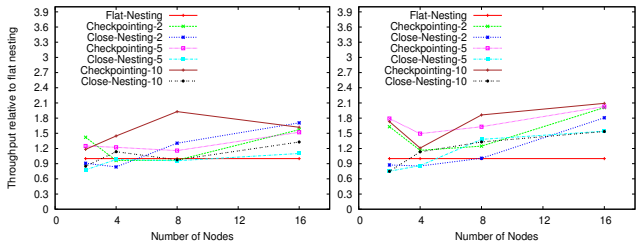


(a) Vacation (80% reads) (b) Vacation (20% reads)

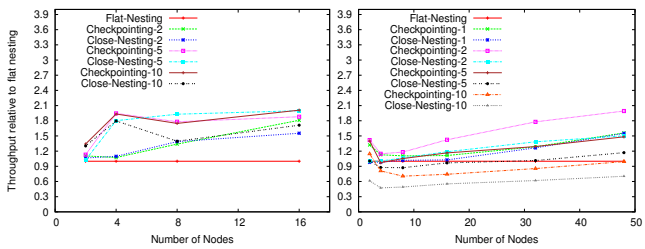


(c) Loan (80% reads) (d) Loan (20% reads)

Fig. 7. Transactional throughput of macro-benchmarks



(a) List (b) Skip List



(c) Hash Table (d) Bank

Fig. 8. HyflowCPP's performance of closed-nesting and checkpointing

forcing checkpointing as the better way to restore transaction execution from a safe point. Indeed, no JVM-based competitors are included.

To understand checkpointing and closed nesting's benefits over flat nesting, we conducted experiments on micro-benchmarks and Bank, because other macro-benchmarks do not define rules for splitting transactions in sub-transactions. For the checkpointing experiments, we inserted a checkpoint just before accessing an object (or a subset of objects according to the selected checkpointing granularity). For the closed nesting experiments, we performed each operation as an inner-transaction. In order to do a fair comparison, we configured the benchmarks to perform 20% of read-only transactions. In this way we increase the conflict probability, therefore the likelihood to incur an abort. In the plots we varied the node count $(2-16)$ and the inner-transactions/checkpoints count $(2, 5, 10)$ to understand the impact of transactional length and partial rollback points on the performance. Increasing the number or inner-transactions/checkpoints allows a more accurate conflict resolution and an efficient transaction restart.

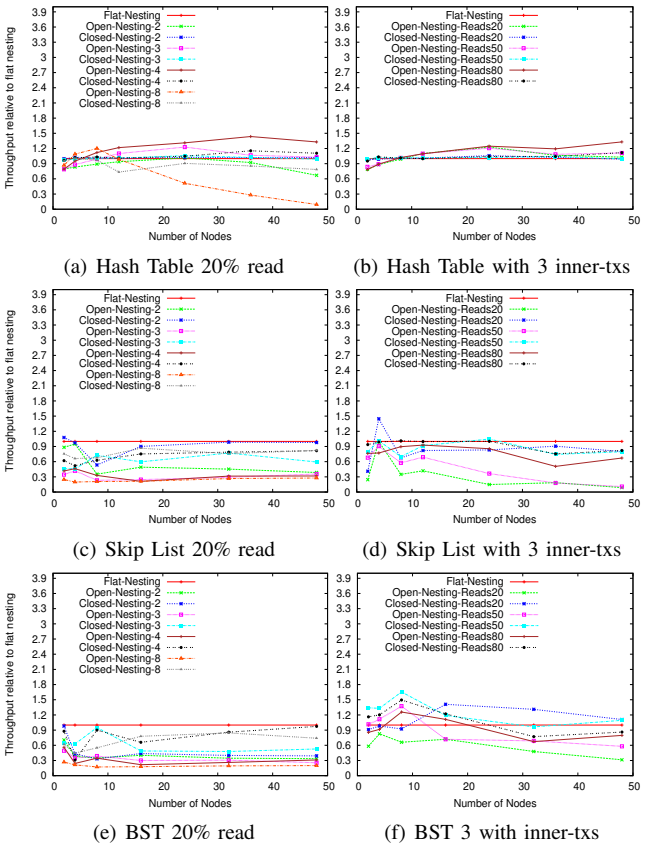
Figure 8 shows checkpointing and closed nesting's throughput relative to flat nesting. Both outperform flat nesting by up to 100%. Also, as the inner-transactions count increases, checkpointing and closed nesting's performance improve, due to increase in partial rollback points. In all the experiments, checkpointing guarantees better performance than closed-nesting. The efficient mechanism for saving and restoring the transaction's context allows HyflowCPP to catch the exact operation that caused the abort, minimizing the time spent by the transaction to re-execute code that is still consistent, and quickly restore the valid memory image. This result has been obtained relying on C++ features not available in other JVM-based languages in which, closed-nesting is unquestionably preferred to checkpointing due to the resulting worse performance of the latter.

Open Nesting

To understand open nesting's benefits, we conducted experiments on Hash Table, Skip List and BST. Open-nested transactions were created similar to closed-nested using multiple read/write operations performed as inner-transactions, but with an additional abstract lock mechanism. In the evaluation we contrasted the performance of open-nesting against closed-nesting, reporting their throughput improvement with respect to flat nesting. For each benchmark two plots are reported. In both we fixed the number of shared objects in the system and we varied the number of processing nodes $(2-48)$. Then, in the first plot we set the percentage of read-only transactions to 20%, generating an high conflict scenario, and we varied the number of inner-transactions $(2, 3, 4, 8)$. In the second plot we fixed the number of nested transactions to 3, and we varied the workload moving the read ratios (20%, 50%, 80%).

Figure 9(a) shows open and closed nesting's throughput relative to flat nesting configuring the shared hash table with 300 buckets. Initially, at small node count, contention is low, and flat and open nesting have similar performance. However, as the node count and inner-transaction count begin to increase, enough partial aborts occur, degrading flat nesting's performance, while open nesting's performance remains steady due to reduced false conflicts. At very high contention, with 8 inner-transactions, open nesting's performance drops due to increased abstract lock contention. Figure 9(b) shows the performance with 3 inner-transactions per parent-transaction. Open nesting's throughput increases with increase in read ratio due to decreased contention. We recall that the cost for committing an open-nested transaction is comparable to the cost for committing a parent-transaction, therefore decreasing the data contention allows the framework to successfully commit more inner-transactions, and maximize the usefulness of adopting the open nesting model.

In Figures 9(c) and 9(d) we present the results of Skip List benchmark configured with 100 share 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 in a scenario characterized by high contention. 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 locally cached after the first few inner-transactions executed. Meanwhile, open-nested transactions are required to fetch the



(a) Hash Table 20% read (b) Hash Table with 3 inner-txs

(c) Skip List 20% read (d) Skip List with 3 inner-txs

(e) BST 20% read (f) BST 3 with inner-txs

Fig. 9. HyflowCPP performance using open-nesting transaction model.

objects accessed over network, without exploiting the read-set of previous committed open-nested transactions, which increases the execution time significantly. It is also worth mentioning that open nesting has additional messaging overhead for maintaining distributed abstract-locks. BST results confirm the same trend of Skip List and are presented in Figure 9(e) and 9(f).

VI. Conclusions, Future Work

HyflowCPP provides a generic programming interface-based DTM abstraction for C++, with pluggable support for various DTM policies, and support for closed nesting, open nesting, and checkpointing. Our experimental studies revealed that, HyflowCPP's throughput scales robustly with increase in read/write ratio, and nodes, and that it outperforms its nearest JAVA-based competitors by up to 6x.

HyflowCPP is freely available as open-source software at *http://www.hyflow.org/hyflow/wiki/HyflowCPP*.

Acknowledgements

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

References

[1] T. Harris, J. Larus, and R. Rajwar, "Transactional memory," *Synthesis Lectures on Computer Architecture*, vol. 5, no. 1, pp. 1–263, 2010.

[2] R. Palmieri, F. Quaglia, P. Romano, and N. Carvalho, "Evaluating database-oriented replication schemes in software transactional memory systems," in *Proc. of DPDNS*, 2010.

[3] M. M. Saad and B. Ravindran, "Hyflow: a high performance distributed software transactional memory framework," in *HPDC*, 2011.

[4] S. Peluso, P. Romano, and F. Quaglia, "Score: A scalable one-copy serializable partial replication protocol," in *Middleware*, 2012.

[5] S. Peluso, P. Ruivo, P. Romano, F. Quaglia, and L. Rodrigues, "When scalability meets consistency: Genuine multiversion update-serializable partial data replication," in *ICDCS*, 2012.

[6] R. L. Bocchino, V. S. Adve, and B. L. Chamberlain, "Software transactional memory for large scale clusters," in *PPoPP*, 2008.

[7] M. Couceiro, P. Romano *et al.*, "D2STM: Dependable distributed software transactional memory," in *PRDC*, 2009, pp. 307 –313.

[8] C. Kotselidis, M. Ansari *et al.*, "DiSTM: A software transactional memory framework for clusters," in *ICPP*, 2008, pp. 51–58.

[9] N. Carvalho, P. Romano, and L. Rodrigues, "A generic framework for replicated software transactional memories," in *NCA*, August 2011.

[10] A. Turcu and B. Ravindran, "Hyflow2: A high performance distributed transactional memory framework in Scala," Virginia Tech, Tech. Rep., 2012, http://hyflow.org/hyflow/chrome/site/pub/hyflow2-tech.pdf.

[11] K. Manassiev, M. Mihailescu, and C. Amza, "Exploiting distributed version concurrency in a transactional memory cluster," in *PPoPP*, 2006, pp. 198–208.

[12] M. K. Aguilera, A. Merchant *et al.*, "Sinfonia: A new paradigm for building scalable distributed systems," *ACM Trans. Comput. Syst.*

[13] R. Hundt, "Loop recognition in C++/Java/Go/Scala," *Scala Days*, 2011.

[14] M. M. Saad and B. Ravindran, "Transactional forwarding algorithm," Virginia Tech, Tech. Rep., January 2012.

[15] M. Herlihy and Y. Sun, "Distributed transactional memory for metric-space networks," *Distributed Computing*, 2007.

[16] A. Turcu and B. Ravindran, "Hyflow2: A high performance distributed transactional memory framework in scala," Virginia Tech, Tech. Rep., April 2012. [Online]. Available: http://hyflow.org/hyflow/chrome/site/pub/hyflow2-tech.pdf

[17] A. Turcu, B. Ravindran, and M. M. Saad, "On closed nesting in distributed transactional memory," in *TRANSACT*, 2012.

[18] A. Turcu and B. Ravindran, "On open nesting in distributed transactional memory," in *SYSTOR*, 2012, pp. 1–12.

[19] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *SPAA*, 2008, pp. 160–168.

[20] B. Z. Aditya Dhoke and B. R. Ravindran, "On closed nesting and checkpointing in replicated distributed transactional memory," Technical report, Virginia Tech, October 2012. URL http://www.ssrg.ece.vt.edu/papers/ipdps.pdf, Tech. Rep.

[21] GNU C Library, "Complete context control," http://www.gnu.org/software/libc/manual/html_node/System-V-contexts.html#System-V-contexts.

[22] A. Turcu and B. Ravindran, "On open nesting in distributed transactional memory," in *SYSTOR*. ACM, 2012, p. 12.

[23] J. E. B. Moss and A. L. Hosking, "Nested tm: Model and architecture sketches," *Sci Comp Prog*, vol. 63, no. 2, pp. 186–201, 2006.

[24] J. E. B. Moss, "Open nested transactions: Semantics and support (poster)," in *Workshop on Mem Perf Issues*, 2006.

[25] R. Koo and S. Toueg, "Checkpointing and rollback-recovery for distributed systems," *Software Engineering, IEEE Transactions on*, 1987.

[26] E. Koskinen and M. Herlihy, "Checkpoints and continuations instead of nested transactions," in *SPAA*, 2008.

[27] C. Flanagan, A. Sabry, B. Duba, and M. Felleisen, "The essence of compiling with continuations," in *ACM SIGPLAN Notices*. ACM, 1993.

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

[29] T. Willhalm and N. Popovici, "Putting Intel® threading building blocks to work," in *Workshop on Multicore software engineering*, 2008.

[30] MsgConnect, "Eldos corporation," 2012, http://www.eldos.com/msgconnect/.

[31] P. Hintjens, "ØMQ-The Guide," 2011, http://zguide.zeromq.org.

[32] A. Bieniusa and T. Fuhrmann, "Consistency in hindsight: A fully decentralized STM algorithm," in *IEEE IPDPS*, 2010, pp. 1 –12.

[33] C. Cao Minh, J. Chung *et al.*, "STAMP: Stanford transactional applications for multi-processing," in *IISWC*, September 2008.

[34] S. Mishra, "HyflowCPP: A distributed transactional memory framework for C++," Master's thesis, Virginia Tech, Blacksburg, VA, USA, 2013, http://hyflow.org/hyflow/chrome/site/pub/Mishra-Thesis.pdf.