Snake: Control Flow Distributed Software Transactional Memory

Mohamed M. Saad and Binoy Ravindran

ECE Dept., Virginia Tech
{msaad,binoy}@vt.edu

Abstract. Remote Method Invocation (RMI), Java's remote procedure call implementation, provides a mechanism for designing distributed Java technology-based applications. It allows methods to be invoked from other Java virtual machines, possibly at different hosts. RMI uses lock-based concurrency control, which suffers from distributed deadlocks, livelocks, and scalability and composability challenges. We present *Snake-DSTM*, a distributed software transactional memory (D-STM) that is based on the RMI as a mechanism for handling remote calls and transactional memory for distributed concurrency control, as an alternative to RMI/locks. Critical sections are defined as atomic transactions, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. The novelty of Snake-DSTM is in manipulating transactional memory by moving control to remote nodes, rather than remote nodes' data being copied to the node at which the transaction runs. Transaction metadata is detached from the transactional context, and the dynamic two phase commitment protocol (D2PC) is employed to coordinate the voting process among participating nodes toward making distributed transactional commit decisions. We propose a simple programming model using (Java 5) annotations to define critical sections and remote methods. Instrumentation is used to generate code at class-load time, which significantly simplifies user-space end code. No changes are needed to the underlying virtual machine or compiler. We describe Snake-DSTM's architecture and implementation, and report on experimental studies comparing it against competing models including RMI with mutual exclusion and read/write locks, distributed shared memory (DSM), and dataflow-based D-STM. Our studies show that Snake-DSTM outperforms competitors by up to 12× on different workloads using a 120-node system.

1 Introduction

Lock-based concurrency control suffers from drawbacks including deadlocks, livelocks, lock convoying, and priority inversion. In addition, it has scalability and composability challenges [10]. These difficulties are exacerbated in distributed systems with nodes, possibly multicore, interconnected using message passing links, due to additional, distributed versions of their centralized problem counterparts [12]. Transactional memory (TM) promises to alleviate these difficulties.

In addition to providing a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking [13, 11]. In TM, atomic sections are defined as *transactions* in which reads and writes to shared objects appear to take effect instantaneously. A transaction maintains its read set and write set, and at commit time, checks for conflicts on shared objects. If conflicts are detected, the transaction rolls-back its changes and retries; otherwise, the changes are made to take effect. Numerous multiprocessor TM implementations have emerged in software (STM) [27], in hardware (HTM) [11], and in a combination (Hybrid TM) [16]. Distributed STM (or D-STM) implementations also exist. Examples include Cluster-STM [5], D^2STM [7], DiSTM [14], and Cloud-TM [21]. Communication overhead, balancing network traffic, and network failures are additional concerns for D-STM.

Previous research on D-STM has largely focused on the dataflow model [30, 17], in which objects are replicated (or migrated) at multiple nodes, and transactions access local object copies. Using cache coherence protocols [12, 8, 33], consistency of the object copies is ensured. However, this model is not suitable in applications (e.g., P2P), where objects cannot be migrated or replicated due to object state dependencies, object sizes, or security restrictions. A control flow model, where objects are immobile and transactions invoke object operations via remote procedure calls (RPCs), is appropriate in such instances.

This paper focuses on the design and implementation of D-STM based on Java's Remote Method Invocation (RMI) mechanism. We are motivated by the popularity of the Java language, and the need for building distributed systems with concurrency control, using the control flow model. Support for distributed computing in Java is provided using RMI since release 1.1. However, distributed concurrency control is (implicitly) provided using locks. Besides, the RMI architecture lacks the transparency required for distributed programming, supporting a remote method requires defining an interface, skeleton and stub objects, plus changing the prototype to throw remote exceptions and extending special base class `UnicastRemoteObject`. We present *Snake-DSTM*, an RMI/D-STM implementation that uses D-STM for distributed concurrency control in (RMI's) control flow model, and exports a simpler programming model with transparent object access. Using (Java 5's) annotations, and our instrumentation engine, a programmer can define *remote* objects (or methods), and define *atomic* sections as transactions, in which reads and writes to shared (local and remote) objects appear to take effect instantaneously. Distributed atomicity, object registration, and remote method declarations are handled transparently without any changes to the underlying virtual machine or compiler. Our experimental studies show that Snake-DSTM outperforms RMI with read/write locks by as much as $12 times$ on a broad range of transactional workloads, and shows comparable performance to distributed shared memory, and dataflow D-STM. To the best of our knowledge, this is the first D-STM design and implementation in the control flow model, and constitutes the paper's contribution.

Snake-DSTM is freely available as part of the HyFlow project [22], which is producing a Java D-STM framework for the design, implementation, and

evaluation of D-STM algorithms and mechanisms, under both control flow and dataflow. We hope this will increase momentum in the TM community in D-STM research.

The rest of the paper is organized as follows. We overview past and related efforts in Section 2. In Section 3, we detail the Snake-DSTM design and implementation and underlying mechanisms. In Section 5, we experimentally evaluate Snake-DSTM against competing distributed programming models and report results. We conclude in Section 6.

2 Related Work

The high popularity of the Java language for developing large, complex systems has motivated significant research on distributed and concurrent programming models. DISK [28] is a distributed Java Virtual Machine (DJVM) for network of heterogenous workstations, and uses a distributed memory model using multiple-writer memory consistency protocol. Java/DSM [32] is a DJVM built on top of the TreadMarks [2] DSM system. JESSICA2 [34] provides transparent memory access for Java applications through a single system image (SSI), with support for thread migration for dynamic load balancing. These implementations facilitate concurrent access for shared memory. However, they rely on locks for distributed concurrency control, and thereby suffer from (distributed) deadlocks, livelocks, lock-convoying, priority inversion, non-composability, and the overhead of lock management.

TM, proposed by Herlihy and Moss [11], is an alternative approach for shared memory concurrent access, with a simpler programming model. Memory transactions are similar to database transactions: a transaction is a self-maintained entity that guarantees atomicity (all or none), isolation (local changes are hidden till commit), and consistency (linearizable execution). TM has gained significant research interest including that on STM [27], HTM [11], and HyTM [16]. STM has relatively larger overhead due to transaction management in software and architecture-independence. HTM has the lowest overhead, but assumes architecture specializations. HyTM seeks to combine the best of HTM and STM.

Similar to multiprocessor STM, D-STM was proposed as an alternative to lock-based distributed concurrency control. In [12], Herlihy *et. al.* classified distributed execution models into control-flow and dataflow models. In the control-flow model [4, 15, 29], objects are immobile and transactions invoke object operations through remote calls, resulting in a distributed locus of control flow movement — "distributed thread" [20] — for a transaction. On the other hand, in the dataflow model [30, 17], objects are replicated (or migrated) at multiple nodes, and transactions access local copies. While the dataflow model preserves the locality of reference principle, it is not applicable in many cases in which objects cannot be transferred due to state, size, or security restrictions. Example dataflow D-STM implementations include Cluster-STM [5], D^2STM [7], DiSTM [14], and Cloud-TM [21]. Communication overhead, balancing network traffic, and network failure models are additional concerns for such designs. These

implementations are mostly specific to a particular programming model (e.g., the partitioned global address space or PGAS model [1]) and often need compiler or virtual machine modifications (e.g., JVSTM [6]), or assume specific architectures (e.g., commodity clusters). While dataflow D-STM has been intensively studied, relatively little efforts have focused on applying TM concepts under the control-flow model.

Snake-DSTM is a control-flow D-STM implementation, based on the Java RMI mechanism for supporting remote procedure calls. Unlike [1, 6], it doesn't require any changes to the underlying virtual machine or compiler, as it uses embedded library as a JVM agent, which is loaded at runtime.

3 System Overview

3.1 System Model

We consider an asynchronous distributed system model, similar to Herlihy and Sun [12], consisting of a set of N nodes $N_1, N_2, ..., N_n$, communicating through weighted message-passing links. We assume that each shared object has an unique identifier. We use a grammar similar to the one in [9], but extend it for distributed systems.

A transaction is a sequence of instructions that are guaranteed to be executed atomically. Any object changes within transactional code must appear to take effect instantaneously. Each transaction has an unique identifier, and is invoked by a node (or process) in a distributed system of N nodes. A transaction can be in one of three states: *active*, *busy*, and *aborted*, or *committed*. When a transaction is aborted, it is retried by the node again using a different identifier.

Objects are resident at their originating nodes. Every object has, one "owner" node that is responsible for handling requests from other nodes for the owned object. Any node that wants to read from, or write to an object, contacts the object's owner using a remote call. A remote call may in turn make other remote calls, which construct, at the end of the transaction, a global graph of remote calls. We call this graph, a *call graph*.

3.2 Programming Model

The Java RMI specifications require defining a `Remote` interface for each remotely accessible class, and modifying class signatures to throw *remote* exceptions. Server side should register the implementation class, while client uses a delegator object that implements the desired `Remote` interface.

In our model, a programmer annotates remotely accessible methods with the `@Remote` annotation, and critical sections are defined as methods annotated with `@Atomic`. An object that contains at least one `@Remote` method is named *remote object*, and it must implement the `IDistinguishable` interface to provide our registry with a unique object identifier. Remote objects register themselves automatically at construction time, and are populated to other node registries. A

```
1 public class SearchAgent implements IDistinguishable {
2  public Object getId() {
3    return id;
4  }
5  @Remote
6  @Atomic{retries = 10}
7  public List search(String keyword) {
8    List found = new LinkedList();
9    // search at neighbors
10   for(String neighbor: neighbors){
11     SearchAgent remoteAgent = Locator.open(neighbor);
12     found.addAll( remoteAgent.search(keyword) );
13   }
14   .... // search at local database
15   return found;
16 }
17 }
```

Fig. 1. A P2P agent using an atomic remote TM method.

transactional object is one that defines one (or more) `@Atomic` methods. Atomic annotation can be, optionally, parametrized by the maximum number of transactional retries. Currently, we support the *closed nesting* model [19], which extends the isolation of an inner transaction until the top-level transaction commits. We "flatten" nested transactions into the top-level one, resulting in a complete abort on conflict, or allow partial abort of inner transactions.

Transactional or remote objects are accessed using *locators*. Traditional object references cannot be used in a distributed environment. Further, locators monitor object accesses and act as early detectors for possible transactional conflicts. Objects can be located (or opened) in read-only or read-write modes. This classification permits concurrent access for concurrent read transactions.

Figure 1 shows a distributed transactional code example. A peer-to-peer (P2P) file sharing agent atomically searches for resources and return a list of resources owners to the caller node. The agent may act recursively and propagate the call to a set of neighbor agents. At the programming level, no locks are used, the code is self-maintained by retrying on failures, and atomicity, consistency, and isolation are guaranteed (for the `search` transaction). Composability is also achieved: any other atomic method can be called within the higher-level atomic `search` operation. A conflicting transaction is transparently retried. Note that the location of the agents is hidden from the program. It is worth noting that other distributed programing models such as DSM or dataflow D-STM cannot be used in such applications, as an agent must search files at its node. This is an example of objects with system-state property.

4 Implementation

Figure 2 shows a layered architecture of our implementation. Similar to the official RMI design, we have the three layers of: 1) *Transport Layer*, where actual networking and communication handling is performed, 2) *Remote Reference Layer*, which is responsible for managing the "liveliness" of the remote objects, and 3) *Stub/Skeleton Layer*, which is responsible for managing the remote object interface between hosts. Additionally, we define an *Object Access Layer*, which provides the required transparency to the application layer. Local and remote objects are accessed in an uniform manner, and a dummy object is created to delegate calls to the RMI stub. Transactional code is maintained by a *Transaction Manager* module, which provides distributed atomicity and memory consistency for applications. As described in Section 4.1, an *Instrumentation Engine* is responsible for load-time code modifications, which is required for the Transaction Manager and Object Access Layer.

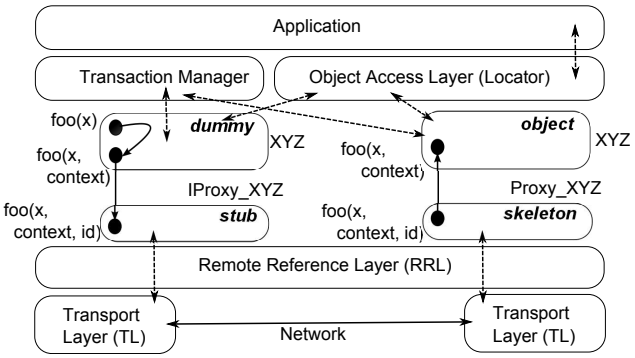


Fig. 2. Snake-DSTM layered architecture overview.

4.1 Instrumentation Engine

Java *Instrumentation* provides a run-time ability to modify and generate bytecode at class load-time. We exploited this feature to modify class code at runtime, add new fields, modify annotated methods to support remote and transactional behavior, and generate helper classes. We consider a Java method as the basic annotated block. This approach has two advantages. First, it retains the familiar programming model, where `@Atomic` replaces `synchronized` methods and `@Remote` substitutes for RMI calls. Secondly, it simplifies transactional memory maintenance, which has a direct impact on performance. Transactions need not handle local method variables as part of their read or write sets.

Our Instrumentation Engine works in two phases; the first phase processes *remote objects*. For any class with one (or more) methods annotated as `@Remote`,

a `Remote` interface is generated with the remote method's signature. Further, a delegator class that implements the `Remote` interface is generated to work as the RMI-client stub. The original class constructors are modified to register objects at the object registry and populate object IDs to other nodes. That has two purposes: i) objects are accessed with a reference of the same type, so objects and object proxies are treated equally and transparently; and ii) no changes to remote method signatures are required, as the modified signature versions are defined by delegator generated code. This phase simplifies the way remote objects are accessed, and reduces the burden of writing complex code.

The second phase handles transactional code generation. This transformation occurs as follows:

- ***Classes.*** A synthetic field is added to represent the state of the object as local or remote. The class constructor(s) code is modified to register the object with the Directory Manager at creation time.
- ***Fields.*** For each instance field, setter and getter methods are generated to delegate any direct access for these fields to the transaction context. Class code is modified accordingly to use these methods.
- ***Methods.*** Two versions of each method are generated. The first version is identical to the original method, while the second one represents the transactional version of the method. During the execution of transactional code, the second version of the method is used, while the first version is used elsewhere.
- ***@Atomic methods.*** Atomic methods are duplicated as described before, however, the first version is not similar to the original implementation. Instead, it encapsulates the code required for maintaining transactional behavior, and it delegates execution to the transactional version of the method.

Figure 3 shows part of the instrumented version of a `SearchAgent` class defined in Figure 1.

4.2 Distributed Software Transactional Memory

Supporting shared memory-like access in distributed systems requires an additional level of indirection. Each transaction must preserve memory consistency, and must expose its local changes instantaneously. In order to do that, old or new values of modified objects must be stored at local-transaction buffers till commit time. Two strategies can be used to achieve this: i) *undo-log* [16], where changes are made to the main object, while old values are stored in a separate log; and ii) *write-buffer* [10], where changes are made to transaction-local memory and written to the main object at commit time. Both strategies are applicable in the distributed context. However, (distributed) transactions cannot move between nodes during their execution with all these metadata (undo-logs or write-buffers) due to high communication costs. Instead, transaction metadata must be detached from the transaction context, while keeping the minimal information mobile with the transaction. In Snake-DSTM, we implemented both approaches. Using a distributed mechanism for storing transaction read-set and write-set, distributed transactions are managed with minimum amount of mobile

```
1 // Generated Remote interface
2 interface $HY$_ISearchAgent
3     extends Remote, Serializable{
4   public List search(Object id, ControlContext context,
        String keyword) throws RemoteException;
5   ....
6 }
7 // Generated Proxy delegator stub
8 class $HY$_Proxy_SearchAgent
9     extends UnicastRemoteObject
10    implements $HY$_ISearchAgent{
11  ....
12 }
13 public class SearchAgent implements IDistinguishable {
14   // Remote Proxy referece
15   $HY$_ISearchAgent $HY$_proxy;
16   // Modified constructor
17   SearchAgent(String id){
18     ....
19     DirectoryManager.register(id, this);
20   }
21   // Synthetic duplicate method
22   public List search(String keyword, Context c) {
23     if($HY$_proxy!=null)  //Invoke remote call
24       return $HY$_proxy.search(id, c, keyword);
25     ....  //execute call locally
26   }
27   // Original method instrumented
28   public List search(String keyword) {
29     //Transaction active thread
30     Context context = ContextDelegator.getInstance();
31     boolean commit = true;
32     List result = null;
33     for (int i=10; i>0; --i) {
34       //Initialize transaction
35       context.init();
36       try{
37         result=search(keyword, context);  //Try execute
38       } catch(TransactionException ex) {
39         commit = false;  //Aborted
40       } catch(Throwable ex) {
41         throwable = ex; //Application Exception
42       }
43       if(commit){
44         if (context.commit()) {
45           if (throwable == null)
46             return result; //Committed
47           throw (IOException)throwable; //Rethrow Exception
48         }
49      }else{
50         context.rollback(); //Rollback
51         commit = true;
52      }
53     }
54     throw new TransactionException();  //Maximum Retries
55   }
56 }
```

Fig. 3. Instrumented version of SearchAgent class.

data (e.g. transaction id, priority). The complete algorithm and more implementation details are available in a technical report [24].

Undo Log (Eager-Pess)

On Write
If(owned) resolve
set owned by me
Backup and Change in master copy
On Read
If(owned) resolve
Read value and version
Try Commit
Validate reads (version < current)
On Commit
Increment owned versions
Release owned
On Rollback
Undo changes for owned
Release owned

Write Buffer (Lazy-Opt)

On Write
Change in private copy
On Read
If(in Write Set) read local value
else read master copy value
Read version
Try Commit
Acquire ownership of write-set
Validate reads (version ¡ current)
On Commit
Write values to main copy
Increment owned versions
Release owned
On Rollback
Discard local changes

Fig. 4. Snake D-STM implementations.

Before (and after) accessing any transactional object field, the transaction is consulted for read (or written) value. A transaction builds up its write and read sets, and handles any private buffers accordingly. At commit time, a distributed validation step is required to guarantee consistent memory view. In this phase, transaction originator nodes trigger a voting request to the participating nodes. Each node uses its portion of write and read sets to make its local decision. If validation succeeds on all nodes, the transaction is committed; otherwise, an abort handler rolls-back the changes. During the validation phase, the transaction state is set to *busy*, which ensures that a transaction cannot be aborted. This helps in ensuring the correctness of the validation (i.e., all nodes unanimously agree on the transaction to be committed and the transactions to be aborted), and also, it prevents transactions at later stages from being aborted by newly started ones.

Figure 4 shows our two implementations of the Snake-DSTM: write-buffer and undo log. Objects use verrsioned lock to enable ownership and validation. `Try Commit` procedure is used during the voting to make sure that all nodes are ready to commit.

4.3 Distributed Contention Management

Two transactions conflict if they concurrently access the same object, and one of them is a write transaction. Upon detecting a conflict, a *contention management policy* (CM) is used to resolve this situation (arbitrarily or priority-based) e.g., one of the transactions is stalled or aborted and retried. A wide range of

transaction contention management policies has been studied for non-distributed STM [26, 25]. We classify CMs into three categories: 1. *Incremental CM* (e.g., Karma, Eruption, Polka), where the CM builds up the priorities of the transactions during transaction execution; 2. *Progressive CM* (e.g., Kindergarten, Priority, Timestamp, Polite), which ensures a system-wide progress guarantee (i.e., at least one transaction will proceed to commit); and 3. *Non-Progressive CM* (e.g., Backoff, Aggressive), which assumes that conflicting transactions will eventually complete, however, livelock situations can occur.

As mentioned earlier, in the control flow model, a distributed transaction T_x is executed over multiple nodes. Under Incremental CM, T_x can have different priorities at each node. This is because, a transaction builds its priority during its execution over multiple nodes. Under this behavior, a live-lock situation can occur. Consider transactions T_x and T_y with priorities P_x, P'_y and P'_x, and P_y at nodes N_1 and N_2, respectively. It is clear that, if $P'_x > P_y$ and $P'_y > P_x$, then both transactions will abort each other, and this will continue forever. The lack of a central store for transactional priorities causes this problem. However, having such a central store will significantly increase the communication overhead during transaction execution, causing a system bottleneck. Non-Progressive CM shows comparable performance for non distributed STM [3]. Nevertheless, our experiments show that it cannot be extended for D-STM due to the expensive cost of retries (see Appendix).

4.4 Global Commitment Protocol

In the control flow model, a remote call on an object may trigger another remote call to a different object. The propagated access of objects forms a *call graph*, which is composed of nodes (i.e., sub-transactions) and undirected edges (i.e., calls). This graph is essential for making a commit decision. Each participating node may have a different decision (on which transaction to abort/commit) based on conflicts with other concurrent transactions. Thus, a voting protocol is required to collect votes from nodes, and the originating transaction can commit only if it receives an "yes" message from all nodes. By default, we implement the D2PC protocol [18], however, any other protocol may substitute it. We choose D2PC, as it yields the minimum possible time for collecting votes [18], which reduces the possibility of conflicts and results in the early release of acquired objects. Furthermore, it balances the overhead of collecting votes by having a variable coordinator for each vote.

5 Experimental Evaluation

Distributed Benchmarks. We developed a set of distributed benchmarks to evaluate Snake-DSTM against competing models including: i) classical RMI, which uses mutual exclusion locks and read/write locks with random timeout mechanism to handle deadlocks and livelocks; ii) distributed shared memory (DSM), which uses the Home directory protocol such as Jackal [31]; and iii)

distributed dataflow STM implementation [23]. Our benchmark suite includes a distributed version of the vacation benchmark from the STAMP benchmark suite [35] (vacation) and two monetary applications (bank and loan).

Testbed. We conducted our experiments on a multiprocessor/multicomputer network comprising of 120 nodes, each of which is an Intel Xeon 1.9GHz processor, running Ubuntu Linux, and interconnected by a network with $1ms$ end-to-end delay. Each node invokes 50-200 sequential transactions. In a single experiment, we thus executed 6-24 thousands transactions, and measured the throughput for each concurrency model, for each benchmark. Our experiments shows that Snake-DSTM write-buffer implementation outperforms undo-log implementations under all benchmarks. The reason for this is that undo-log pessimistic approach incur relatively larger number of retries, which in turn increases objects requests over the network. In this section we focus on Snake-DSTM write-buffer results against other concurrency models.

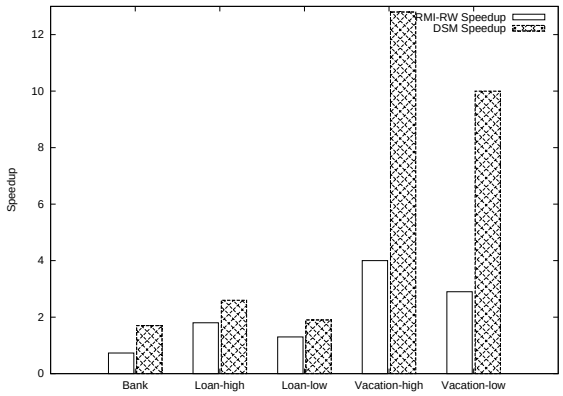


Fig. 5. Snake-DSTM speedup for a distributed benchmark suit over 120-node system.

Evaluation. Figure 5 [1] shows the relative throughput speedup achieved by Snake-DSTM over other concurrency models on the benchmarks. We observe that Snake-DSTM outperforms all other models under loan and vacation (the speedup ratio ranges between $1.3\times$ and $12.8\times$). Under Bank benchmark only two nodes are involved into the transfer transaction, so Snake-DSTM overhead (voting, validation and versioning) outweighs the performance gain for this simple transaction, relative to RMI.

Using the Loan benchmark, transaction execution time was $200ms$ under ideal conditions. Six different objects were accessed per each transaction, issuing twenty remote calls. Figure 6(a) shows the scalability of Snake-DSTM under

[1] High and low indicate the benchmark contention which is controlled by either increasing write transactions or reducing the number of objects

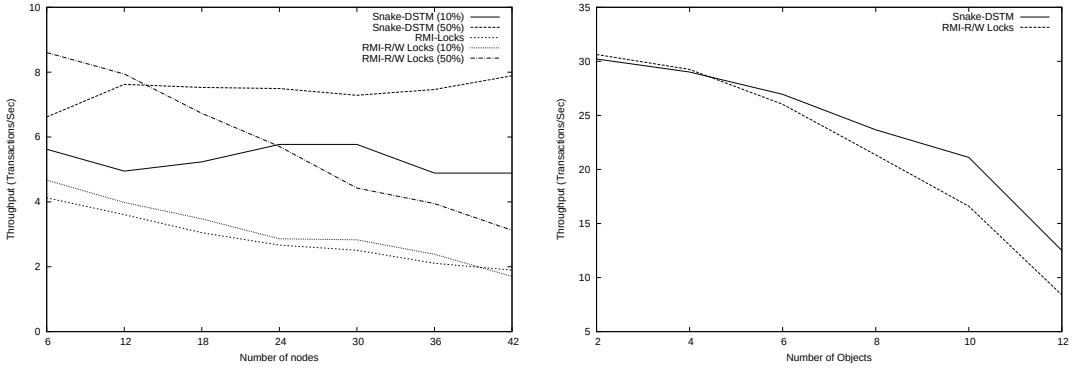


Fig. 6. Throughput of Loan benchmark: a) under increasing number of nodes, b) using pure read transactions over 12 nodes, and variable object count per transaction.

increasing number of nodes, and using 50% and 10% read-only transactions. Figure 6(b) shows the throughput under increasing number of participating objects in each transaction (transaction execution time under no contention is $350ms$ in this experiment). Greater the number of accessed objects, higher the algorithm overhead, and higher the number of remote calls per each transaction (e.g., a transaction of twelve objects issues 376 remote object calls during its execution).

From Figure 6(a), we observe that Snake-DSTM outperforms classical RMI using mutual exclusion locks (RMI-Locks), and also using read/write locks (RMI-R/W), by 180% at high contention (10% reads), and by 150% at normal contention (50% reads). Though RMI with read/write locks shows better performance at a single point (6 nodes) due to the voting protocol overhead, yet, it suffers from performance degradation at increasing loads. It worth noting that the y-axis represents the nodal throughput, which means that in Figure 6(a) Snake-DSTM sustains almost the same nodal throughput with increasing the objects contention.

Figure 6(b) uses the no-contention situation (100% reads) to compare the overhead of Snake-DSTM and RMI-R/W. At small number of shared objects per transaction, the TM overhead outweighs the provided concurrency, and both Snake-DSTM and RMI-R/W incur the same overhead. With increasing number of objects, Snake-DSTM outperforms RMI-R/W by 50%. Notice that the throughput degradation is not due to contention (100% reads), but it is because the transaction execution time is different (more objects at each data point), so the relevance of this figure is the relative implementation overhead of RMI and Snake-DSTM appraoches.

Figure 7(a) compares control-flow and dataflow D-STM implementations using the Bank Benchmark, where the end-to-end latency is changed, due to network conditions or object size. Figure 7(b) shows the effect of increasing the number of calls per a single remote object on Snake-DSTM throughput. This experiment illustrates the trade-off between employing locality of reference under

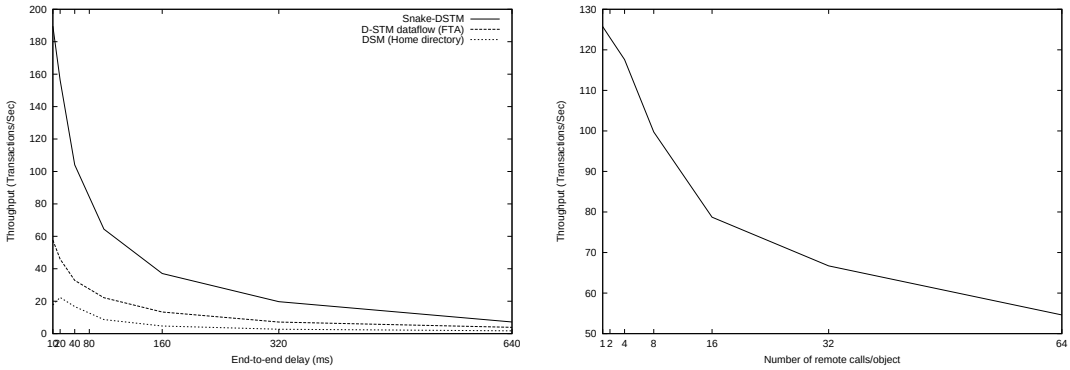


Fig. 7. Throughput of Bank benchmark: a) under dataflow and control-flow models using different end-to-end delay (ρ=1 and #calls/object=1), b) Snake-DSTM throughput under increasing number of calls per object.

dataflow model, and invoking remote calls at immobile objects using control-flow model. The best strategy is application based, which leaves a space for having both models in use.

End-to-end delay plays an important role in the design of distributed systems. We can decompose it into: network delay (propagation, processing, transmission, and queuing delay) and JVM delay (serialization, marshaling, and type checking). We define the object-to-parameter ratio (ρ) as the ratio of the end-to-end delay incurred in sending an object to the end-to-end delay incurred in sending the remote call parameters for this object. For example, ρ=2, when sending an object requires double the end-to-end delay of sending parameters of a remote call.

Figure 7(a) shows the effect of end-to-end delay on throughput when ρ=1 (i.e., sending the object is equivalent to sending the remote call parameters). Here, only one call is issued per any remote object within a transaction, which means that, for an application with ρ=4, the throughput of the dataflow flow model should be compared to the control-flow throughput multiplied by four. Similarly, for an application that invokes four calls per each object within a transaction, the equivalent control-flow throughput is divided by four.

Figure 7(b) demonstrates the effect of not employing locality of reference: in the control flow model, each remote call incurs a round-trip network delay. As shown in the figure, it reduces throughput by 25% for four to eight calls. This should be considered in environments with high link latency.

6 Conclusions

We presented Snake-DSTM, a high performance, scalable, distributed STM based on the control flow execution model. Our experiments show that Snake-DSTM outperforms other distributed concurrency control models, with accept-

able number of messages and low network traffic. Control flow is beneficial under non-frequent object calls or when objects must be immobile due to object state dependencies, object sizes, or security restrictions. Our implementation shows that Snake-DSTM provides comparable performance to classical distributed concurrency control models, and exports a simpler programming interface, while avoiding dataraces, deadlocks, and livelocks.

The HyFlow project provides a testbed for the TM research community to design, implement, and evaluate algorithms for D-STM. HyFlow is publicly available at *hyflow.org*.

References

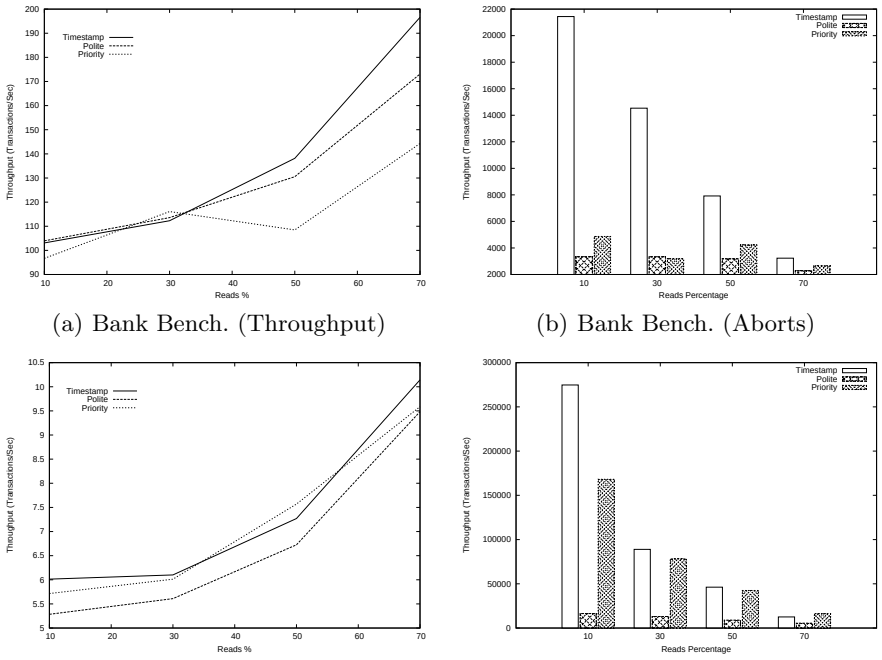
1. Partitioned Global Address Space (PGAS), 2003.
2. C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, (29), 1996.
3. M. Ansari, C. Kotselidis, M. Lujn, C. Kirkham, and I. Watson. Investigating contention management for complex transactional memory benchmarks. In *In MULTIPROG09*.
4. K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.
5. R. L. Bocchino, V. S. Adve, and B. L. Chamberlain. Software transactional memory for large scale clusters. In *PPoPP '08*, pages 247–258, NY, USA, 2008. ACM.
6. J. a. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63:172–185, December 2006.
7. M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC '09*, nov 2009.
8. M. J. Demmer and M. Herlihy. The Arrow distributed directory protocol. In *DISC '98*, pages 119–133, London, UK, 1998. Springer-Verlag.
9. R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44:404–415, January 2009.
10. M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. pages 253–262, NY, USA, Oct 2006. ACM.
11. M. Herlihy, J. E. B. Moss, J. Eliot, and B. Moss. Transactional memory: Architectural support for lock-free data structures. In *in Proceedings of the 20th Annual International Symposium on Computer Architecture*, pages 289–300, 1993.
12. M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *In DISC'05*, pages 324–338. Springer, 2005.
13. T. Johnson. Characterizing the performance of algorithms for lock-free objects. *Computers, IEEE Transactions on*, 44(10):1194 –1207, Oct. 1995.
14. C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.
15. B. Liskov, M. Day, M. Herlihy, P. Johnson, and G. Leavens. Argus reference manual. Technical report, Cambridge University, Cambridge, MA, USA, 1987.
16. K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood. LogTM: Log-based transactional memory. In *In Proc. 12th Annual International Symposium on High Performance Computer Architecture*, 2006.

17. M. Philippsen and M. Zenger. Java Party transparent remote objects in Java. concurrency practice and experience, 1997.
18. Y. Raz. The Dynamic Two Phase Commitment (D2PC) Protocol. In *ICDT'95*, pages 162–176, London, UK, 1995. Springer-Verlag.
19. D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978.
20. F. Reynolds. An architectural overview of alpha: A real-time distributed kernel. In *Proceedings of the Workshop on Micro-kernels and Other Kernel Architectures*, pages 127–146, Berkeley, CA, USA, 1992. USENIX Association.
21. P. Romano, L. Rodrigues, N. Carvalho, and J. Cachopo. Cloud-TM: harnessing the cloud with distributed transactional memories. *SIGOPS Oper. Syst. Rev.*, 44:1–6, April 2010.
22. M. M. Saad and B. Ravindran. Distributed Hybrid-Flow STM : Technical Report. Technical report, ECE Dept., Virginia Tech, December 2010.
23. M. M. Saad and B. Ravindran. Transactional Forwarding Algorithm : Technical Report. Technical report, ECE Dept., Virginia Tech, January 2011.
24. M. M. Saad and B. Ravindran. RMI-DSTM: Control Flow Distributed Software Transactional Memory: Technical Report. Technical report, ECE Dept., Virginia Tech, February 2011.
25. W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05*, pages 240–248, New York, NY, USA, 2005. ACM.
26. W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC '04*, NL, Canada, 2004. ACM.
27. N. Shavit and D. Touitou. Software transactional memory. In *PODC'95*, pages 204–213, New York, NY, USA, 1995. ACM.
28. M. Surdeanu and D. Moldovan. Design and performance analysis of a distributed java virtual machine. *IEEE Trans. Parallel Distrib. Syst.*, 13:611–627, June 2002.
29. M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of legacy Java software. In *ECOOP'01*.
30. E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *In ECOOP'02*.
31. R. A. F. B. R. Veldema and H. E. Bal. Distributed shared memory management for java. In *In ASCII2000*, page 256, 2000.
32. W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.
33. B. Zhang and B. Ravindran. Brief announcement: Relay: A cache-coherence protocol for distributed transactional memory. In *OPODIS '09*, pages 48–53, Berlin, Heidelberg, 2009. Springer-Verlag.
34. W. Zhu, C.-L. Wang, and F. Lau. Jessica2: a distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, 2002.
35. C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "STAMP: Stanford transactional applications for multi-processing," in *IISWC '08*.

Appendix: Distributed Contention Policy

As illustrated in Section 4.3, progressive contention management policies are the most suitable CM for distributed environments.

In Figure 8, the effect of progressive contention management policies is shown. Six shared objects for the Loan benchmark (and two for the Bank benchmark) were accessed using twelve nodes issuing concurrent transactions. To increase contention, we forced every transaction to access *all* shared objects during its execution.



(a) Bank Bench. (Throughput) (b) Bank Bench. (Aborts)

(c) Loan Bench. (Throughput) (d) Loan Bench. (Aborts)

Fig. 8. Throughput and number of aborts of Loan and Bank benchmarks under different progressive contention management policies.

From Figure 8, we observe the effect of CMs on throughput under the Loan and Bank benchmarks. The Timestamp CM performs better than the Polite and Priority CMs, but it results in the highest abort rate, and thus incurs more processing overhead. The Polite back-off mechanism with retries manages to significantly reduce aborts (7-14 times less), while yielding moderate throughput. The Static Priority CM gives the worst performance. Besides, it suffers from starvation situations for low priority transactions.