Distributed Software
Transactional Memories
Foundations, Algorithms and Tools

Maria Couceiro
(with Paolo Romano and Luís Rodrigues)
IST/ INESC-ID
Contents

- Part I: (Non-Distributed) STMs
- Part II: Distributed STMs
- Part III: Case-studies
- Part IV: Conclusions
Contents

- Part I: (Non-Distributed) STMs
- Part II: Distributed STMs
- Part III: Case-studies
- Part IV: Conclusions
(Non-Distributed) STMs

- Basic Concepts
- Example Algorithms
Basic Concepts

- Concurrent programming has always been a challenge
- One needs to control the concurrent access to shared data by multiple threads
- This is hard for most programmers.
- Concurrent programming has been a “niche”
Basic Concepts

- In the past:
  - More performance via faster CPUs

- Now:
  - More performance via more CPUs
  - Concurrent programming has to become mainstream
Basic Concepts

- Ideally
  - Performance would scale linearly with the number of cores
  - (with 8 cores we would have a program 8 times faster)

- Reality:
  - Speed up limited by % serial code
  - Small % can kill performance (Amdahl’s Law)
  - Say 25% of the program is serial
  - 8 cores = 2.9 speedup.
Basic Concepts

- Ideally
  - Performance would scale linearly with the number of cores
  - (with 8 cores we would have a program 8 times faster)

- Reality:
  - Small % of serial code can kill performance (Amdahl’s Law)
  - Say 25% of the program is serial
  - 32 cores = 3.7 speedup.
Basic Concepts

- Ideally
  - Performance would scale linearly with the number of cores
  - (with 8 cores we would have a program 8 times faster)

- Reality:
  - Small % of serial code can kill performance (Amdahl’s Law)
  - Say 25% of the program is serial
  - 128 cores = 3.9 speedup.
Basic Concepts

- It is hard or impossible to structure a program in a set of parallel independent tasks.
- We need efficient and simple mechanisms to manage concurrency.
Explicit synchronization

- One of the most fundamental and simple synchronization primitive is the lock

non-synchronized code;
lock ();
do stuff on shared data;
unlock ();
more non-synchronized code;
Many problems with locks

- Deadlock:
  - locks acquired in “wrong” order.

- Races:
  - due to forgotten locks

- Error recovery tricky:
  - need to restore invariants and release locks in exception handlers
Fine Grained Parallelism?

- Very complex:
  - Need to reason about deadlocks, livelocks, priority inversions.
  - Verification nightmare as bugs may be hard to reproduce.
- Make parallel programming accessible to the masses!!!
Concurrent Programming
Without Locks

- Lock-free algorithms.
- Hard to design and prove correct.
- Only for very specialized applications.
- Designed and implemented by top experts.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Abstractions for simplifying concurrent programming...

WE WANT YOU!

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Atomic transactions

atomic {
    access object 1;
    access object 2;
}

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Transactional Memories

- Hide away synchronization issues from the programmer.

- Advantages:
  - avoid deadlocks, priority inversions, convoying;
  - simpler to reason about, verify, compose.
TMs: where we are, challenges, trends

- Theoretical Aspects
  - Formalization of adequate consistency guarantees, performance bounds.

- Hardware support
  - Very promising simulation-based results, but no support in commercial processors.
TMs: where we are, challenges, trends

- Software-based implementations (STM)
  - Performance/scalability improving, but overhead still not satisfactory.

- Language integration
  - Advanced supports (parallel nesting, conditional synchronization) are appearing...
  - ...but lack of standard APIs & tools hampers industrial penetration.
TMs: where we are, challenges, trends

- Operating system support
  - Still in its infancy, but badly needed (conflict aware scheduling, transactional I/O).

- Recent trends:
  - Shift towards distributed environments to enhance scalability & dependability.
Run-time

How does it work?
- The run time implements concurrency control in an automated manner.

Two main approaches:
- Pessimistic concurrency control (locking).
- Optimistic concurrency control.
Example of pessimistic concurrency control

- Each item has a read/write lock.
- When an object is read, get the read lock.
  - Block if write lock is taken.
- When an object is written, get the write lock.
  - Block if read or write lock is taken.
- Upon commit/abort:
  - Release all locks.
Example of optimistic concurrency control

- Each item has a version number.
- Read items and store read version.
- Write local copy of items.
- Upon commit do atomically:
  - If all read items still have the read version (no other concurrent transaction updated the items)
    - then apply all writes (increasing the version number of written items).
  - Else,
    - abort.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Many, many, variants exist

- For instance, assume that two phase locking is used and a deadlock is detected. It is possible:
  - Abort both transactions.
  - Abort the oldest transaction.
  - Abort the newest transaction.
  - Abort the transaction that did less work.
For instance, assume that two phase locking is used and a deadlock is detected. It is possible:

- Abort both transactions
- Abort the oldest transaction
- Abort the newest transaction
- Abort the transaction that did less work

Each alternative offers different performance with different workloads.
How to choose?

- What is a correct behavior?
- Which safety properties should be preserved?
- Which liveness properties should be preserved?
How to choose?

- What is a correct behavior?
- Which safety properties should be preserved?
- Which liveness properties should be preserved?

To answer these questions we need a bit of theory.
Theoretical Foundations

- Safety:
  - What schedules are acceptable by an STM?
  - Is classic atomicity property appropriate?

- Liveness:
  - What progress guarantees can we expect from an STM?
Theoretical Foundations

- **Safety:**
  - What schedules are acceptable by an STM?
  - Is classic atomicity property appropriate?

- **Liveness:**
  - What progress guarantees can we expect from an STM?
Classic atomicity property

- A transaction is a sequence of read/write operations on variables:
  - sequence unknown a priori (otherwise called static transactions).
  - asynchronous (we do not know a priori how long it takes to execute each operation).

- Every operation is expected to complete.

- Every transaction is expected to abort or commit.
The execution of a set of transactions on a set of objects is modeled by a history.

A history is a total order of operation, commit and abort events.
Histories

Two transactions are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.
Histories

- Two transactions are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.

- Non-sequential:
Histories

- Two transactions are **sequential** (in a history) if one invokes its first operation after the other one commits or aborts; they are **concurrent** otherwise.

- Sequential:

```
OP C OP OP OP C
```
Histories

- A history is sequential if it has only sequential transactions; it is concurrent otherwise.
Histories

- A history is sequential if it has only sequential transactions; it is concurrent otherwise.
- Sequential:

```
OP C OP OP C OP C
```

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Histories

- A history is sequential if it has only sequential transactions; it is concurrent otherwise.

- Non-sequential:
Histories

- Two histories are **equivalent** if they have the same transactions.
Histories

- Two histories are equivalent if they have the same transactions

- Equivalent:

```
OP  C  OP  OP  C  OP  C
OP  C  OP  OP  OP  C  C
```
Two histories are **equivalent** if they have the same transactions.

**Non-equivalent:**

- First history: \(\text{OP} \rightarrow \text{C} \rightarrow \text{OP} \rightarrow \text{OP} \rightarrow \text{C} \rightarrow \text{OP} \rightarrow \text{C}\)
- Second history: \(\text{OP} \rightarrow \text{C} \rightarrow \text{OP} \rightarrow \text{OP} \rightarrow \text{OP} \rightarrow \text{C} \rightarrow \text{C}\)
What the programmer wants?

- Programmer does not want to be concerned about concurrency issues.
- Execute transactions “as if” they were serial
- No need to be “serially executed” as long as results are the same
Serializability’s definition
(Papa79 - View Serializability)

- A history $H$ of **committed** transactions is **serializable** if there is a history $S(H)$ that is:
  - equivalent to $H$
  - sequential
  - **every read returns the last value written**
Serializability

 Serializable?

 WO2(1) → RO2(1) → RO1(0) → C → WO1(1) → C
Serializable

 Serializable!

 WO2(1)  RO2(1)  RO1(0)  C  WO1(1)  C

 WO2(1)  RO1(0)  C  RO2(1)  WO1(1)  C
Serializable?

WO2(1) → RO2(0) → RO1(0) → C → WO1(1) → C
Serializability

Non-serializable!

- WO2(1) → RO2(0) → RO1(0) → C → WO1(1) → C
- RO2(0) → WO1(1) → C → WO2(1) → RO1(0) → C
- WO2(1) → RO1(0) → C → RO2(0) → WO1(1) → C
Opacity

 Serializable (blue aborts)?

WO2(1) → RO2(1) → RO1(0) → A → WO1(1) → C
Opacity

Serializable: only committed transactions matter!
Opacity

- In a database environment, transactions run SQL:
  - no harm if inconsistent values are read as long as the transaction aborts.

- This is not the same in a general programming language:
  - observing inconsistent values may crash or hang an otherwise correct program!
Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (y-x);

If T1 and T2 are atomic, the program is correct.
Opacity: example

Initially: \( x := 1; y := 2 \)

- \( T1: x := x+1; y := y+1 \)

- \( T2: z := \frac{1}{y-x}; \)

Otherwise...
Opacity: example

Initially: \( x := 1; \ y := 2 \)

- \( T1: \ x := x + 1; \ y := y + 1 \)
- \( T2: \ z := 1 / (y - x); \)

Otherwise...
Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (2-x);

Otherwise...
Opacity: example

Initially: x:=1; y:=2

- T1: x := x+1; y := y+1
- T2: z:= 1 / (2-x);

Otherwise...
Opacity: example

Initially: \( x := 1; y := 2 \)

After \( T_1 \): \( x := 2; y := 3 \)

- \( T_1: x := x + 1; y := y + 1 \)
- \( T_2: z := 1 / (2 - x) \)

Otherwise...
Opacity: example

Initially: \( x := 1; \ y := 2 \)

After T1: \( x := 2; \ y := 3 \)

- T1: \( x := x + 1; \ y := y + 1 \)
- T2: \( z := 1 / (2 - 2) \);

Otherwise... *divide by zero!*
Opacity
[ GK08 ]

- Intuitive definition:
  - every operation sees a consistent state
    (even if the transaction ends up aborting)
Opacity
[GK08]

- Intuitive definition:
  - every operation sees a consistent state
    (even if the transaction ends up aborting)

- Following history is serializable but violates opacity!
Does classic optimistic concurrency control guarantee opacity?

- Writes are buffered to private workspace and applied atomically at commit time.
- Reads are optimistic and the transaction is validated at commit time.
- Opacity is not guaranteed!
Theoretical Foundations

- **Safety:**
  - What schedules are acceptable by an STM?
  - Is classic atomicity property appropriate?

- **Liveness:**
  - What progress guarantees can we expect from an STM?
Progress

- STMs can abort transactions or block operations...
- But we want to avoid implementations that abort all transactions!
- We want operations to return and transactions to commit!
Requirements

- **Correct transactions:**
  - `commit` is invoked after a finite number of operations
  - either `commit` or perform an infinite number of (low-level) steps

- **Well-formed histories:**
  - every transaction that aborts is immediately repeated until it commits
Conditional progress: obstruction freedom

- A correct transaction that eventually does not encounter contention eventually commits

- ...but what to do upon contention?
Contention-managers

- Abort is unavoidable
- But want to maximize the number of commits
- Obstruction freedom property: progress and correctness are addressed by different modules.

Contention-managers encapsulate policies for dealing with contention scenarios.
Contention-managers

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.
CM: Aggressive

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- Aggressive contention manager:
  - always aborts TA
CM: Backoff

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Backoff contention manager:**
  - TB waits an exponential backoff time
  - If conflict persists, abort TA
CM: Karma

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Karma contention manager:**
  - Assign priority to TA and TB
    - Priority proportional to work already performed
  - Let $Ba$ be how many times TB has been aborted
  - Abort TA if $Ba > (TA-TB)$
CM: Greedy

Let TA be executing and TB a new transaction that arrives and creates a conflict with TA.

- **Greedy contention manager:**
  - Assign priority to TA and TB based on start time
  - If TB < TA and TA not blocked then wait
  - Otherwise abort TA
(Non-Distributed) STMs

- Basic Concepts
- Example Algorithms
  - DSTM
  - JVSTM
(Non-Distributed) STMs

- Basic Concepts
- Example Algorithms
  - DSTM
  - JVSTM
DSTM

- Software transactional memory for dynamic-sized data structures.


- Prior designs: static transactions.

- DSTM: dynamic creation of transactional objects.
DSTM

- Killer write:
  - Ownership.

- Careful read:
  - Validation.
DSTM - Writes

- To write $o$, $T$ requires a write-lock on $o$.
- $T$ aborts $T'$ if some $T'$ acquired a write-lock on $o$:
  - Locks implemented via Compare & Swap.
- Contention manager can be used to reduce aborts.
DSTM – Reads and Validation

- Concurrent reads do not conflict.

- To read o, T checks if all objects read remain valid;
  - else abort T.

- Before committing, T checks if all objects read remain valid and releases all its locks.
  - Make sure that the transaction observes a consistent state.
  - If the validation fails, transaction is restarted.
DSTM - Why is careful read needed?

- No lock is acquired upon a read:
  - invisible reads
  - visible read invalidate cache lines
  - bad performance with read-dominate workloads due to high bus contention

- What if we validated only at commit time?

  Serializability? Opacity?

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
DSTM - Why is careful read needed?

- No lock is acquired upon a read:
  - invisible reads
  - visible read invalidate cache lines
  - bad performance with read-dominate workloads due to high bus contention

- What if we validated only at commit time?

  Serializability? Y  Opacity? N
(Non-Distributed) STMs

- Basic Concepts
- Example Algorithms
  - DSTM
  - JVSTM
JVSTM

- Java Versioned Software Transactional Memory.
- Cachopo and Rito-Silva. 2006.
- Versioned boxes as the basis for memory transactions.
Optimized for read-only transactions:
- Never aborted or blocked;
- No overhead associated with readset tracking.

How?
- Multi-version concurrency control.
- Local writes (no locking, optimistic approach)
- Commit phase in global mutual exclusion.
  - Recently introduced a parallel commit version [FC09].
- Global version number (GVN)
JVSTM - Versioned boxes

- Versioned boxes
- Each transactional location uses a versioned box to hold the history of values for that location.

<table>
<thead>
<tr>
<th></th>
<th>previous:</th>
<th>value:</th>
<th>version:</th>
</tr>
</thead>
<tbody>
<tr>
<td>body:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>body:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>body:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>previous:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value:</td>
<td>2</td>
<td></td>
<td>87</td>
</tr>
<tr>
<td>version:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>previous:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>value:</td>
<td>1</td>
<td></td>
<td>23</td>
</tr>
<tr>
<td>version:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>previous:</td>
<td>null</td>
<td></td>
<td></td>
</tr>
<tr>
<td>value:</td>
<td>0</td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>version:</td>
<td>null</td>
<td></td>
<td>5</td>
</tr>
</tbody>
</table>
JVSTM - Algorithm

- Upon begin **T**, read GVN and assigned it to **T** snapshot ID (sID).

- Upon read on object **o**:
  - If **o** is in **T**’s writeset, return last value written,
  - else return the version of the data item whose sID is “the largest sID to be smaller than the **T**’ sID”.
  - If **T** is not read-only, add **o** to readset.
JVSTM - Algorithm

- Upon write, just add to the writeset.
  - No early conflict detection.

- Upon commit:
  - Validate readset:
    - Abort if any object read has changed.
  - Acquire new sID (atomic increase of GVN).
  - Apply writeset: add new version in each written VBox.
JVSTM - Execution
Contents

- Part I: (Non-Distributed) STMs
- **Part II: Distributed STMs**
- Part III: Case-studies
- Part IV: Conclusions
Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox
Origins

- Convergence of two main areas:
  - Distributed Shared Memory
  - Database Replication
Distributed Shared Memory

- DSM aims at providing a single system image
  - Fault-tolerance via checkpointing

- Strong consistency performs poorly
  - Myriad of weak-consistency models
  - Programming more complex

- Explicit synchronization
  - Locks, barriers, etc
DSTMs vs DSM

- DSTMs are simpler to program
- Transactions introduce boundaries where synchronization is required
- By avoiding to keep memory consistency at every (page) access or at the level of fine-grain locks, it may be possible to achieve more efficient implementations

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Database Replication

- Databases use transactions
  - Constrained programming model
  - Durability is typically a must
- Database replication was considered too slow
- In the last 10 years new database replication schemes have emerged
  - Based on atomic broadcast and on a single coordination phase at the beginning/end of the transaction.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
DSTMs vs DBMS

- Transactions are often much shorter in the STM world
  - This makes coordination comparatively more costly

- Durability is often not an issue
  - This makes coordination comparatively more costly

- Database replication techniques can be used as a source of inspiration to build fault-tolerant DSTMs
Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox
Goals

- Better performance:
  - Doing reads in parallel on different nodes.
  - Computing writes in parallel on different items.

- Fault-tolerance:
  - Replication the memory state so that it survives the failure of a subset of nodes.
Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Potential Problems
- Toolbox
Distribution Strategies

- Single System Image
  - Distribution is hidden
  - Easier when full replication is implemented
  - No control of the data locality

- Partitioned Global Address Space
  - Different nodes have different data
  - Distribution is visible to the programmer
  - Programmer has fine control of data locality
  - Complex programming model
Distribution Strategies

- Partitioned non-replicated
  - Max capacity
  - No fault-tolerance
  - No load balancing for reads on multiple nodes

- Full replication
  - No extra capacity
  - Max fault-tolerance
  - Max potential load balancing for reads
Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox
Dataflow Model

- Transactions are immobile and objects move through the network.
- Write: processor locates the object and acquires ownership.
- Read: processor locates the object and acquires a read-only copy.
- Avoids distributed coordination.
- Locating objects can be very expensive.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Control Flow Model

- Data is statically assigned to a home node and does not change over time.

- Manipulating objects:
  - In the node (via RPC);
  - First data is copied from the node then the are changes written back.

- Relies on fast data location mechanism.

- Static data placement may lead to poor data locality.
Distributed STMs

- Origins
- Goals
- Distribution Strategies
- Programming Models
- Toolbox
Toolbox

- Atomic Commitment
- Uniform Reliable Broadcast (URB)
- Atomic Broadcast (AB)
- Replication Strategies
Atomic Commitment

- Atomicity: all nodes either commit or abort the entire transaction.

- Set of nodes, each node has input:
  - CanCommit
  - MustAbort

- All nodes output same value
  - Commit
  - Abort

- Commit is only output if all nodes CanCommit
2-phase commit

- **Coordinator**
  - `prepare msg`
  - `vote msg (Yes or No)`
  - `decision msg (Commit or Abort)`

- **Participant 1**
  - `validate/acquire locks`
  - `apply decision`

- **Participant 2**
  - `validate/acquire locks`
  - `apply decision`

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
2PC is blocking

prepare msg

vote msg (Yes or No)

decision msg (Commit or Abort)

validate/ acquire locks

validate/ acquire locks

? ?

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
3PC

- **Coordinator**
  - Prepare msg
  - Pre-decision msg (Pre-Commit)
  - Decision msg (Commit)

- **Participant**
  - Vote msg (Yes)
  - Validate/ acquire locks
  - Log pre-commit
  - Apply decision

- **Participant**
  - Validate/ acquire locks
  - Log pre-commit
  - Apply decision

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Toolbox

- Atomic Commitment
- **Uniform Reliable Broadcast (URB)**
- Atomic Broadcast (AB)
- Replication Strategies
Uniform Reliable Broadcast

- Allows to broadcast a message $m$ to all replicas
- If a node delivers $m$, every correct node will deliver $m$
- Useful to propagate updates
Toolbox

- Atomic Commitment
- Uniform Reliable Broadcast (URB)
- Atomic Broadcast (AB)
- Replication Strategies
Atomic Broadcast

- Reliable broadcast with total order
- If replica R1 receives $m_1$ before $m_2$, any other correct replica Ri also receives $m_1$ before $m_2$
- Can be used to allow different nodes to obtain locks in the same order.
**Sequencer-based ABcast**

R1: sequencer

Assigns SN

Receive Msg + order

Receive Msg + order

Commit order

Commit order

Commit order

final uniform order

R2

Sends message

R3

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Abcast with optimistic delivery

- Total order with optimistic delivery.
- Unless the sequencer node crashes, final uniform total order is the same as regular total order.
- Application may start certificating the transaction locally based on optimistic total order delivery.
ABcast with optimistic delivery

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
ABcast with optimistic delivery

R1: sequencer
Assigns SN
Sends message
R2
Spontaneous order
Sends message
R3
Spontaneous order
Sends message
Commit order

Final delivery

Spontaneous order delivery

Commit order

Commit order

Commit order

final uniform order
Toolbox

- Atomic Commitment
- Uniform Reliable Broadcast (URB)
- Atomic Broadcast (AB)
- Replication Strategies
In absence of replication, there’s no chance to fall into deadlocks with a single lock... what if we add replication?
Replicating a single lock

T1
lock()
Update R1
Waiting for R2

T2
lock()

T1
lock()

T2
lock()
Update R2
Waiting for R1

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Replicating a single lock

T1
lock()
Update R1
unlock()

T2
lock()
Update R1

T1
lock()
Update R2
unlock()

T2
lock()
Update R1
Coordination is slow

Drawback of previous approach:
- Coordination among replicas needs to be executed at every lock operation.
- Atomic broadcast is an expensive primitive.
- The system becomes too slow.

Solution:
- Limit the coordination among replicas to a single phase, at the beginning of the transaction or commit time.
Single-phase schemes

- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting
State-machine replication

- All replicas execute the same set of transactions, in the same order.
- Transactions are shipped to all replicas using atomic broadcast.
- Replicas receive transactions in the same order.
- Replicas execute transaction by that order.
  - Transactions need to be deterministic!
State-machine replication

AB of T1’s input params

R1

T1 pre-acquires its locks

T1 execs

T1 commits

T2 is blocked due to T1

T2 execs

T2 commits

AB of T2’s input params

R2

T1 pre-acquires its locks

T1 execs

T1 commits

T2 is blocked due to T1

T2 execs

T2 commits

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Single-phase schemes

- State machine replication
- **Single master (primary-backup)**
- Multiple master (certification)
  - Non-voting
  - Voting

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Primary-backup

- Write transactions are executed entirely in a single replica (the primary)
- If the transaction aborts, no coordination is required.
- If the transaction is ready to commit, coordination is required to update all the other replicas (backups).
  - Reliable broadcast primitive.
- Read transactions may be executed on backup replicas.
  - Works fine for workloads with very few update transactions.
  - Otherwise the primary becomes a bottleneck.
Primary-backup

- **Synchronous updates:**
  - Updates are propagated during the commit phase:
    - Data is replicated immediately
    - Read transactions observe up to date data in backup replicas
    - Commit must wait for reliable broadcast to finish

- **Asynchronous updates:**
  - The propagation of updates happens in the background:
    - Multiple updates may be batched
    - Commit is faster
    - There is a window where a single failure may cause data to be lost
    - Read transactions may read stale data

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Single-phase schemes

- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting
Multi-master

- A transaction is executed entirely in a single replica.
- Different transactions may be executed on different replicas.
- If the transaction aborts, no coordination is required.
- If the transaction is ready to commit, coordination is required:
  - To ensure serializability
  - To propagate the updates
Multi-master

- Two transactions may update concurrently the same data in different replicas.
- Coordination must detect this situation and abort at least one of the transactions.
- Two main alternatives:
  - Non-voting algorithm
  - Voting algorithm
Single-phase schemes

- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting
Non-voting

- The transaction executes locally.
- When the transaction is ready to commit, the read and write set are sent to all replicas using atomic broadcast.
- Transactions are certified in total order.
- A transaction may commit if its read set is still valid (i.e., no other transaction has updated the read set).
Non-voting

Execution Transaction T1

Execution Transaction T2

AB of T1’s read & writeset

Validation & Commit T1

Validation & Commit T1

Validation & Abort T2

Validation & Abort T2

AB of T2’s read & writeset

R1

R2

R3

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Single-phase schemes

- State machine replication
- Single master (primary-backup)
- Multiple master (certification)
  - Non-voting
  - Voting
Voting

- The transaction executes locally at replica R
- When the transaction is ready to commit, only the write set is sent to all replicas using atomic broadcast
- Transactions’ commit requests are processed in total order
- A transaction may commit if its read set is still valid (i.e., no other transaction has updated the read set):
  - Only R can certify the transaction!
- R send the outcome of the transaction to all replicas:
  - Reliable broadcast
Voting

Transaction T1

R1

T1’s AB (write set)

wait for R1’s vote

R2

T1’s RB (vote)

commit

commit

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Contents

- Part I: (Non-Distributed) STMs
- Part II: Distributed STMs
- Part III: Case-studies
- Part IV: Conclusions
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D²STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)

- Partitioned (Replicated)
  - Static Transactions (Sinfonia)

- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D²STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)
Cluster-STM

- Software Transactional Memory for Large Scale Clusters
- Bocchino, Adve, and Chamberlain. 2008
- Partitioned (word-based) address space
- No persistency, no replication, no caching
- Supports only single thread per node
- Various lock acquisition schemes + 2PC
Cluster-STM

- Various methods for dealing with partitioned space
- Data movement (Dataflow model):
  - `stm get(src proc, dest, work proc, src, size, open)`
  - `stm put(src proc, work proc, dest, src, size, open)`
- Remote execution (Control flow model):
  - `stm on(src proc, work proc, function, arg buf, arg buf size, result buf, result buf size)`
Cluster-STM

```cpp
increment(proc_t proc, int *addr) {
    atomic {
        on(proc) {
            +++*addr
        }
    }
}
```
Cluster-STM

```c
increment(proc_t proc, int* addr) {
    stm_start(MY_ID)
    stm_on(MY_ID, proc, increment_local, addr, sizeof(int*), 0, 0)
    stm_commit(MY_ID)
}
```
Cluster-STM

```c
increment_local (proc_t src_proc,
    void* arg,
    size_t arg_size,
    void *result,
    size_t result_size) {
    int *addr = *((int*) arg);
    int tmp;
    stm_open_read (src_proc, addr, sizeof(int))
    stm_read (src_proc, &tmp, addr, sizeof(int))
    ++tmp;
    stm_open_write (src_proc, addr, sizeof(int))
    stm_write (src_proc, addr, &tmp, sizeof(int))
}```
Cluster-STM

- Read locks (RL) vs. read validation (RV)
  - RL:
    - immediately acquire a lock as a read (local or remote) is issued
    - abort upon contention (avoid deadlock)
    - as coordinator ends transaction, it can be committed w/o 2PC
  - Note: distributed model w/o caching:
    - each access to non local data implies remote access:
    - eager locking is for free
    - with caching only RV could be employable
Cluster-STM

- Read locks (RL) vs. read validation (RV)
- RV:
  - commit time validation (not opaque)
  - validity check requires 2PC
Cluster-STM

- Write buffering schemes

- UL undo log:
  - write is applied and an undo log is maintained
  - forced sync upon each write

- WB write buffering:
  - writes applied in local buffer
  - avoid communications for writes during exec phase
  - requires additional communication at commit time
Cluster-STM

- Write buffering: two lock acquisition schemes
  - LA: Late acquire
    - at commit time.
    - may allow for more concurrency
  - EA: Early acquire
    - as the write is issued
    - may avoid wasted work by doomed transactions
Cluster-STM: Graph Analysis (SSCA2)

Figure 5. Runtimes for SSCA2 kernel 4 implemented with locks. (b) Runtimes for SSCA2 kernel 4 implemented with Cluster-STM.
Cluster-STM: Graph Analysis (SSCA2)

Good scaling after $p = 4$

STM overhead about 2.5x
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)

- Partitioned (Replicated)
  - Static Transactions (Sinfonia)

- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D²STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)
Sinfonia

- Sinfonia: A new paradigm for building scalable distributed systems.


- Partitioned global (linear) address space

- Optimized for **static** transactions
Mini-transactions:
- A-priori knowledge on the data to be accessed

Two types of nodes:
- Application nodes
- Memory nodes

Fault-tolerance via:
- In-memory replication
- Sync (log) + async checkpoint for persistency on memory nodes
Sinfonia

In a nutshell, Sinfonia is a service that allows hosts to share application data in a fault-tolerant, scalable, and consistent manner. We propose a new paradigm for building scalable distributed systems that enables efficient and consistent access to data, while hiding the complexities that arise from concurrency and failures.

Keywords
Algorithms, Design, Experimentation, Performance, Reliability

1. INTRODUCTION

Distributed transactions, two-phase commit protocols—a major complication in existing distributed systems.

C.2.4 [Categories and Subject Descriptors]: Distributed systems, scalability, fault tolerance, shared memory,

Our implementations perform well and scale to hundreds of nodes. In a few months: a cluster file system and a group communication service. Our implementations perform well and scale to hundreds of nodes. In a few months: a cluster file system and a group communication service.

At the core of Sinfonia is a novel minitransaction primitive that enables efficient and consistent access to data, while hiding the non-trivial to develop.

Instead, developers just design and manipulate data structures within our service called Sinfonia. Sinfonia keeps data for applications on a set of memory nodes, each exporting a linear address space. At the core of Sinfonia is a lightweight minitransaction primitive that enables efficient and consistent access to data, while hiding the non-trivial to develop.

Figure 1: Sinfonia allows application nodes to share data in a fault-tolerant, scalable, and consistent manner.

We propose a new paradigm for building scalable distributed systems. Our approach targets particularly data center applications. Instead of developing complex protocols for handling distributed state, Sinfonia allows application nodes to share data in a fault-tolerant, scalable, and consistent manner. Figure 1 illustrates this paradigm.

Our approach transforms the problem of protocol design into the much easier problem of data structure design. Our approach targets particularly data center applications. Instead of developing complex protocols for handling distributed state, Sinfonia allows application nodes to share data in a fault-tolerant, scalable, and consistent manner. Figure 1 illustrates this paradigm.

Sinfonia seeks to provide a balance between functionality and efficiency is vital. This is because database systems provide more functionality than needed, resulting in performance overheads. For infrastructure applications, lock managers, and group communication services. These systems lack the performance needed for infrastructure applications, where performance overheads.

To prevent Sinfonia from becoming a bottleneck, Sinfonia itself is designed to be scalable and fault-tolerant. Sinfonia can handle data in Sinfonia relatively independently of each other. Towards this goal, Sinfonia provides fine-grained address spaces on which to store data, without imposing any structure, such as types, schemas, tuples, or tables, which all tend to increase coupling. Thus, application hosts do not have to deal with the non-trivial to develop.

As the number of nodes increases, Sinfonia’s space and bandwidth capacity of Sinfonia.

Section 6 describes how to implement Sinfonia as a library.

Section 7 shows that Sinfonia is easy to implement and can handle data in Sinfonia relatively independently of each other. Towards this goal, Sinfonia provides fine-grained address spaces on which to store data, without imposing any structure, such as types, schemas, tuples, or tables, which all tend to increase coupling. Thus, application hosts do not have to deal with the non-trivial to develop.

Section 8 discusses some of the DSM infrastructure applications. Section 8 discusses some of the DSM infrastructure applications. Section 8 discusses some of the DSM infrastructure applications.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
are chosen before the minitransaction starts executing. Upon execution onto the commit protocol. We designed minitransactions can read the item and adjust its vote to abort if the result is zero. returns zero (and will commit otherwise), then the coordinator can piggyback this last action onto the first phase of two-Phase commit (e.g., this is the case if this action is a data update). In Sinfonia, coordinators are application nodes and participants are informed of their data better than what Sinfonia can infer. And third, controlled caching has three clear advantages: First, there is greater flexibility on policies of what to cache and what to evict. Second, as applications cache is always current (not stale). Managing caches in through Sinfonia is always current (not stale).
### Sinfonia

#### API

```cpp
class Minitransaction {
public:
    void cmp(memid, addr, len, data); // add cmp item
    void read(memid, addr, len, buf); // add read item
    void write(memid, addr, len, data); // add write item
    int exec_and_commit();    // execute and commit
};
```

#### Example

```cpp
t = new Minitransaction;
t->cmp(memid, addr, len, data);
t->write(memid, addr, len, newdata);
status = t->exec_and_commit();
...```

---

**Figure 2:** Minitransactions have compare items, read items, and write items. Compare items are locations to compare against given values, while read items are locations to read and write items are locations to write. Methods `cmp`, `read`, and `write` populate a minitransaction without communication. The API is straightforward:

- `(const memid, addr, len, data)`
- `(memid, addr, len, buf)`
- `(memid, addr, len, data)`

Roughly speaking, a coordinator executes a transaction by asking participants to perform one or more transaction items (equality comparison), and commit. Roughly speaking, a coordinator executes a transaction by asking participants to perform one or more transaction items (equality comparison), (2) if all comparisons succeed, or if a validation fails, abort. Thus, the compare items control whether the transaction commits or aborts.

### 3.4 Caching and consistency

Sinfonia does not cache data at application nodes, but provides a minitransaction system for executing file system operations, such as`stat` (NFS's `getattr`). A minitransaction is a lightweight transaction that allows a set of operations to be executed atomically and consistently. SinfoniaFS uses this system to provide consistent data access.

In Section 4 we explain how minitransactions are executed and commit. Each minitransaction has a status that indicates whether it has executed and so the validations were successful. SinfoniaFS uses minitransactions to manage cache consistency.

A frequent minitransaction idiom is to use compare items to validate updates. For example, if the last action is a read, then the participant knows that the coordinator will abort if the read item is invalid. If the last action is a write, then the participant knows how the coordinator will make its decision to abort or commit, if the participant knows how the coordinator will make its decision to abort or commit, then the participant can also piggyback the action onto the commit protocol. For example, if the last action is a read, then the participant knows that the coordinator will abort if the read item is invalid. If the last action is a write, then the participant knows how the coordinator will make its decision to abort or commit, if the participant knows how the coordinator will make its decision to abort or commit, then the participant can also piggyback the action onto the commit protocol.

A minitransaction execution consists of two phases:

1. **Prepare**:
   - The coordinator asks all participants if they are ready to execute the transaction.
   - Participants that are ready respond with their votes (aborted or committed).

2. **Commit**:
   - If all participants vote to commit, the transaction is committed.
   - If any participant votes to abort, the transaction is aborted.

In practice, participants are assigned swap addresses using a minitransaction with one compare and one write item on the same location—a compare-and-swap operation. Methods `cmp` and `write` populate a minitransaction without communication. Methods `cmp` and `write` populate a minitransaction without communication. Methods `cmp` and `write` populate a minitransaction without communication.

#### API Example

```cpp
public:
    void write(memid, addr, len, data); // add
    void read(memid, addr, len, buf); // add
    void cmp(memid, addr, len, data); // add

Minitransaction t = new Minitransaction;
    t->cmp(memid, addr, len, data);
    t->write(memid, addr, len, data);
    status = t->exec_and_commit();
    ...
```
Sinfonia

- Global space is partitioned
  - Transaction may need to access different memory nodes
  - It can only commit if it can commit at all memory nodes
  - 2-phase commit
4.1 Basic architecture

A minitransaction in Sinfonia consists of a set of items, which can be
of four types: compare items, read items, write items, and in-doubt
items. To execute and commit a minitransaction, the coordinator
generates a unique identifier for the minitransaction and sends it to
the participants. Each participant then votes for committing or for
aborting the minitransaction.

4.2 Minitransaction protocol overview

Phase 1, the coordinator (application node) generates a new transac-
tion id (tid), and sends the minitransaction to the participants (mem-
ory nodes), so that minitransactions are blocked in-doubt (if logging is
enabled); logging occurs only if the participant votes for committing.

4.3 Minitransaction protocol details

The two-phase protocol in Figure 4. Phase 1 executes and prepares
equality against supplied data; if any test fails, the minitransaction
in-doubt. Another deadlock-avoidance scheme is to acquire locks
in some predefined order, but with that scheme, the coordinator in
participants are memory nodes that keep application data, so if they go
coordinator crashes. This is reasonable for Sinfonia because partic-
ipants are memory nodes that keep application data, so if they go
coordinator crashes. This is reasonable for Sinfonia because partic-

Sinfonia comprises a set of memory nodes and a
server process that keeps Sinfonia data and the minitransaction
of which we run the minitransaction protocol. Memory nodes run

Recall that a minitransaction has compare items, read items, and
write items, and we consider a transaction to be committed if all participants
have a yes vote in their log. Standard two-phase commit requires
the minitransaction, while phase 2 commits it. More precisely, in

Recall that Sinfonia comprises a set of memory nodes and a
server process that keeps Sinfonia data and the minitransaction
of which we run the minitransaction protocol. Memory nodes run

To execute and commit minitransaction

<table>
<thead>
<tr>
<th>participant 1</th>
<th>participant 2</th>
<th>participant 3</th>
<th>coordinator</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXEC&amp;PREPARE</td>
<td>vote</td>
<td>COMMIT</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Protocol for executing and committing minitransactions.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Sinfonia

No support for caching:
- delegated to application level
- same applies for load balancing

Replication:
- aimed at fault-tolerance, not enhancing performance
- fixed number of replicas per memory node
- primary-backup scheme ran within first phase of 2PC
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)
- Partitioned (Replicated)
  - Static Transactions (Sinfonia)
- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D²STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)
D²STM

- D²STM: Dependable Distributed STM
- Couceiro, Romano, Rodrigues, Carvalho, 2009
- Single-image system
  - Full replication
  - Strong consistency
- Certification-based replication scheme
  - Based on Atomic Broadcast
  - Built on top of JVSTM
**D²STM**

- Non-voting replication scheme
- Transactions execute in a single replica
- No communication during the execution
- Writeset and readset AB at commit time
- Deterministic certification executed in total order by all replicas
- No distributed deadlocks
D²STM

Execution
Transaction T1

AB of T1’s read & writeset

Validation&Commit T1

Validation&Commit T1

R1

R2

R3
D$^2$STM

AB of both T1’s readset & writeset

Problem:
(very) big message size
**D²STM**

- In STMs, transaction’s execution time is often 10-100 times short than in DBs:
  - the cost of AB is correspondingly amplified

- Bloom Filter Certification:
  - space-efficient encoding (via Bloom Filter) to reduce message size
**D²STM**

- Bloom filters
  - A set of $n$ items is encoded through a vector of $m$ bits
  - Each item is associated with $k$ bits through $k$ hash functions having as image $\{1..m\}$:
    - insert: set $k$ bits to 1
    - query: check if all $k$ bits set to 1
D²STM

- **False Positives:**
  - An item is wrongly identified as belonging to a given set
  - Depend on the number of bits used per item \((m/n)\) and the number of hash functions \((k)\)

- **D²STM computes the size of the Bloom filter based on:**
  - User-defined false positive rate
  - Number of items in the read set (known)
  - Number of BF queries, estimated via moving average over recently committed transactions
D²STM

- Read-only transactions:
  - local execution and commit
\textbf{D}²\textbf{STM}

\begin{itemize}
  \item Write transaction T:
    \begin{itemize}
      \item Local validation (read set)
      \item If the transaction is not locally aborted, the read set is encoded in a Bloom filter
      \item Atomic broadcast of a message containing:
        \begin{itemize}
          \item the Bloom filter encoding of tx readset
          \item the tx write set
          \item the snapshotID of the tx
        \end{itemize}
      \item Upon message delivery: validate tx using Bloom filter’s information
    \end{itemize}
\end{itemize}
D²STM

for each committed $T'$ s.t. $T'$.snapshotID > $T$.snapshotID
for each data item $d$ in the writeset of $T'$
  if $d$ is in Bloom filter associated with $T'$’s readset
    abort $T$
  
  // otherwise...

commit $T$
D²STM

- STM Bench7: Results
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)

- Partitioned (Replicated)
  - Static Transactions (Sinfonia)

- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D²STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)
AlC

- Asynchronous Lease Certification Replication of Software Transactional Memory
- Carvalho, Romano, Rodrigues, 2010
- Exploit data access locality by letting replicas dynamically establish ownership of memory regions:
  - replace AB with faster coordination primitives:
  - no need to establish serialization order among non-conflicting transactions
  - shelter transactions from remote conflicts

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Data ownership established by acquiring an Asynchronous Lease
- mutual exclusion abstraction, as in classic leases...
- ...but detached from the notion of time:
- implementable in a partially synchronous system
- Lease requests disseminated via AB to avoid distributed deadlocks.
ALC

- Transactions are locally processed
- At commit time check for leases:
  - An Asynchronous Lease may need to be established
- Proceed with local validation
ALC

- If local validation succeeds, its writeset is propagated using Uniform Reliable Broadcast (URB):
  - No ordering guarantee, 30-60% faster than AB

- If validation fails, upon re-execution the node holds the lease:
  - Transaction cannot be aborted due to a remote conflict!
ALC

Lease Request (AB)

Lease Ensured (URB)

Apply (URB)

P1

P2

P3

Certification

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
ALC

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
If applications exhibit some access locality:
- avoid, or reduce frequency of AB
- locality improved via conflict-aware load balancing

Ensure transactions are aborted at most once due to remote conflicts:
- essential to ensure liveness of long running transactions
- benefic at high contention rate even with small running transactions
ALC

Application

Distributed STM API Wrapper

JVSTM

Replication Manager

Lease Manager

Group Communication Service

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
ALC

- Synthetic “Best case” scenario
- Replicas accessing distinct memory regions

![Graph showing throughput vs. number of replicas](image)

- ALC: 3x improvement
- CERT: 10x improvement

---

M. Conceiro, P. Romano, L. Rodrigues, HPCS 2011
ALC

- Synthetic “Worst case” scenario
- All replicas accessing the same memory region

![Graphs showing throughput and abort rate vs number of replicas for ALC and CERT.](image-url)
Lee Benchmark

Graphs showing the speed-up and abort rate of ALC and CERT with varying number of replicas.
Part III: Case-Studies

- Partitioned Non-Replicated
  - STM for clusters (Cluster-STM)

- Partitioned (Replicated)
  - Static Transactions (Sinfonia)

- Replicated Non-Partitioned
  - Certification-based with Bloom Filters (D²STM)
  - Certification with Leases (ALC)
  - Active Replication with Speculation (AGGRO)
AGGRO

- AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing
- R. Palmieri, Paolo Romano and F. Quaglia, 2010

- Active Replication for STMs
  - Multiple replicas
  - All replicas execute update transactions
  - Read-only transactions can execute in any replica
  - Data survives failures of replicas
Basic Active Replication

Atomic Broadcast

Tx Exec C
With Optimistic Delivery

- Opt
- Atomic Broadcast
- Tx Exec
- Hold Locks
- C

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Improvement

Atomic Broadcast

Tx Exec

Opt

Atomic Broadcast

Tx Exec

Hold Locks

C

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
But...

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Speculative

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
AGGRO

- Transactions are started in speculative order immediately after the optimistic delivery.
- Writes kill all transactions that have read stale data.
- Items touched by speculative transactions Tspec are marked as “work-in-progress (WIP)” while Tspec executes.
  - When Tspec terminates (but not yet committed) items are unmarked as WIP.
  - Transaction only read values from terminated transactions.

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
AGGRO Algorithm

upon opt-Deliver(Ti)

start transaction Ti in a speculative fashion
AGGRO Algorithm

upon \texttt{write}(Ti, X, v)

\begin{itemize}
\item if (X not already in Ti.WS)
  \begin{itemize}
  \item add X to Ti.WS
  \item mark X as WIP // C&S
  \end{itemize}
\item for each Tj that follows Ti in OAB order:
  \begin{itemize}
  \item if (Tj read X from Tk preceding Ti) abort Tj
  \end{itemize}
\item else
  \begin{itemize}
  \item update X in Ti.WS
  \end{itemize}
\end{itemize}
AGGRO Algorithm

upon read(Ti, X)

if (X in Ti.WS) return X.value from Ti.WS

if (X in Ti.RS) return X.value from Ti.RS

wait while (X is marked WIP)

let Tj be tx preceding Ti in OAB order that wrote X

Ti.readFrom.add(Tj)
AGGRO Algorithm

upon completed (\( Ti \))

atomically {
    for each \( X \) in \( Ti.WS \): unmark \( X \) as WIP by \( Ti \)
}

upon commit(\( Ti \))

atomically {
    for each \( X \) in \( Ti.WS \): mark \( X \) as committed
}
AGGRO Algorithm

upon abort($Ti$)

   abort any transaction that read from $Ti$
   restart $Ti$

upon TO-Deliver($Ti$)

   append $Ti$ to TO-order
   wait until all xacts preceding $Ti$ in TO-order committed
   if (validation of $Ti$’s readset fails) abort ($Ti$)
   else commit($Ti$)
AGGRO Performance

- Performance speed-up
  (20% reordering, only one SO explored)

![Graph showing performance speed-up with and without speculation. The x-axis represents transactions per second, and the y-axis represents response time in microseconds. The graph shows a significant speedup with speculation compared to no speculation.](image)
Contents

- Part I: (Non-Distributed) STMs
- Part II: Distributed STMs
- Part III: Case-studies
- Part IV: Conclusions
Conclusions

- Replication helps in read-dominated workloads or when writes have low conflicts
- Replication provides fault-tolerance
- Some techniques have promising results

M. Couceiro, P. Romano, L. Rodrigues, HPCS 2011
Conclusions

- No technique outperforms the others for all workloads, networks, number of machines, etc.

- Autonomic management of the distributed consistency and replication protocols
  - Change the protocols in runtime, in face of changing workloads.
A bit of publicity

- Time for the commercials
CLOUD-TM

- DTM: a programming paradigm for the Cloud?

- Stay tuned on www.cloudtm.eu
Euro-TM Cost Action

- Research network bringing together leading European experts in the area of TMs
- Contact us if you are interested in joining it:
  - romano@inesc-id.pt
  - ler@inesc-id.pt
- www.eurotm.org
Bibliography


Bibliography


Bibliography


[FC10] Sérgio Fernandes and João Cachopo, A scalable and efficient commit algorithm for the JVSTM, 5th ACM SIGPLAN Workshop on Transactional Computing

Bibliography

- [GHP05] Rachid Guerraoui, Maurice Herlihy, Bastian Pochon: Toward a theory of transactional contention managers. PODC 2005: 258-264

- [GU09] Rachid Guerraoui, Tutorial on transactional memory, CAV 2009

Bibliography


Bibliography

- [KA]LKW08] Christos Kotselidis, Mohammad Ansari, Kim Jarvis, Mikel Luján, Chris Kirkham and Ian Watson, DiSTM: A Software Transactional Memory Framework for Clusters, In the 37th International Conference on Parallel Processing (ICPP'08), September 2008


Bibliography


The End

Thank you.