# Distributed Hybrid-Flow STM

[Technical Report]

Mohamed M. Saad
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
msaad@vt.edu

Binoy Ravindran
ECE Dept., Virginia Tech
Blacksburg, VA 24061, USA
binoy@vt.edu

## ABSTRACT

We present *HyFlow* — a distributed software transactional memory (D-STM) framework for distributed concurrency control. Lock-based concurrency control suffers from drawbacks including deadlocks, livelocks, and scalability and composability challenges. These problems are exacerbated in distributed systems due to their distributed versions which are more complex to cope with (e.g., distributed deadlocks). STM and D-STM are promising alternatives to lock-based and distributed lock-based concurrency control for centralized and distributed systems, respectively, that overcome these difficulties. HyFlow is a Java framework for D-STM, with pluggable support for directory lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols. HyFlow exports a simple distributed programming model that excludes locks: using (Java 5) annotations, atomic sections are defined as transactions, in which reads and writes to shared, local and remote objects appear to take effect instantaneously. No changes are needed to the underlying virtual machine or compiler. We describe HyFlow's architecture and implementation, and report on experimental studies comparing HyFlow against competing models including Java remote method invocation (RMI) with mutual exclusion and read/write locks, distributed shared memory (DSM), and directory-based D-STM. Our studies show that HyFlow outperforms competitors by as much as 40-190% on a broad range of transactional workloads on a 72-node system, with more than 500 concurrent transactions.

## 1. INTRODUCTION

Lock-based synchronization is inherently error-prone. Coarse-grained locking, in which a large data structure is protected using a single lock is simple and easy to use, but permits little concurrency. In contrast, with fine-grained locking [44, 55], in which each component of a data structure (e.g., a bucket of a hash table) is protected by a lock, programmers must acquire necessary and sufficient locks to obtain maximum concurrency without compromising safety. Both these situations are highly prone to programmer errors. In addition, lock-based code is non-composable. For example, atomically moving an element from one hash table to another using those tables' (lock-based) atomic methods is difficult: if the methods internally use locks, a thread cannot simultaneously acquire and hold the locks of the two tables' methods; if the methods were to export their locks, that will compromise safety. Furthermore, lock-inherent problems such as deadlocks, livelocks, lock convoying, and priority inversion haven't gone away. For these reasons, lock-based concurrent code is difficult to reason about, program, and maintain.

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. For example, RPC calls, while holding locks, can become remotely blocked on other calls for locks, causing distributed deadlocks. Distributed versions of livelocks, lock convoying, priority inversion, and scalability and composability challenges similarly occur.

Transactional memory (TM) [33] is a promising alternative to lock-based concurrency control. With TM, programmers write concurrent code using threads, but organize code that read/write shared objects as transactions, which appear to execute atomically. Two transactions conflict if they access the same object and one access is a write. When that happens, a contention manager [70] resolves the conflict by aborting one and allowing the other to proceed to commit, yielding (the illusion of) atomicity. Aborted transactions are re-started, often immediately. Thus, a transaction ends by either committing (i.e., its operations take effect), or by aborting (i.e., its operations have no effect). In addition to providing a simple programming model, TM provides performance comparable to highly concurrent, fine-grained locking [6]. Numerous multiprocessor TM implementations have emerged in software (STM) [32, 34, 38, 71], hardware (HTM) [31,39], and in a combination (HybridTM) [6,22,58].

Similar to multiprocessor TM, distributed software transactional memory (or D-STM) is an alternative to lock-based distributed concurrency control. D-STM can be supported in any of the classical distributed programming models, including a) control flow [9,51,73], where objects are immobile and transactions invoke object operations through RPCs; b) dataflow [60, 74], where transactions are immobile, and ob-

```
public class BankAccount implements
    IDistinguishable {
  ....
  @Override
  public Object getId() {
    return id;
  }

  @Remote  @Atomic{retries=100}
  public void deposit(int dollars) {
    amount = amount + dollars;
  }

  @Remote  @Atomic
  public boolean withdraw(int dollars) {
    if(amount>=dollars) return false;
      amount = amount - dollars;
    return true;
  }
}
```

```
public class TransferTransaction {
  @Atomic{retries=50}
  public boolean transfer(String accNum1,
      String accNum2, int amount) {
    BankAccount account1 =
        ObjectAccessManager.open(accNum1);
    BankAccount account2 =
        ObjectAccessManager.open(accNum2);

    if(!account1.withdraw(amount))
      return false;
    account2.deposit(amount);

    return true;
  }
}
```

**Figure 1: A bank transaction using an atomic TM method.**

jects are migrated to invoking transactions; and c) a hybrid model (e.g., [17]) where transactions or objects are migrated, heuristically, based on properties such as access profiles, object size, or locality. The different models have their concomitant tradeoffs.

The least common denominators for supporting D-STM in any distributed programming model include mechanisms/protocols for directory lookup [23, 40, 41, 77, 78], transactional synchronization and recovery [17, 21, 48, 53, 66], contention management [69, 70], cache coherence, and network communication. We present HyFlow — a Java D-STM framework that provides pluggable support for these mechanisms/protocols as modules. HyFlow exports a simple distributed programming model that excludes locks: atomic sections are defined as transactions using (Java 5) annotations. Inside an atomic section, reads and writes to shared, local and remote objects appear to take effect instantaneously. No changes are needed to the underlying virtual machine or compiler.

Figure 1 shows example transactional code in HyFlow, in which two bank accounts are accessed and an amount is atomically transferred between them. 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 `withdraw` and `deposit` methods have been composed into the higher-level atomic `transfer` operation. A conflicting transaction is transparently retried. Note that the location of the bank account is hidden from the program. It may be cached locally, migrate to the current node, or accessed remotely using remote calls, which is transparently accomplished by HyFlow.

Multiprocessor TM has been intensively studied, resulting in a number of implementations. Example HTM implementations include TCC [31], UTM [5], OneTM [16], and LogSPoTM [30]. They often extend multiprocessor cache coherence protocols, or modify underlying hardware to support transactions. Example STM implementations include DSTM2 [37], RSTM [54], and Deuce [47]. They often use basic atomic hardware primitives (e.g., compare-and-swap) to provide atomicity, and thread-local memory locations are used to provide consistent view of memory. Example HybridTM implementations include LogTM [58], HyTM [49], and McRT-STM [6].

D-STM implementations also exist. Examples include Cluster-STM [17], $D^2STM$ [21], DiSTM [48], and Cloud-TM [67]. 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 (PGAS) [2]) and often needs compiler or virtual machine modifications (e.g., JVSTM [18]), or assume specific architectures (e.g., commodity clusters).

HyFlow supports both dataflow and control flow models, and ensures distributed transactional properties including atomicity, consistency, and isolation. HyFlow's architecture is module-based, with well-defined APIs for further plugins. Default implementations exist for all needed modules. The framework currently includes two algorithms to support distributed memory transactions: the transactional forwarding algorithm (TFA) [68], and a distributed variant of the UndoLog algorithm [65]. A wide range of transaction contention management policies (e.g., Karma, Aggressive, Polite, Kindergarten, Eruption [69, 70]) are included in HyFlow. Four directory protocols [23, 41, 77] are implemented in HyFlow to track objects which are distributed over the network. HyFlow uses a voting algorithm, the dynamic two phase commitment protocol (D2PC) [62], to support control flow transactions. Network communication is supported using protocols including TCP [20], UDP [61], and SCTP [72]. We also implement a suite of distributed benchmark applications in HyFlow, which are largely inspired by the multiprocessor STM STAMP benchmark suite [19], to evaluate D-STM.

We experimentally evaluated HyFlow against competing models including Java remote method invocation (RMI) with mutual exclusion and read/write locks, distributed shared memory (DSM), and directory-based D-STM. Our studies show that HyFlow outperforms competitors by as much as 40-190% on a broad range of transactional workloads on a 72-node system, with more than 500 concurrent transactions.

The paper's central contribution is HyFlow — the first ever D-STM framework implementation. HyFlow provides several advantages over existing D-STM implementations including pluggable support for D-STM algorithms, without changes to the underlying virtual machine or compiler. The framework also provides a testbed for the research community to design, implement, and evaluate algorithms for D-STM. We hope that this will increase the momentum 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 HyFlow's system architecture and underlying mechanisms. Section 4 formally analyzes dataflow and control flow D-STM models, and establishes their tradeoff. In Section 5, we experimentally evaluate HyFlow against competing efforts. We conclude in Section 6.

## 2. RELATED WORK

***Transactional Memory.*** The classical solution for handling shared memory during concurrent access is lock-based techniques [7,45], where locks are used to protect shared objects. Locks have many drawbacks including deadlocks, livelocks, lock-convoying, priority inversion, non-composability, and the overhead of lock management. TM, proposed by Herlihy and Moss [39], is an alternative approach for shared memory 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 [32, 34, 37, 38, 54, 56, 71], HTM [5,16,31,39,57], and HyTM [6,12,22,49, 58]. STM has relatively larger overhead due to transaction management and architecture-independence. HTM has the lowest overhead, but assumes architecture specializations. HyTM seeks to combine the best of HTM and STM.

***Distributed Shared Memory.*** Supporting shared memory access in distributed systems has been extensively studied through the DSM model. Earlier DSM proposals were page-based [4, 8, 27, 50] that provide sequential consistency using single-writer/multiple-reader protocol at the level of memory pages. Though they still have a large user base, they suffer from several drawbacks including *false sharing*. This problem occurs when two different locations, located in the same page, are used concurrently by different nodes, causing the page to "bounce" between nodes, even though there is no shared data [26]. In addition, DSM protocols that provide sequential consistency have poor performance due to the high message overhead incurred [4]. Furthermore, single-writer/multiple-reader protocols often have "hotspots," degrading their performance. Also, most DSM implementations are platform-dependent and does not allow node heterogeneity.

Variable-based DSM [13, 14] provides language support for DSM based on shared variables, which overcomes the false-sharing problem and allows the use of multiple-writer/multiple-reader protocols. With the emergence of object-oriented programming, object-based DSM implementations were introduced [9,51,60,73,74] to facilitate object-oriented parallel applications.

- D-STM
  - Control Flow [9, 51, 73]
  - Data Flow
    - Single version [25, 36]
      - Directory Based
        - Tightly coupled [41]
        - Variable [10, 23, 40, 77]
      - Invalidate [64]
      - Replicate [3, 11, 52]
    - Multiversion
      - Linearizable [43]
      - Non-Linearizable [65]
  - Hybrid Flow [17]

**Figure 2: A Distributed STM taxonomy**

***Distributed STM.*** Similar to multiprocessor STM, D-STM was proposed as an alternative to lock-based distributed concurrency control. Figure 2 shows a taxonomy of different D-STM designs. D-STM models can be classified based on the mobility of transactions and objects. Mobile transactions [9, 51, 73] use an underlying mechanism (e.g., RMI) for invoking operations on remote objects. The mobile object model [40, 60, 67, 74, 77] allows objects to move through the network to requesting transactions, and guarantees object consistency using cache coherence protocols. Usually, these protocols employ a directory that can be tightly coupled with its registered objects [41], or permits objects to change their directory [10, 23, 40, 77].

The mobile object model can also be classified based on the number of active object copies. Most implementations assume a single active copy, called *single version.* Object changes can then be a) applied locally, invalidating other replicas [64], b) applied to one object (e.g., latest version of the object [25, 36]), which is discovered using directory protocols [23,41], or c) applied to all replicated objects [3,11, 52]. In contrast, multiversion concurrency control (MVCC) proposals allow multiple copies or replicas of an object in the network [43,65]. The MVCC models often favor performance over linearizable execution [42]. For example, in [65], reads and writes are decoupled to increase transaction throughput, but allows reading of older versions instead of the up-to-date version to prevent aborts.

System architecture and the scale of the targeted problem can affect design choices. With a small number of nodes (e.g., 10) interconnected using message-passing links, cache-coherent D-STM (cc D-STM) [23,40,77] is appropriate. However, for a cluster computer, in which a group of linked computers work closely together to form a single computer, researchers have proposed cluster D-STM [17, 21, 48, 53, 66]. 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.

Herlihy and Sun proposed cc D-STM [40]. They present a dataflow model, where transactions are immobile and objects are mobile. Object conflicts and object consistency are managed and ensured, respectively, by contention management and cache coherence protocols. In [40], they present a cache-coherence protocol, called Ballistic. Ballistic's hierarchical structure degrades its scalability—e.g., whenever a node joins or departs the network, the whole structure has to be rebuilt. This drawback is overcome in Zhang and Ravindran's Relay protocol [77, 78], which improves scalability by using a peer-to-peer structure. Relay assumes encounter time object access, which is applicable only for pessimistic STM implementations, which, relative to optimistic approaches, suffer from large number of conflicts [24]. Saad and Ravindran proposed an object-level lock-based algorithm [68] with lazy acquisition. No central clocking (or ticketing) mechanism is required. Network traffic is reduced by limiting broadcasting to just the object identifiers. Transactions are immobile, objects are replicated and detached from any "home" node, and they ensure a single writable copy of each object.

In [17], Bocchino *et. al.* proposed a word-level cluster D-STM. They decompose a set of existing cache-coherent STM designs into a set of design choices, and select a combination of such choices to support their design. However, each processor is limited to one active transaction at a time. Also, no progress guarantees are provided, except for deadlock-freedom. In [53], Manassiev *et. al.* present a page-level distributed concurrency control algorithm for cluster D-STM, which automatically detects and resolves conflicts caused by data races for distributed transactions accessing shared memory data. In their algorithm, page differences are broadcast to all other replicas, and a transaction commits successfully upon receiving acknowledgments from all nodes. A central timestamp is employed, which allows only a single update transaction to commit at a time.

Kotselidis *et. al.* present the DiSTM [48] object-level, cluster D-STM framework, as an extension of DSTM2 [37]. They compare three cache-coherence protocols on benchmarks for clusters. They show that, under the TCC protocol [31], DiSTM induces large traffic overhead at commit time, as a transaction broadcasts its read/write sets to all other transactions, which compare their read/write sets with those of the committing transaction. Using lease protocols [28], this overheard is eliminated. However, an extra validation step is added to the master node, as well as bottlenecks are created upon acquiring and releasing the leases. These implementations assume that every memory location is assigned to a *home* processor that maintains its access requests. Also, a central, system-wide ticket is needed at each commit event for any update transaction (except [17]).

Couceiro *et. al.* present $D^2STM$ [21]. Here, STM is replicated on distributed system nodes, and strong transactional consistency is enforced at commit time by a non-blocking distributed certification scheme. In [46], Kim and Ravindran develop a D-STM transactional scheduler, called Bi-interval, that optimizes the execution order of transactional operations to minimize conflicts, yielding throughput improvement of up to 200%. Romano *et. al.* extend cluster D-STM for Web services [66] and Cloud platforms [67].

HyFlow is an object-level D-STM framework, with pluggable support for the least common D-STM denominators, including directory lookup, transactional synchronization and recovery, contention management, cache coherence, and network communication. It supports both control and data flow, and implements a variety of algorithms as defaults. In addition, it doesn't require any changes to the underlying virtual machine or compiler, unlike [2, 18].

## 3. HYFLOW ARCHITECTURE

### 3.1 System Model

We consider an asynchronous distributed system model, similar to Herlihy and Sun [40], consisting of a set of $N$ nodes $N_1, N_2, ....., N_n$, communicating through weighted message-passing links $E$. Let $G = (N, E, c)$ be an undirected graph representing the network, where $c$ is a function that defines the link communication cost. Let $M$ denote the set of messages transferred in the network, and $Size(M_i)$ the size of a message $M_i$. A message could be a remote call request, vote request, resource publish message, or any type of message defined in HyFlow's protocols. A fixed minimum spanning tree $S$ or $G$ is used for broadcasting. Thus, the cost of message broadcasting is $O(|N|)$, which we define as the constant $\Omega$.

We assume that each shared object has an unique identifier. We use a grammar similar to the one in [29], but extend it for distributed systems. Let $O = \{o_1, o_2, ...\}$ denote the set of objects shared by transactions. An object may be replicated or may migrate to any other node. Without loss of generality, objects export only *read* and *write* methods (or operations). Thus, we consider them as shared registers. Let $T = \{T_1, T_2, \ldots\}$ denote the set of transactions. Each transaction has an unique identifier, and is invoked by a node (or process) in a distributed system of $N$ nodes. We denote the sets of shared objects accessed by transaction $T_k$ for read and write as *read-set*($T_k$) and *write-set*($T_k$), respectively. A transaction can be in one of three states: *active*, *aborted*, or *committed*. When a transaction is aborted, it is retried by the node again using a different identifier.

Every object has, at least, one "owner" node that is responsible for handling requests from other nodes for the owned object. Let $Own(O_i)$ and $Size(O_i)$ be functions that represent the owner and size of object $O_i$, respectively. In the data-flow model, a cache-coherence protocol locates the current cached copy of the object in the network, and moves it to the requesting node's cache. Under some circumstances, the protocol may change the object's owner to a new owner. Changes to the ownership of an object occurs at the successful commit of the object-modifying transaction. At that time, the new owner broadcasts a *publish* message with the owned object identifier.

In the control flow model, any node that wants to read from, or write to an object, contacts the object's owner using a remote call. The remote call may in turn produce 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 Architecture

Figure 3 shows the nodal architecture of HyFlow. Five modules and a *runtime handler* form the basis of the architecture. The modules include the *Transaction Manager*, *Instrumentation Engine*, *Object Access Module*, *Transaction Validation Module*, and *Communication Manager*.

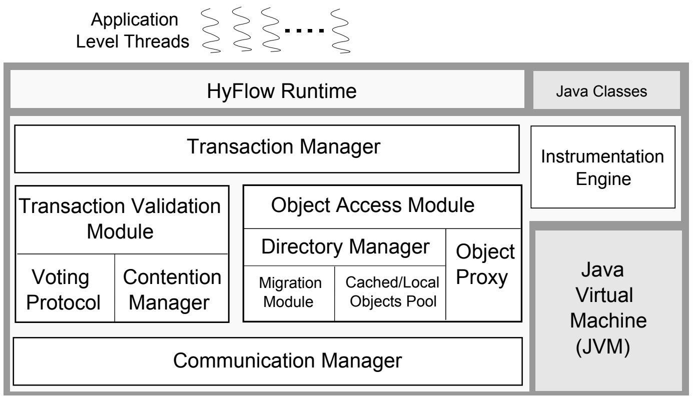


**Figure 3: HyFlow Node Architecture**

The HyFlow runtime handler represents a standalone entity that delegates application-level requests to the framework. HyFlow uses run-time instrumentation to generate transactional code, like other (multiprocessor) STM such as Deuce [47], yielding almost two orders of magnitude superior performance than reflection-based STM (e.g., [37]).

The Transaction Manager contains mechanisms for ensuring a consistent view of memory for transactions, validating memory locations, and retrying transactional code when needed. Based on the access profile and object size, object migration is permitted.

The Instrumentation Engine modifies class code at runtime, adds new fields, and modifies annotated methods to support transactional behavior. Further, it generates callback functions that work as "hooks" for Transaction Manager events such as onWrite, beforeWrite, beforeRead, etc.

Every node employs a Transaction Manager, which runs locally and handles local transactional code. The Transaction Manager treats remote transactions and local transactions equally. Thus, the distributed nature of the system is seamless at the level of transaction management.

The Object Access Module has three main tasks: 1) providing access to the object owned by the current node, 2) locating and sending access requests to remote objects, and 3) retrieving any required object meta-data (e.g., latest version number). Objects are located with their IDs using the Directory Manager, which encapsulates a directory lookup protocol [23, 41, 77]. Upon object creation, the Directory Manager is notified and publishes the object to other nodes. The Migration Module decides when to move an object to another owner or keep it locally. The purpose of doing so is to exploit object locality and reduce the overall communication traffic between nodes.

The Transaction Validation Module ensures data consistency by validating transactions upon their completion. It uses two sub-modules:

- *Contention Manager.* This sub-module is consulted when conflicts occur—i.e., when two transactions access a shared object, and one access is a write. When local transactions conflict, a contention management policy (e.g., Karma, Aggressive, Polite, Kindergarten, Eruption [69, 70]) is used to abort or postpone one of the conflicting transactions. However, when one of the conflicting transactions is remote, the contention policy decision is made globally based on heuristics (we explain this later in Section 3.2.3).
- *Global Voting handler.* In order to validate a transaction based on control flow, a global decision must be made across all participating nodes. This sub-module is responsible for collecting votes from other nodes and make a global commit decision such as by a voting protocol (e.g., D2PC [62]).

### *3.2.1 Instrumentation Engine*

Instrumentation is a Java feature that allows the addition of byte-codes to classes at run-time. In contrast with reflection, instrumentation works just once at class load time, which incurs much less overhead. HyFlow's Instrumentation Engine (HyIE) is a generic Java source code processor, which inserts transactional code at specified locations in a given source code. HyIE employs *Annotations* — a Java 5 feature that provides runtime access to syntactic form of metadata defined in source code, to recognize portions of code that need to be transformed. HyIE is built as an extension of the Deuce (multiprocessor) STM [47], which is based on ASM [15], a Java bytecode manipulation and analysis framework.

Like Deuce, 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. The Transaction Manager need not handle local method variables as part of a transaction.

Any distributed class must implement the `IDistinguishable` interface with a single method `getId()`. The purpose of this restriction is to decouple object references from their memory locations. HyIE detects any loaded class of type `IDistinguishable` and transforms it to a transactional version. Further, it instruments every class that may be used within transactional code. 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 manager. 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.

- ***@Remote methods.*** RMI-like code is generated to handle remote method calls at remote objects. In the control flow model, the Directory Manager can open the object, but cannot move it to the local node. An object appears to application code as a local object, while transformed methods call their corresponding original methods at the remote object.

Figure 4 shows part of the instrumented version of a `BankAccount` class defined in Figure 1.

It is worth noting that the *closed nesting* model [63], which extends the isolation of an inner transaction until the top-level transaction commits, is implicitly implemented. HyIE "flattens" nested transactions into the top-level one, resulting in a complete abort on conflict, or allow partial abort of inner transactions. Whenever an atomic method is called within the scope of another atomic method, the duplicate method is called with the parent's `Context` object, instead of the instrumented version.

### *3.2.2 Object Access Module*

During transaction execution, a transaction accesses one or more shared objects. The Directory Manager delegates access to all shared objects. An object may reside at the current node. If so, it is accessed directly from the *local object pool.* Or, it may reside at another node, and if so, it is considered as a *remote object.* Remote objects may be accessed differently according to the transaction execution model—i.e., control or dataflow. In the dataflow model, a *Migration Module* guarantees local access to the object. It can move the object, or copy it to the current node, and update the directory accordingly. In the control flow model, a *Proxy Module* provides access to the object through an empty instance of the object "facade" that acts as a proxy to the remote object. At the application level, these details are hidden, resulting in an uniform access interface for all objects.

It is interesting to see how the example in Figure 1 works using the dataflow and control flow models. Assume that the two bank accounts accessed in this example reside at different nodes. In the dataflow model, the transaction will call the Object Access Manager, which in turn, will use the Directory Manager to retrieve the required objects. The Directory Manager will do so according to the underlying implementation and contention management policy. Eventually objects (or copies of them) will be transferred to the current node. Upon completion, the transaction will be validated and the new versions of the objects will be committed.

Now, let us repeat the scenario using the control flow model. In this case, the Object Access Manager will employ an *Object Proxy* to obtain proxies to the remote object. Remote calls will be sent to the original objects. As we explain in the next section, once the transaction completes, a voting protocol will decide whether to commit the transaction's changes or to retry again.

```
public class BankAccount implements
    IDistinguishable {
  // Remote access flag
  boolean remote_obj$ = false;
  ....
  // Modified constructor
  BankAccount(String id){
    ....
    DirectoryManager.register(id, this);
  }
  ....
  // Synthetic duplicate method
  public void deposit(int dollars,
      Context c) {
    amount__Setter$(c, amount__Getter$(c)
        + dollars);
  }
  // Original method instrumented
  public void deposit(int dollars) {
    if(remote_obj$){
      //Invoke remote call
      Proxy.open(id).deposit(dollars);
      return;
    }
    //Transaction active thread
    Context context = ContextDelegator.
        getInstance();
    boolean commit = true;
    for (int i=100; i>0; --i) {
      //Initialize transaction
      context.init();
      try{
        //Try execute
        result=deposit(dollars,context);
      } catch(TransactionException ex) {
        commit = false;  //Aborted
      } catch(Throwable ex) {
        //Application Exception
        throwable = ex;
      }
      if(commit){
        if (context.commit()) {
          if (throwable == null)
            return result; //Committed
          //Rethrow Application exception
          throw (IOException)throwable;
        }
     }else{
        context.rollback(); //Rollback
        commit = true;
     }
    }
    //Maximum retries reached
    throw new TransactionException();
  }
}
```

**Figure 4: Instrumented version of BankAccount class.**

### *3.2.3 Transaction Validation Module*

The main task of this module is to guarantee transaction consistency, and to achieve system-wide progress. Recall that, in HyFlow, a transaction may be mobile or immobile. Thus, this module employs two sub-modules: 1) a Voting Manager, which is used for mobile transactions to collect votes from other participating nodes, and 2) a Global Contention Manager, which is consulted to resolve conflicting transactions (this is needed for both mobile and immobile transactions).

***Voting Manager*** 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 (sub-transactions) and undirected edges (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 [62], 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 (more information is available in [62]).

In D2PC, the originating node, which started the 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."

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, 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 (Lemma 3.1 proves this argument). 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.

LEMMA 3.1. *Under D2PC there will be one and only one node that will receive all the sent votes, assuming no partitions exist in the network.*

PROOF. The proof is by contradiction. We will assume that the protocol ends with two nodes that work as coordinators. Thus, each of these nodes must receive replies from

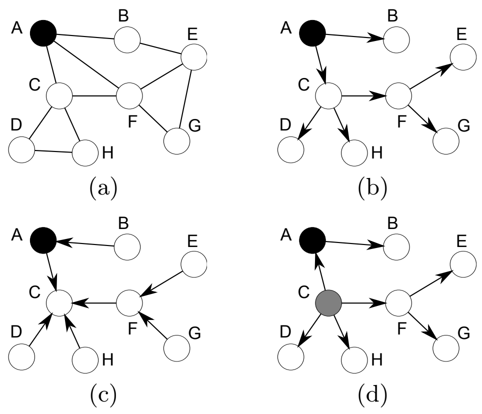


**Figure 5: (a) Call graph: each node represents a sub-transaction, and edges denote remote calls between them. (b) Originating node $A$ sends a PREPARE message that is forwarded to other nodes. (c) The vote is replied, and the coordinator is selected through the process of last-neighbor forwarding. (d) Coordinator node $C$ publishes the vote result**

*all* their neighbors, which implies that one node sent its votes to two of its neighbors. However, according to the described protocol, any node will send its collected votes to just *one* node (the last neighbor that did not reply). This contradicts the initial assumption, implying that there is exactly one coordinator at the end of the election protocol. □

Figure 5 shows a possible execution of D2PC for collecting votes for a transaction distributed over seven nodes.

***Global Contention Manager*** In contention manager-based STM implementations, the progress of the system is guaranteed by the contention policy. Having a special module for global contention management enables us to achieve effective decisions for resolving distributed transactional conflicts. Using classical non-distributed contention policies for this may be misleading and expensive. This module employs a set of heuristics for making such decisions, including the following:

- A local transaction that accesses local objects is aborted only when it conflicts with any distributed transaction.
- A distributed transaction that follows the dataflow model is favored over one that uses control flow, because the former is usually more communication-expensive.
- If two distributed transactions in the dataflow model conflict, we abort the one that a) accesses objects having a smaller total size, or b) communicates with less number of remote nodes.
- In all other cases, a contention manager employs any local methodology such as Karma, Aggressive, Polite, Kindergarten, or Eruption [70] [69].

## 4. ANALYSIS

We now illustrate the factors of communication and processing overhead through a comparison between control flow and dataflow D-STM models. A compromise between the two models can be used to design a hybrid D-STM model.

## 4.1 Dataflow Model

In the dataflow model, transactions are immobile, and objects move through the network. To estimate the transaction cost under this model, we state the following theorem. For simplicity, we consider a single transactional execution.

THEOREM 4.1. *The communication cost for a transaction $T_i$ running on node $N_S$ and accessing k remote objects $O_j$, $1 \leq j \leq k$, using the dataflow model is given by:* $DF_{cost}(T_i) = \sum_{1<j<k}[[Size(O_j) + \Pi(T_i, O_j)] * c(N_S, Own(O_j)) + \lambda + \beta * (1 - \Pi(T_i, O_j))]$. *Here, $\lambda$ is the lookup cost, $\beta$ is the directory update cost, and $\Pi$ is a function that returns 1 if the transaction accesses the object for read-only and 0 for read-write operations.*

PROOF. To execute a transaction using the dataflow model, three steps must be done for each remote object: locate the object, move the object, and validate or own the object.

There has been significant research efforts on object lookup (or discovery) protocols in distributed systems. Object (or service) lookup protocols can be either directory-based [9, 75, 76] or directory-less [1, 35, 59], according to the existence of a centralized directory that maintains the locations of services/resources. There could be one or more directories in the network or a single directory that is fully distributed.

Directory-based architectures suffer from scalability and single points-of-failure problems. Directory-less protocols can be classified as *push protocols* or *pull protocols*. In a push protocol, a node advertises its object to other nodes, and is responsible for updating other nodes with any changes to the provided object. In a pull protocol, when a node needs to access an object, it broadcasts a discover request to find the object provider. Usually, caching of object locations is exploited to avoid storms of discover requests. We denote the cost for object location by $\lambda$, which may differ according to the underlying protocol.

The cost for moving an object to the current node is proportional to the object size and the total communication cost between the current node and the object owner.

At commit time, a transaction needs to validate a read object, or obtain the ownership of the object, and thus will need to update the directory. $\beta$ is an implementation-specific constant that represents the cost to update the directory. It may vary from a single message cost as in Arrow and Relay directory protocols [23, 77], logarithmic in the size of nodes as in the Ballistic protocol [23], or may require message broadcasting over the entire network as in the Home directory protocol [41]. □

## 4.2 Control Flow Model

In contrast to the dataflow model, in the control flow model, a transaction can be viewed as being composed of a set of sub-transactions. Remote objects remain at their nodes, and are requested to do some operations on behalf of a transaction. Such operations may request other remote objects. For simplicity, we assume that the voting protocol will use a static minimum spanning tree $S$ for sending messages, and the nodes which are not interested in voting will not respond.

THEOREM 4.2. *The communication cost for a transaction $T_i$ running on a set of nodes $N_{t_i} = \{N_{1t_i}, N_{2t_i}, \ldots, N_{nt_i}\}$, and accessing k remote objects $O_j$, $1 \leq j \leq k$, using the control flow model is given by:* $CF_{cost}(T_i) = Voting(N_{t_i}) + \sum_{\substack{1<j<k \\ 1<s<n}}[c(N_{st_i}, Own(O_j)) * Calls(T_i, O_j) * \Theta(N_{st_i}, O_j)]$. *Here, $Calls$ is the number of method calls per object in transaction $T_i$, $Voting$ is the cost of collecting votes of a given set of nodes, and $\Theta$ is a function that returns 1 if a node needs to access a remote object during its execution, and 0 otherwise.*

PROOF. Let us divide the distributed transaction $T_i$ into a set of sub-transactions. Each sub-transaction is executed on a node, and during the sub-transaction, the node can access one (or more) remote object(s). The communication cost per each sub-transaction is the cost for accessing remote objects using remote calls. Each remote call requires a round-trip message for the request and its response. The total communication cost per node, is the sum of the costs of all sub-transactions which run on the node.

The second term of the equation of the theorem shows the aggregate cost for all nodes involved in executing the distributed transaction. The D2PC voting protocol [62] needs to broadcast at most three messages, one for requesting the votes, one for collecting the votes, and one for publishing the result of the global decision—i.e., $Voting(N_{t_i}) \leq 3 * \Omega$. □

## 4.3 Tradeoff

The previous two sections show that there is a trade-off between using the dataflow or control flow model for D-STM, which hinges on the number of messages and the size of objects transferred. The definition of functions $Calls$, $\Theta$, and $\Pi$ could be obtained either by code-analysis that identifies *object-object* relationships for objects defined as shared ones, or by transaction profiling based on past retries. The $Own(O)$ function is implemented as one of the functions of the Directory Manager, while $\lambda$, $\beta$, and $Voting$ functions are implementation-specific according to underlying protocols. Under a network with stable topology conditions, we can define a fixed communication cost metric.

Another factor that affects the communication cost is the locality of objects. Exploiting locality can reduce network traffic and thereby enhance D-STM performance. Consider an object, which is accessed by a group of geographically-close nodes, but far from the object's current location. Sending several remote requests to the object can introduce high network traffic, causing a communication bottleneck. The following two lemmas show the cost of moving an object versus sending remote requests to it.

LEMMA 4.3. *The communication cost introduced by relocating an object $O_j$ to a new node $N_i$ is given by:* $Reloc_{cost}(O_j, N_i) = \beta + Size(O_j) * c(N_i, Own(O_j))$.

PROOF. Object relocation or duplication recquires; i) updating the objects directory ($\beta$), and ii) moving or copying the object over the network, which is proportional to the object size and depends on the link costs between the source and distination nodes. □

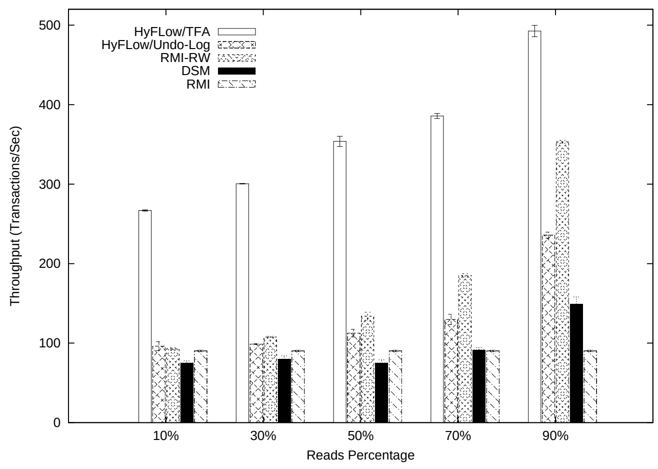
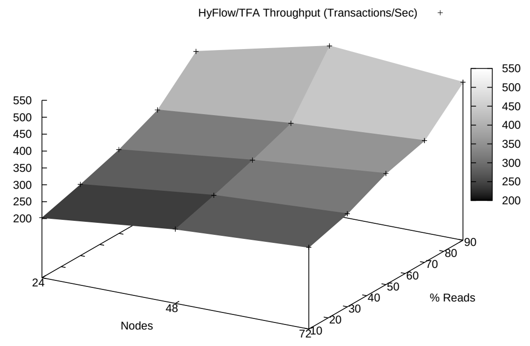
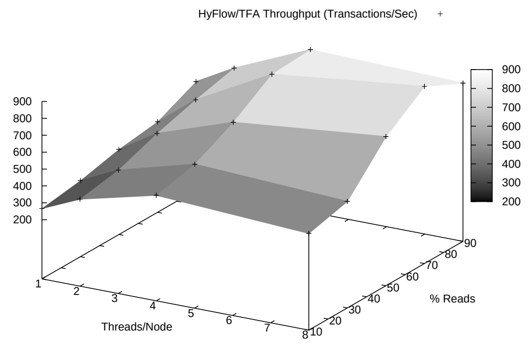


**Figure 6: Throughput of bank benchmark at different read percentages over 72 nodes.**



(a) Throughput under increasing number of nodes.



(b) Throughput under increasing number of threads per node.

**Figure 7: Scalability of HyFlow/TFA.**

LEMMA 4.4. *For a distributed transaction in the control flow model, the communication cost for sending a remote request to an object $O_j$ is given by:*
$Msg_{cost}(O_j, Own(O_j)) = \sum_{1<s<n}[c(N_{st_i}, Own(O_j)) * \Theta(N_{st_i}, O_j)]$.

PROOF. As distributed trabsaction can run over multiple nodes, a remote object may be accessed by some of the participating nodes. We formulate this using the $\Theta$ funtion, and aggregate the cost over all the participating nodes. □

We conclude that even in the control flow model, object relocation may be beneficial. For any object $O_j$ accessed by some transaction running on a set of nodes $N_{t_i}$ under the control flow model, if $Msg_{cost}(O_j, Own(O_j)) > Msg_{cost}(O_j, N_i) + Reloc_{cost}(O_j, N_i)$, then the object should be moved to node $N_i$.

## 5. EXPERIMENTAL RESULTS

***Bank Benchmark.*** We developed a distributed version of a banking application, which maintains a set of accounts distributed over bank branches. Two atomic transactions were implemented: *transfer transaction* (see Figure 1), which transfers a given amount between two accounts, and *total balance transaction* that computes the total balance for given accounts. Three versions of this benchmark were implemented. The first version uses Java RMI as the remote method invocation mechanism, and locks to guard critical sections. Two variants of locks were used, one using normal (mutual exclusion) spin locks, and the other using read-write locks. A random timeout mechanism was used to handle deadlock and livelock situations. The second version uses atomic transactions using two D-STM implementations in HyFlow: TFA [68] and UndoLog [65]. The Home directory manager was used to locate and trace object locations. The third version of the benchmark was based on a distributed shared memory (DSM) implementation using the Home directory protocol, like Jackal [64]. It uses the single-writer multiple-readers pattern.

***Testbed.*** We conducted the 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.

***Evaluation.*** Ten bank accounts were distributed equally on nodes, and hundred transactions were executed at each node. Figures 6 shows the throughput of the different schemes at 10%, 30%, 50%, 70% and 90% read-only transactions, respectively, under increasing number of nodes, which increases contention (with all else being equal). The confidence intervals of the data-points of the figure are in the 5% range. Figure 7 shows the scalability of HyFLow/TFA under increasing number of nodes and number of concurrent threads at each node. Figure 8 shows the effect of increasing the number of method calls per object on throughput, under control flow.

From Figure 6 we observe that HyFlow/TFA outperforms all other distributed concurrency control models by 40-190%.

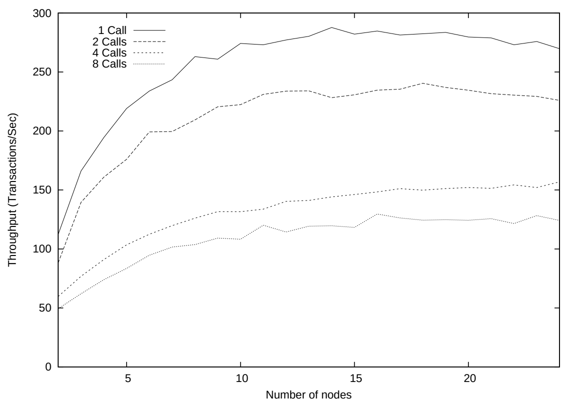


**Figure 8: Control flow throughput degradation under increasing number of calls per object.**

HyFlow/TFA is scalable and provides linear throughput at large number of nodes, while RMI throughput saturates after 15 nodes. For read-dominant transactions, HyFlow/TFA still gives a comparable performance against RMI with read-write locks. Further, from Figure 7, we observe the excellent scalability of HyFlow/TFA under high contention (72 nodes, more than 500 concurrent transactions).

UndoLog, which was not originally designed for D-STM, still gives comparable performance to DSM and RMI with mutual exclusion locks. However, relying on the Home directory for accessing objects increases the number of conflicts, which offset other STM features. Our experiments show that the number of conflicts in UndoLog is ten times more than that in HyFlow/TFA.

Figure 8 demonstrates the importance of employing locality of reference: in the control flow model, each remote call incurs a round-trip network delay, whereas in dataflow (e.g., TFA), an object is retrieved only once.

## 6. CONCLUSIONS

We presented HyFlow, a high performance pluggable, distributed STM that supports both dataflow and control flow distributed transactional execution. Our experiments show that HyFlow outperforms other distributed concurrency control models, with acceptable number of messages and low network traffic, thanks to a cache coherence D-STM algorithm called TFA.

The dataflow model scales well with increasing number of calls per object, as it permits remote objects to move toward geographically-close nodes that access them frequently, reducing communication costs. Control flow is beneficial under non-frequent object calls or calls to objects with large sizes. Our implementation shows that D-STM, in general, provides comparable performance to classical distributed concurrency control models, and exports a simpler programming interface, while avoiding dataraces, deadlocks, and livelocks.

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

## 7. REFERENCES

[1] UPnP Forum: Understanding universal plug and play white paper, 2000.

[2] Partitioned Global Address Space (PGAS), 2003.

[3] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8):26–34, 1986.

[4] 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.

[5] 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.

[6] B. S. andAli-Reza Adl-Tabatabai andRichard L. Hudson andChi Cao Minh andBen Hertzberg. McRT-STM: a high performance software transactional memorysystem for a multi-core runtime. In *PPOPP*, pages 187–197, 2006.

[7] T. Anderson. The performance of spin lock alternatives for shared-money multiprocessors. *Parallel and Distributed Systems, IEEE Transactions on*, 1(1):6 –16, Jan. 1990.

[8] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos. The Hyperion project: From data integration to data coordination. In *In: SIGMOD RECORD (2003*, 2003.

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

[10] H. Attiya, V. Gramoli, and A. Milani. COMBINE: An Improved Directory-Based Consistency Protocol. Technical report, EPFL, 2010.

[11] H. E. Bal, M. F. Kaashoek, and A. S. Tanenbaum. Orca: A language for parallel programming of distributed systems. *IEEE Trans. Softw. Eng.*, 18(3):190–205, 1992.

[12] 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.

[13] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Distributed shared memory based on type-specific memory coherence. In *In PPOPP*, pages 168–176. ACM, 1990.

[14] B. N. Bershad and M. J. Zekauskas. Midway: Shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical report, Carnegie-Mellon University, 1991.

[15] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. PPPJ '07, pages 135–144, New York, NY, USA, 2007. ACM.

[16] 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.

[17] 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.

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

[19] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization*, September 2008.

[20] V. G. Cerf and R. E. Icahn. A protocol for packet network intercommunication. *SIGCOMM Comput. Commun. Rev.*, 35:71–82, April 2005.

[21] 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.

[22] 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.

[23] 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.

[24] D. Dice, O. Shalev, and N. Shavit. Transactional Locking II. In *In Proc. of the 20th Intl. Symp. on Distributed Computing*, 2006.

[25] M. Factor, A. Schuster, and K. Shagin. A platform-independent distributed runtime for standard multithreaded Java. *Int. J. Parallel Program.*, 34(2):113–142, 2006.

[26] V. W. Freeh. Dynamically controlling false sharing in distributed shared memory. In *Proceedings of the 5th IEEE International Symposium on High Performance Distributed Computing*, HPDC '96, pages 403–, Washington, DC, USA, 1996. IEEE Computer Society.

[27] R. Friedman, M. Goldin, A. Itzkovitz, and A. Schuster. MILLIPEDE: Easy parallel programming in easy parallel programming in available distributed environments.

[28] C. Gray and D. Cheriton. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the twelfth ACM symposium on Operating systems principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM.

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

[30] R. Guo, H. An, R. Dou, M. Cong, Y. Wang, and Q. Li. Logspotm: a scalable thread level speculation model based on transactional memory. In *ACSAC 2008. 13th Asia-Pacific*, pages 1 –8, 2008.

[31] 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.

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

[33] T. Harris, J. Larus, and R. Rajwar. Transactional Memory, 2nd edition. *Synthesis Lectures on Computer Architecture*, 5(1):1–263, 2010.

[34] 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.

[35] S. Helal, N. Desai, V. Verma, and C. Lee. Konark - A Service Discovery and Delivery Protocol for Ad-Hoc Networks, 2003.

[36] M. Herlihy. The Aleph Toolkit: Support for scalable distributed shared objects. In *CANPC '99: Proceedings of the Third International Workshop on Network-Based Parallel Computing*, pages 137–149, London, UK, 1999. Springer-Verlag.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] M. Herlihy and M. P. Warres. A tale of two directories: implementing distributed shared objects in Java. In *JAVA '99: Proceedings of the ACM 1999 conference on Java Grande*, pages 99–108, New York, NY, USA, 1999. ACM.

[42] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12:463–492, 1990.

[43] R. Hickey. The clojure programming language. In *Proceedings of the 2008 symposium on Dynamic languages*, DLS '08, pages 1:1–1:1, New York, NY, USA, 2008. ACM.

[44] G. C. Hunt, M. M. Michael, S. Parthasarathy, and M. L. Scott. An efficient algorithm for concurrent priority queue heaps. *Inf. Process. Lett.*, 60:151–157, November 1996.

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

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

[47] G. Korland, N. Shavit, and P. Felber. Noninvasive concurrency with Java STM. In *Third Workshop on Programmability Issues for Multi-Core Computers (MULTIPROG)*, 2010.

[48] 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.

[49] 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.

[50] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM*, (7), 1989.

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

[52] J. Maassen, T. Kielmann, and H. E. Bal. Efficient replicated method invocation in Java. In *JAVA '00: Proceedings of the ACM 2000 conference on Java Grande*, pages 88–96, New York, NY, USA, 2000. ACM.

[53] 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.

[54] 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.

[55] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

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

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

[58] 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.

[59] M. Nidd. Timeliness of service discovery in DEAP space. In *ICPP '00: Proceedings of the 2000 International Workshop on Parallel Processing*, page 73, Washington, DC, USA, 2000. IEEE Computer Society.

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

[61] J. Postel. Rfc 768: User datagram protocol, internet engineering task force, August 1980.

[62] 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.

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

[64] A. A. Reeves and J. D. Schlesinger. JACKAL: A hierarchical approach to program understanding. In *WCRE '97: Proceedings of the Fourth Working Conference on Reverse Engineering (WCRE '97)*, page 84, Washington, DC, USA, 1997. IEEE Computer Society.

[65] 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.

[66] 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).

[67] 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.

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

[69] 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.

[70] 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.

[71] 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.

[72] R. R. Stewart and Q. Xie. *Stream control transmission protocol (SCTP): a reference guide.* Addison-Wesley

Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[73] 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.

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

[75] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol, 1997.

[76] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Salutation consortium: Salutation architecture specification. version 2.0c, 1999.

[77] 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.

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