RMI-DSTM: Control Flow Distributed Software Transactional Memory

[Technical Report]

Mohamed M. Saad
ECE Dept., Virginia Tech
msaad@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
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*, a distributed software transactional memory (D-STM) that is based on the RMI control-flow model for distributed programming and D-STM for distributed concurrency control, as an alternative to RMI/locks. 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. Snake-DSTM is based on RMI as a mechanism for handling remote calls, and defines critical sections as atomic transactions, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. Snake-DSTM uses the dynamic two phase commitment protocol (D2PC) to coordinate the voting process among participating nodes toward making distributed transactional commit decisions. 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 as much as 150-180% on a broad range of transactional workloads on a 72-node system.

Categories and Subject Descriptors D.1.3 [*Software*]: Concurrent Programming; H.2.4 [*Systems*]: Transaction processing

General Terms Design, Performance, Parallel Programming

Keywords Distributed Software Transactional Memory (D-STM), Control Flow, Remote Method Invocation (RMI)

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 [21]. 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 [24]. 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 [23, 25]. 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) [19, 20, 22, 48], in hardware (HTM) [18, 23], and in a combination (Hybrid TM) [13, 35]. Distributed STM (or D-STM) implementations also exist. Examples include Cluster-STM [9], D^2STM [12], DiSTM [26], and Cloud-TM [42]. 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 [36, 51], in which objects are replicated (or migrated) at multiple nodes, and transactions access local object copies. Using cache coherence protocols [14, 24, 54], 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. 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 150-180% 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 [43], which is producing a Java D-STM framework for the design, imple-

mentation, 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 [49] 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 [53] is a DJVM built on top of the TreadMarks [2] DSM system. JESSICA2 [55] 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 [23], 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 [19–22, 32, 33, 48], HTM [3, 8, 18, 23, 34], and HyTM [7, 13, 27, 35]. 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 [24], Herlihy *et. al.* classified distributed execution models into control-flow and dataflow models. In the control-flow model [5, 29, 50], objects are immobile and transactions invoke object operations through remote calls, resulting in a distributed locus of control flow movement — "distributed thread" [39] — for a transaction. On the other hand, in the dataflow model [36, 51], 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 [9], D^2STM [12], DiSTM [26], and Cloud-TM [42]. 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 [10]), 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.

D-STM can also be classified based on system architecture: cache-coherent D-STM (cc D-STM) [24], where a small number of nodes (e.g., 10) are interconnected using message-passing links [14, 24, 54], and a cluster model (cluster D-STM), where a group of linked computers works closely together to form a single computer [9, 12, 26, 30, 41]. The most important difference between the two is communication cost. cc D-STM assumes a metric-space network between nodes, while cluster D-STM differentiates between access to local cluster memory and remote memory at other clusters.

Snake-DSTM is a control-flow D-STM implementation, based on the Java RMI mechanism for supporting remote procedure calls. Unlike [1, 10], 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.

```
1 public class SearchAgent implements
      IDistinguishable {
2   ....
3   public Object getId() {
4     return id;
5   }
6
7   @Remote  @Atomic{retries=10}
8   public Object transfer(URI name) {
9     ....
10  }
11
12  @Remote
13  public URI find(String keyword) {
14    ....
15  }
16
17  @Atomic
18  public static Object search(String keyword) {
19    for(String tracker: trackers){
20      SearchAgent agent = Locator.open(tracker);
21      URI url = agent.find(keyword);
22      if(url!=null)
23        return agent.transfer(url);
24      return null;
25    }
26  }
27 }
```

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

3. System Overview

3.1 System Model

We consider an asynchronous distributed system model, similar to Herlihy and Sun [24], 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 [17], 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 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 [38], 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 transfers them 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, and atomicity, consistency, and isolation are guaranteed (for the `transfer` transaction). Composability is also achieved: the atomic `find` and `transfer` methods have been composed into 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.

4.1 Instrumentation Engine

Java *Instrumentation* provides a run-time ability to modify and generate byte-code 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

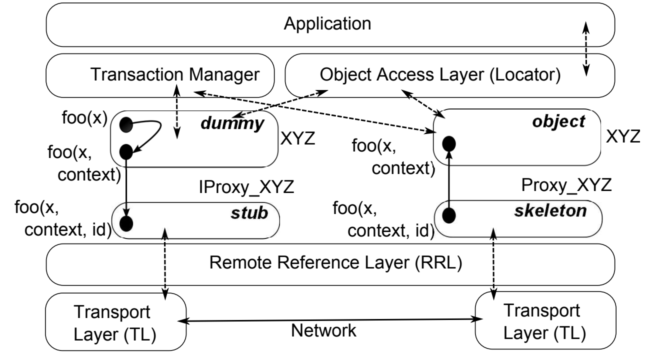


Figure 2. Snake-DSTM layered architecture overview.

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 com-

```
1 // Generated Remote interface
2 interface $HY$_ISearchAgent
3     extends Remote, Serializable{
4   public Object transfer(Object id,
        ControlContext context, URI name) throws
        RemoteException;
5   ....
6 }
7
8 // Generated Proxy delegator stub
9 class $HY$_Proxy_SearchAgent
10    extends UnicastRemoteObject
11    implements $HY$_ISearchAgent{
12  ....
13 }
14
15 public class SearchAgent implements
      IDistinguishable {
16   // Remote Proxy referece
17   $HY$_ISearchAgent $HY$_proxy;
18   ....
19   // Modified constructor
20   SearchAgent(String id){
21     ....
22     DirectoryManager.register(id, this);
23   }
24   ....
25   // Synthetic duplicate method
26   public Object transfer(URI name, Context c) {
27     if(remote_obj$){
28       //Invoke remote call
29       return Proxy.open(id).deposit(dollars);
30     }
31     ....
32   }
33   // Original method instrumented
34   public Object transfer(URI name) {
35     //Transaction active thread
36     Context context = ContextDelegator.
          getInstance();
37     boolean commit = true;
38     Object result = null;
39     for (int i=10; i>0; --i) {
40       //Initialize transaction
41       context.init();
42       try{
43         //Try execute
44         result=transfer(name, context);
45       } catch(TransactionException ex) {
46         commit = false;   //Aborted
47       } catch(Throwable ex) {
48         //Application Exception
49         throwable = ex;
50       }
51       if(commit){
52         if (context.commit()) {
53           if (throwable == null)
54             return result; //Committed
55           //Rethrow Application exception
56           throw (IOException)throwable;
57         }
58       }else{
59         context.rollback(); //Rollback
60         commit = true;
61       }
62     }
63     //Maximum retries reached
64     throw new TransactionException();
65   }
66 }
```

Figure 3. Instrumented version of SearchAgent class.

mit time. Two strategies can be used to achieve this: i) *undo-log* [35, 40], where changes are made to the main object, while old values are stored in a separate log; and ii) *write-buffer* [21, 31, 32], 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 data (e.g. transaction id, priority). The complete algorithm and more implementation details are available in a technical report [44].

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.

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* (CP) 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 [46, 47]. We classify CPs into three categories: 1. *Incremental CP* (e.g., Karma, Eruption, Polka), where the CP builds up the priorities of the transactions during transaction execution; 2. *Progressive CP* (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 CP* (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 CP, 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 CP shows comparable performance for non distributed STM [4]. Nevertheless, our experiments show that it cannot be extended for D-STM due to the expensive cost of retries (see Section 5).

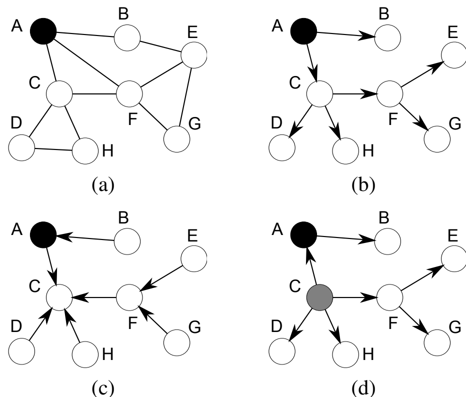


Figure 4. (a) Call graph: each node represents a sub-transaction, and edges denote remote calls between them. (b) Originating node A sends a PREPARE message that is forwarded to other nodes. (c) The vote is replied, and the coordinator is selected through the process of last-neighbor forwarding. (d) Coordinator node C publishes the vote result

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, however, any other protocol may substitute it. We choose D2PC, as it yields the minimum possible time for collecting votes [37], 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. For completeness, here, we overview the key idea of D2PC in the context of transactional memory (complete details can be found in [37]).

In D2PC, an originating node, which starts a transaction, sends a *PREPARE* message to its neighbors, containing the transaction identifier. Each node forwards this message to its neighbors in the call graph, except its parent. If a node receives the *PREPARE* message again, it discards it. The number of messages sent is the number of edges in the call graph. Each node consults its *Contention Manager* for committing the requested transaction. The message propagation results in the construction of a spanning tree of the *call graph*: each node remembers its parent and the children nodes to which it propagates its message to. D2PC doesn't distinguish between parent and children nodes; it treats them equally as "neighbors." It is worth noting that this simple construction for a spanning tree is better than trying to obtain the minimum spanning tree (MST). The problem of finding the MST has well known algorithms such as Prim and Kruskal's algorithm [11], and also parallel and distributed algorithms [6, 15, 16, 28]. Although the best MST algorithm has $O(\sqrt{|N|log^*|N|} + D)$, where D is the network diameter, it involves sending many number of messages to construct the MST. In our case, the voting protocol broadcasts a maximum of three messages, so it is impractical to construct the MST. In addition, the spanning tree changes according to the call graph, which is constructed for each transaction, so we cannot rely on pre-generated MST.

Now, assume that one or more nodes decide to send an *ABORT* message. The nodes will forward the message to their neighbors, which in turn, will recursively forward to theirs, except the sender. Sub-transactions will be terminated and the originating transaction will be retried. However, in the success case (i.e., no conflicts were to occur), all nodes will send a *COMMIT* message. Upon receiving a "COMMIT" message from all its neighbors, including its parent, except the last, a node forwards the commit decision to the last neighbor. It can be shown that there will be just one node that will receive all the votes, and this node is selected dynamically based on message speed and nodal delays. This node will be elected as a *coordinator* for the current vote. Similar to the failure case, the coordinator populates the *COMMIT* decision to others, commits the distributed changes, and the transaction completes.

Figure 4 shows a possible execution of D2PC for collecting votes for a transaction distributed over seven nodes.

5. Experimental Evaluation

5.1 Distributed Benchmarks.

We developed a set of distributed benchmarks to evaluate Snake-DSTM against competing models including: 1. classical RMI, which uses mutual exclusion locks and read/write locks with random timeout mechanism to handle deadlocks and livelocks; 2. distributed shared memory (DSM), which uses the Home directory protocol such as Jackal [52]; and 3. distributed dataflow STM implementation [45].

Our benchmarks include the following.

Loan Benchmark. This includes a set of bank accounts with money distributed over different nodes. A loan request is issued from one of the accounts to one (or more) remote accounts. A remote account can forward the loan request and ask others for sub-loans. The request propagation forms a directed graph of sub-requests between the accounts. An entire loan request takes effect atomically, by virtue of the request being implemented as an atomic transaction. The nesting level of the requests and the number of participating accounts are configurable. The number of inter-node remote object calls grow exponentially with the number of participating objects (i.e., the nesting level).

Bank Benchmark. This is a distributed banking application, which maintains a set of accounts distributed over bank branches. The application contains two atomic transactions: a) *transfer transaction*, which transfers a given amount between two accounts, and b) *total balance transaction*, which computes the total balance for given accounts. The object size and the number of operations per object during a transaction is configurable.

File Sharing Benchmark. This is a P2P file sharing agent (see Figure 1), which searches shared file contents by keywords, atomically transfers data from other remote agents, and updates some application-specific data. DSM and dataflow D-STM cannot be used with this benchmark, as a remote agent must search files at its node. This is an example of immobile objects with system-state property.

5.2 Evaluation.

Testbed. We conducted our experiments on a multiprocessor/multicomputer network comprising of 72 nodes, each of which is an Intel Xeon 1.9GHz processor, running Ubuntu Linux, and interconnected by a network with $1ms$ link delay. We ran the experiments using one processor per node, because we found that this configuration was necessary to obtain stable runtimes. This configuration also generated faster runtimes than using multiple processors per node, because it eliminated resource contention between two processors on one node.

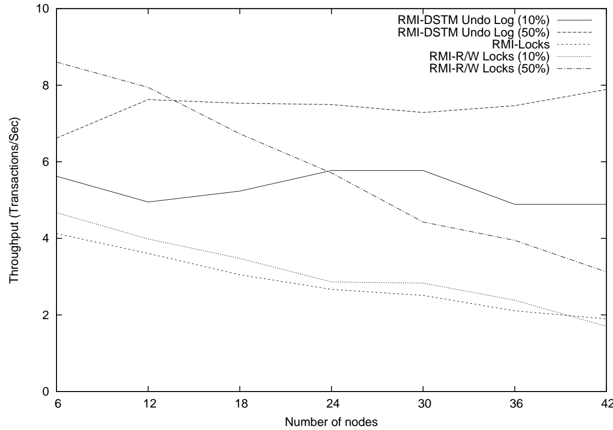


Figure 5. Throughput of Loan benchmark under increasing number of nodes.

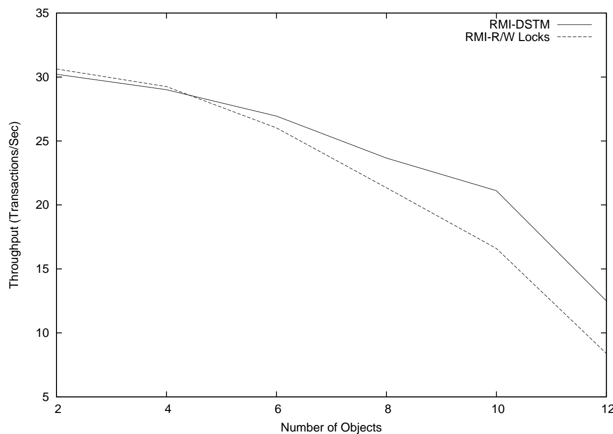
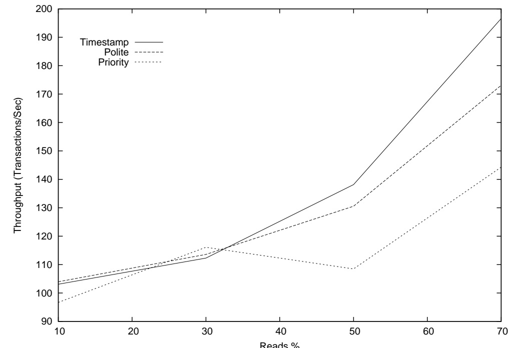


Figure 6. Throughput of Loan benchmark using pure read transactions over 12 nodes, and variable object count per transaction.

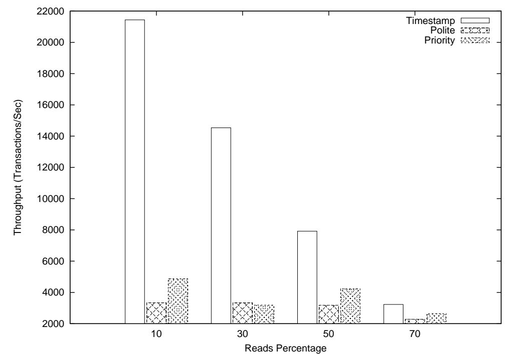
Using the Loan benchmark, twelve accounts were distributed equally on nodes, and hundred transactions were executed at each node. Transaction execution time was $200ms$ under ideal conditions. Six different objects were accessed per each transaction, issuing twenty remote calls. Figure 5 shows the scalability of Snake-DSTM under increasing number of nodes, and using 50% and 10% read-only transactions.

Figure 6 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). In Figure 7, 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.

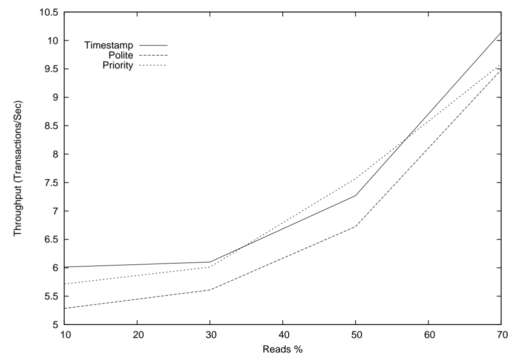
Figure 8 compares control-flow and dataflow D-STM implementations using the Bank Benchmark, where the end-to-end latency is changed, due to network conditions or object size. Figure 9 shows the effect of increasing the number of calls per a single remote object on Snake-DSTM throughput.



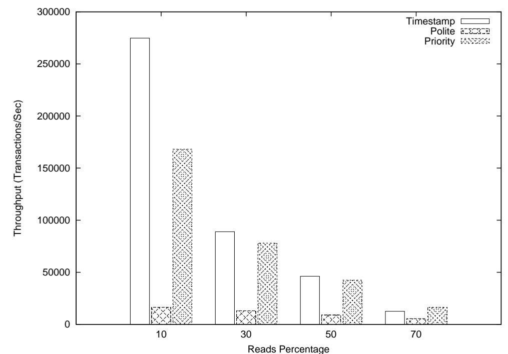
(a) Bank Bench. (Throughput)



(b) Bank Bench. (Aborts)



(c) Loan Bench. (Throughput)



(d) Loan Bench. (Aborts)

Figure 7. Throughput and number of aborts of Loan and Bank benchmarks under different progressive contention management policies.

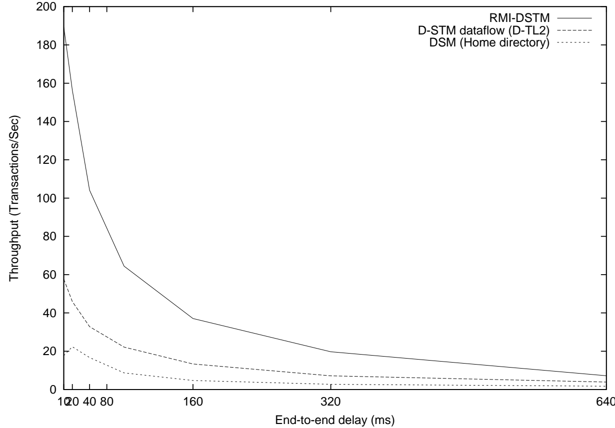


Figure 8. Throughput of Bank benchmark under dataflow and control-flow models using different end-to-end delay (ρ=1 and #calls/object=1).

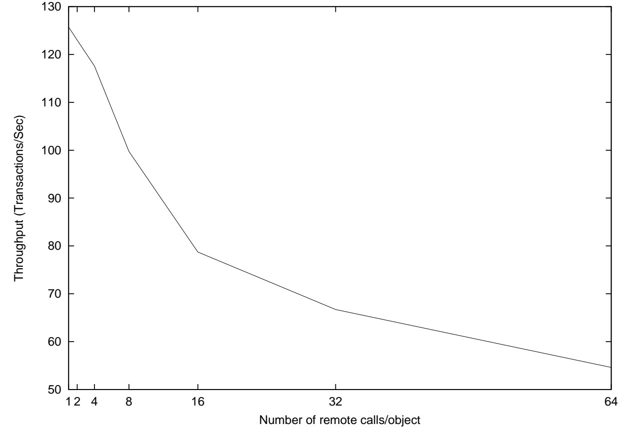


Figure 9. Bank benchmark: Snake-DSTM throughput under increasing number of calls per object.

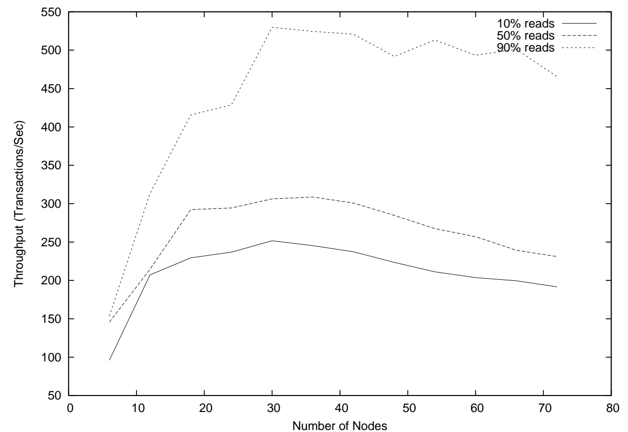


Figure 10. Throughput of File Sharing (P2P Agent) benchmark.

In Figure 10, we show the scalability of Snake-DSTM using File Sharing Benchmark (P2P search agent with two trackers) running over 72 nodes.

From Figure 5, 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.

Figure 6 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%.

As illustrated in Section 4.3, progressive contention management policies are the most suitable CP for distributed environments. From Figure 7, we observe the effect of CPs on throughput under the Loan and Bank benchmarks. The Timestamp CP performs better than the Polite and Priority CPs, 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 CP gives the worst performance. Besides, it suffers from starvation situations for low priority transactions.

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 8 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 9 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.

Figure 10 illustrates the scalability of our model for P2P File Sharing benchmark running over the 72-node system. 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 (immobile object). Besides, code simplicity and maintainability are additional features provided by our approach.

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 acceptable 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] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie. Unbounded transactional memory. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 316–327, Washington, DC, USA, 2005. IEEE Computer Society.

[4] M. Ansari, C. Kotselidis, M. Lujn, C. Kirkham, and I. Watson. Investigating contention management for complex transactional memory benchmarks. In *In Proc. Second Workshop on Programmability Issues for Multi-core Computers (MULTIPROG09*, 2009.

[5] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[6] B. Awerbuch. Optimal distributed algorithms for minimum weight spanning tree, counting, leader election, and related problems. In *STOC '87: Proceedings of the nineteenth annual ACM symposium on Theory of computing*, pages 230–240, New York, NY, USA, 1987. ACM.

[7] L. Baugh, N. Neelakantam, and C. Zilles. Using hardware memory protection to build a high-performance, strongly atomic hybrid transactional memory. In *In Proceedings of the 35th 8 International Symposium on Computer Architecture*, 2008.

[8] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin. Making the fast case common and the uncommon case simple in unbounded transactional memory. *SIGARCH Comput. Archit. News*, 35(2):24–34, 2007.

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

[10] J. a. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63:172–185, December 2006.

[11] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. Section 23.2: The algorithms of Kruskal and Prim, 2009.

[12] M. Couceiro, P. Romano, N. Carvalho, and L. Rodrigues. D2STM: Dependable distributed software transactional memory. In *PRDC '09: Proc. 15th Pacific Rim International Symposium on Dependable Computing*, nov 2009.

[13] P. Damron, A. Fedorova, Y. Lev, V. Luchangco, M. Moir, and D. Nussbaum. Hybrid transactional memory. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 336–346, New York, NY, USA, 2006. ACM.

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

[15] R. G. Gallager, P. A. Humblet, and P. M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Program. Lang. Syst.*, 5(1):66–77, 1983.

[16] J. Garay, S. Kutten, and D. Peleg. A sub-linear time distributed algorithm for minimum-weight spanning trees. In *Foundations of Computer Science, 1993. Proceedings., 34th Annual Symposium on*, pages 659 –668, 3-5 1993.

[17] R. Guerraoui and M. Kapalka. The semantics of progress in lock-based transactional memory. *SIGPLAN Not.*, 44:404–415, January 2009.

[18] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun. Transactional memory coherence and consistency. In *in Proc. of ISCA*, page 102, 2004.

[19] T. Harris and K. Fraser. Language support for lightweight transactions. *ACM SIGPLAN Notices*, (38), 2003.

[20] T. Harris, S. Marlow, S. Peyton-Jones, and M. Herlihy. Composable memory transactions. In *PPoPP '05: Proceedings of the tenth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 48–60, New York, NY, USA, 2005. ACM.

[21] M. Herlihy, V. Luchangco, and M. Moir. A flexible framework for implementing software transactional memory. volume 41, pages 253–262, New York, NY, USA, October 2006. ACM.

[22] M. Herlihy, V. Luchangco, M. Moir, and W. N. Scherer. Software transactional memory for dynamic-sized data structures. In *In Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing*, pages 92–101. ACM Press, 2003.

[23] 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.

[24] M. Herlihy and Y. Sun. Distributed transactional memory for metric-space networks. In *In Proc. International Symposium on Distributed Computing (DISC 2005)*, pages 324–338. Springer, 2005.

[25] T. Johnson. Characterizing the performance of algorithms for lock-free objects. *Computers, IEEE Transactions on*, 44(10):1194 –1207, Oct. 1995.

[26] C. Kotselidis, M. Ansari, K. Jarvis, M. Luján, C. Kirkham, and I. Watson. DiSTM: A software transactional memory framework for clusters. In *ICPP '08: Proceedings of the 2008 37th International Conference on Parallel Processing*, pages 51–58, Washington, DC, USA, 2008. IEEE Computer Society.

[27] S. Kumar, M. Chu, C. J. Hughes, P. Kundu, and A. Nguyen. Hybrid transactional memory. In *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '06, pages 209–220, New York, NY, USA, 2006. ACM.

[28] S. Kutten and D. Peleg. Fast distributed construction of small k-dominating sets and applications. *Journal of Algorithms*, 28(1):40 – 66, 1998.

[29] B. Liskov, M. Day, M. Herlihy, P. Johnson, and G. Leavens. Argus reference manual. Technical report, Cambridge University, Cambridge, MA, USA, 1987.

[30] K. Manassiev, M. Mihailescu, and C. Amza. Exploiting distributed version concurrency in a transactional memory cluster. In *PPoPP '06*, pages 198–208. ACM Press, Mar 2006.

[31] J. Mankin, D. Kaeli, and J. Ardini. Software transactional memory for multicore embedded systems. *SIGPLAN Not.*, 44(7):90–98, 2009.

[32] V. J. Marathe, M. F. Spear, C. Heriot, A. Acharya, D. Eisenstat, W. N. S. III, and M. L. Scott. Lowering the overhead of nonblocking software transactional memory. *Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT)*, June 2006.

[33] M. Moir. Practical implementations of non-blocking synchronization primitives. In *In Proc. of 16th PODC*, pages 219–228, 1997.

[34] K. E. Moore. Thread-level transactional memory. In *Wisconsin Industrial Affiliates Meeting*. Oct 2004. Wisconsin Industrial Affiliates Meeting.

[35] 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.

[36] M. Philippsen and M. Zenger. Java Party transparent remote objects in Java. concurrency practice and experience, 1997.

[37] Y. Raz. The Dynamic Two Phase Commitment (D2PC) Protocol. In *ICDT '95: Proceedings of the 5th International Conference on Database Theory*, pages 162–176, London, UK, 1995. Springer-Verlag.

[38] D. P. Reed. Naming and synchronization in a decentralized computer system. Technical report, Cambridge, MA, USA, 1978.

[39] 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.

[40] T. Riegel, P. Felber, and C. Fetzer. A lazy snapshot algorithm with eager validation. In S. Dolev, editor, *Distributed Computing*, Lecture Notes in Computer Science, pages 284–298. Springer Berlin / Heidelberg, 2006.

[41] 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).

[42] 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.

[43] M. M. Saad and B. Ravindran. Distributed Hybrid-Flow STM : Technical Report. Technical report, ECE Dept., Virginia Tech, December 2010.

[44] M. M. Saad and B. Ravindran. RMI-DSTM: Control Flow Distributed Software Transactional Memory: Technical Report. Technical report, ECE Dept., Virginia Tech, February 2011.

[45] M. M. Saad and B. Ravindran. Transactional Forwarding Algorithm : Technical Report. Technical report, ECE Dept., Virginia Tech, January 2011.

[46] W. N. Scherer, III and M. L. Scott. Advanced contention management for dynamic software transactional memory. In *PODC '05: Proceedings of the twenty-fourth annual ACM symposium on Principles of distributed computing*, pages 240–248, New York, NY, USA, 2005. ACM.

[47] W. N. Scherer III and M. L. Scott. Contention management in dynamic software transactional memory. In *PODC '04: Proceedings of Workshop on Concurrency and Synchronization in Java Programs.*, NL, Canada, 2004. ACM.

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

[49] 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.

[50] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A bytecode translator for distributed execution of legacy Java software. In *In European Conference on Object-Oriented Programming (ECOOP*, 2001.

[51] E. Tilevich and Y. Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *In Proceedings of the European Conference on Object-Oriented Programming (ECOOP*, 2002.

[52] R. A. F. B. R. Veldema and H. E. Bal. Distributed shared memory management for java. In *In ASCI2000*, page 256, 2000.

[53] W. Yu and A. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency: Practice and Experience*, 9(11):1213–1224, 1997.

[54] 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.

[55] 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.