Supporting STM in Distributed Systems: Mechanisms & Java Framework

Mohamed M. Saad & Binoy Ravindran

VT-MENA Program
Electrical & Computer Engineering Department
Virginia Polytechnic Institute and State University
An operation (or set of operations) appears to the rest of the system to occur instantaneously.

Example (Money Transfer):

......
......
from = from - amount
......
to = to + amount
......
Example (Money Transfer):

```
account1.lock()
account2.lock()
from = from - amount
to = to + amount
account1.unlock()
account2.unlock()
```

- Deadlock
- Livelock
- Starvation
- Priority Inversion
- Non-composable
- Non-scalable on multiprocessors
Distributed Atomicity

Multiple nodes
Message passing links
Objects are distributed over the network
Distributed transactions !!!
Current Approaches
- Remote Procedure Calls (RPC)
  - e.g. Java™ RMI
- Distributed Shared Memory
  - Home based
  - Directory based
  - Replication

Extending Transactional Memory concepts to Distributed Environment

Not designed for supporting atomicity
Inherited drawbacks of locks
High overhead
Requires significant code changes
Motivation

- Complex systems implies the need for distributed environment
- **Complexity** of current programming model
- **Distributed** deadlock, race conditions, ....
- **High** performance transactions
- The lack for **D-STM framework & testbed suit**
- **Locality** ... Locality ... Locality
- Towards **Hybrid** execution model (Hybrid Flow)
Contributions

- We present **HyFlow**, a distributed STM framework with modular design, and pluggable interface.
  **Testbed** suit as a distributed set of benchmarks

- **Simple programming model** based on code generation and **annotation** for accessing remote & atomic code

- We propose two mechanisms for **dataflow & control-flow** D-STM
Distributed STM Java framework, with pluggable support for: directory lookup protocols, transactional synchronization and recovery mechanisms, contention management policies, cache coherence protocols, and network communication protocols.

- Employ the correct execution model (data or control or hybrid)
- Focus more on business logic & less on remote access (stubs, MPI, ...) or transactional semantics (concurrency)
Dataflow model

Objects are mobile, transactions permanent at their invoked nodes
HyFlow

Execution Models

- Control-flow model
  - Immobile objects with mobile transactions
HyFlow

Execution Models

- Hybrid model
  - Automatically select suitable flow (data/control) according to access patterns and transaction costs/overhead
HyFlow

Object Access

- Changing ownership
- Copy / Replica
- Proxy
**HyFlow**

**Access Modes**

- **Write**
  - Exclusive Access & added to write set

- **Read**
  - Shared Access & added to read set

- **Shared**
  - Shared Access & not added to read set
  - Should be promoted at commit time to read or write
  - Useful for data structure implementations
  - Careful usage to preserve linearizability or opacity

```
<table>
<thead>
<tr>
<th></th>
<th>W</th>
<th>R</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>W</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>R</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>S</td>
<td>x</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
```
HyFlow Programming Model

- No special compiler, or underlying virtual machine modifications
- Uses Annotations @........
- Employs Instrumentation for code generation at load time
- Locates objects by “Locators” with three modes; shared, read & write
- Flat nesting model support
```java
class BankAccount{
    int amount;
    String id;

    BankAccount(String id){
        this.id = id;
    }

    @remote
    void getId(){    return this.id;   }

    @atomic @remote
    void deposit(int dollars){
        amount = amount + dollars;
    }

    @atomic @remote
    void withdraw(int dollars){
        amount = amount – dollars;
    }
}

class Transaction{
    @atomic { retries=50, timeout=1000 }
    void transfer(String acc1, String acc2, int amount){
        Locator&lt;BankAccount&gt; locator  = HyFlow.getLocator();
        BankAccount account1 = locator .locate(acc1);
        BankAccount account2 = locator .locate(acc2);
        account1.withdraw(amount);
        account2.deposit(amount);
    }
}
```
Dataflow algorithm (mobile objects/immobile transactions)

Rationale

- Every object associated with a versioned lock
- Every node has a local clock (version generator)
- Transaction reads clock when it starts $TC$
- Clocks
  - Objects requests are piggybacked with node clock
  - If recipient found incoming clock > local clock $\rightarrow$ advance its clock
  - **Transaction Forwarding mechanism**
- At commit time all object versions must be $< TC$
Snake D-STM

- Control-flow algorithm (immobile objects/mobile transactions)

- Rationale
  - Transaction moves between nodes, while objects are immobile
  - Each node has a portion of the write and read sets
  - Transaction metadata are detached from the transaction context
  - Distributed validation at commit time using voting mechanism

- Implementation
  - Undo-log & Write buffer variants
  - D2PC voting protocol
Experimental Results

- 120 nodes, 1.9 GHz each, 0.5~1 ms end-to-end delay
- 8 threads per node (1000 concurrent transactions)
- 50-200 sequential transactions
- ≈ 4 millions transactions
- 5% confidence interval (variance)
- Use 5 distributed benchmarks: Bank, Loan, Vacation, Linked List & Binary Search Tree.
Experimental Results

TFA Performance

![Graph showing TFA/Flow Speedup over RMI-RW and DSM]
Experimental Results

Snake TM Performance

![Graph showing the speedup of Snake/Hyflow over RMI-RW and DSM for different scenarios: Bank, Loan-high, Loan-low, Vacation-high, and Vacation-low. The speedup values range from 0 to 16.]
Experimental Results

Locality (Dataflow vs. Control-flow)

Bank Benchmark
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.
- The dataflow model scales well with increasing number of calls per object. It moves objects 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.
- We introduce Hybrid flow model analysis to understand the tradeoff between control-flow and data flow execution models.
Future Work

- Reduce retries overhead using **schedulers**
- **Hybrid flow** execution model
- Support **closed & open nesting** in distributed transactions
- **Multi-versioned** objects approach
Thanks

Please visit us at

www.hyflow.org