Tutorial on Distributed Transactional Memories

Paolo Romano and Luís Rodrigues

ToC

PART I

- Non-Distributed Transactional Memories
  - Concepts (45 min)
  - Systems (45 min)

(break)

PART II

- Distributed Transactional Memories
  - Concepts (45 min)
  - Systems (45 min)

Part I

Non-Distributed Transactional Memories

ToC

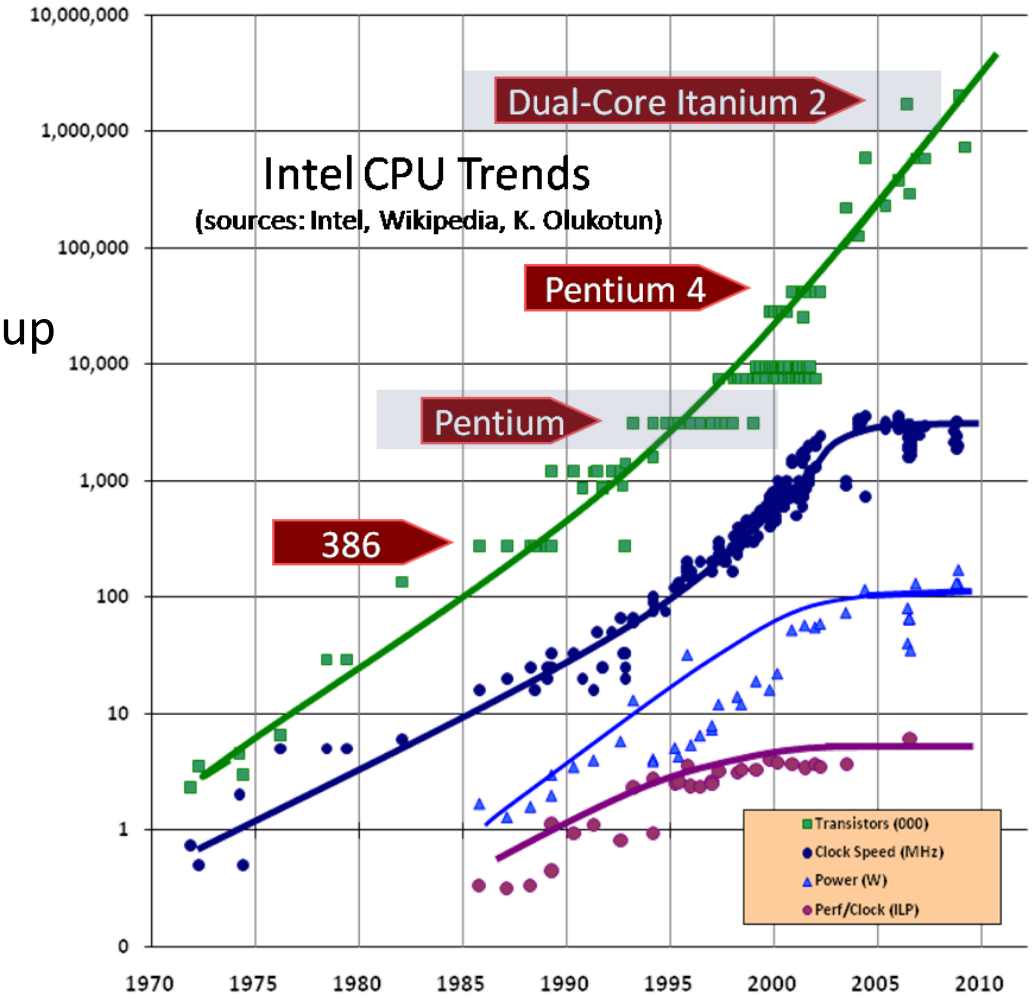
PART I

- Non-Distributed Transactional Memories
  - Concepts (45 min)
  - Systems (45 min)

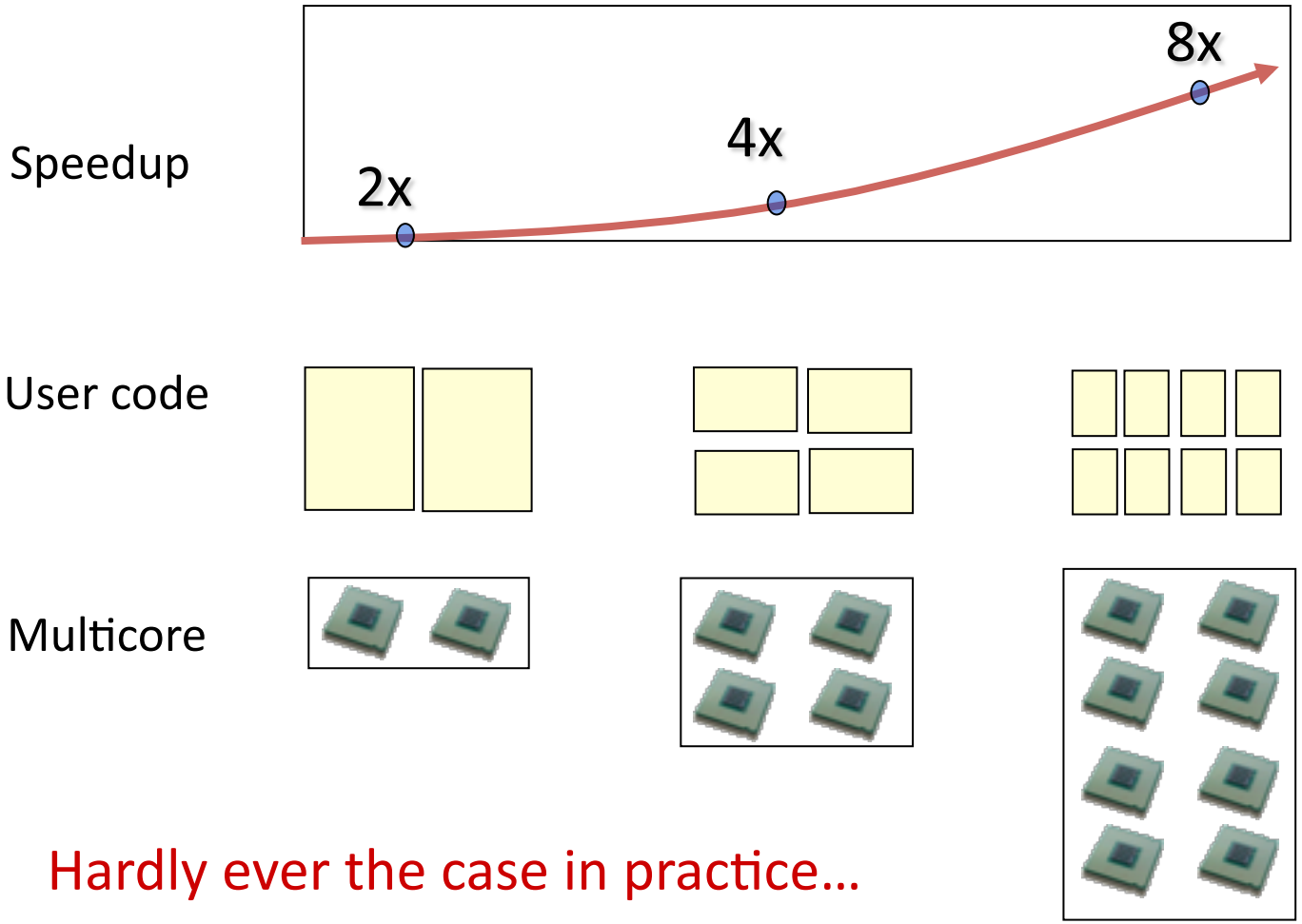
The era of free performance gains is over

- Over the last 30 years:
  - new CPU generation = free speed-up
- Since 2003:
  - CPU clock speed plateaued...
  - but Moore's law chase continues:
    - Multi-cores, Hyperthreading...

FUTURE IS PARALLEL



Multicore Software Scaling

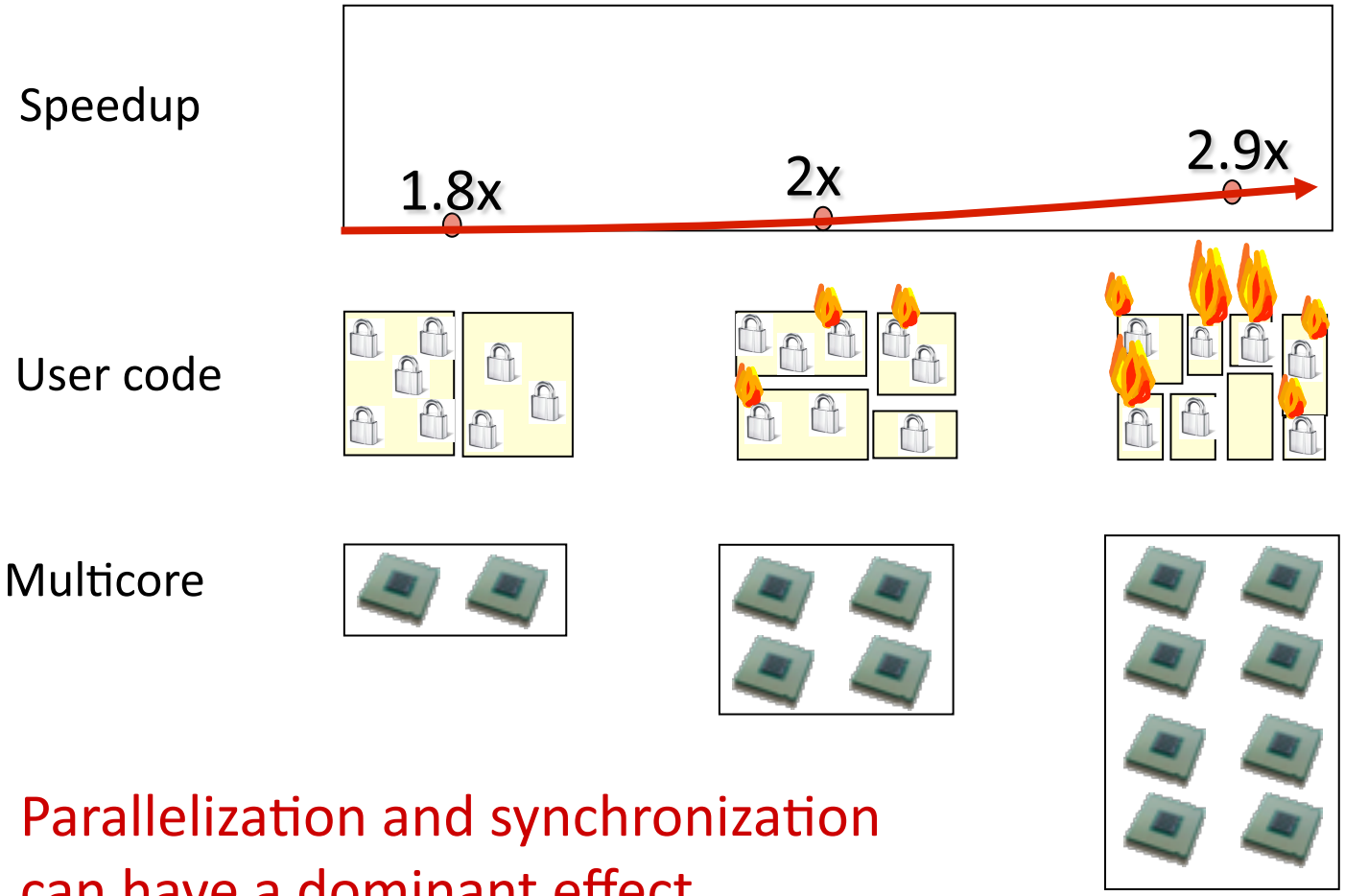


Hardly ever the case in practice…

Programs are not entirely parallel

- It is hard or impossible to structure a program in a set of parallel independent tasks
- Part of the program may need to be performed in a serial manner
- Parallel parts may need to share data
  - Access to shared data needs to be synchronized

Real-World Multicore Scaling



Parallelization and synchronization
can have a dominant effect
on performance!

Coarse grained parallelism?

simple but does not scale

Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

SequentialPart = 25%

4 cores -> speedup = 2.3!

Coarse grained parallelism?

simple but does not scale

Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

SequentialPart = 25%

8 cores -> speedup = 2.9!

Coarse grained parallelism?

simple but does not scale

Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

SequentialPart = 25%

32 cores -> speedup = 3.7!

Coarse grained parallelism?

simple but does not scale

Amdahl's Law:

Speedup = 1/(ParallelPart/N + SequentialPart)

SequentialPart = 25%

128 cores -> speedup = 3.9 ☹

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;
```

Explicit synchronization

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

  ...

  non synchronized code;

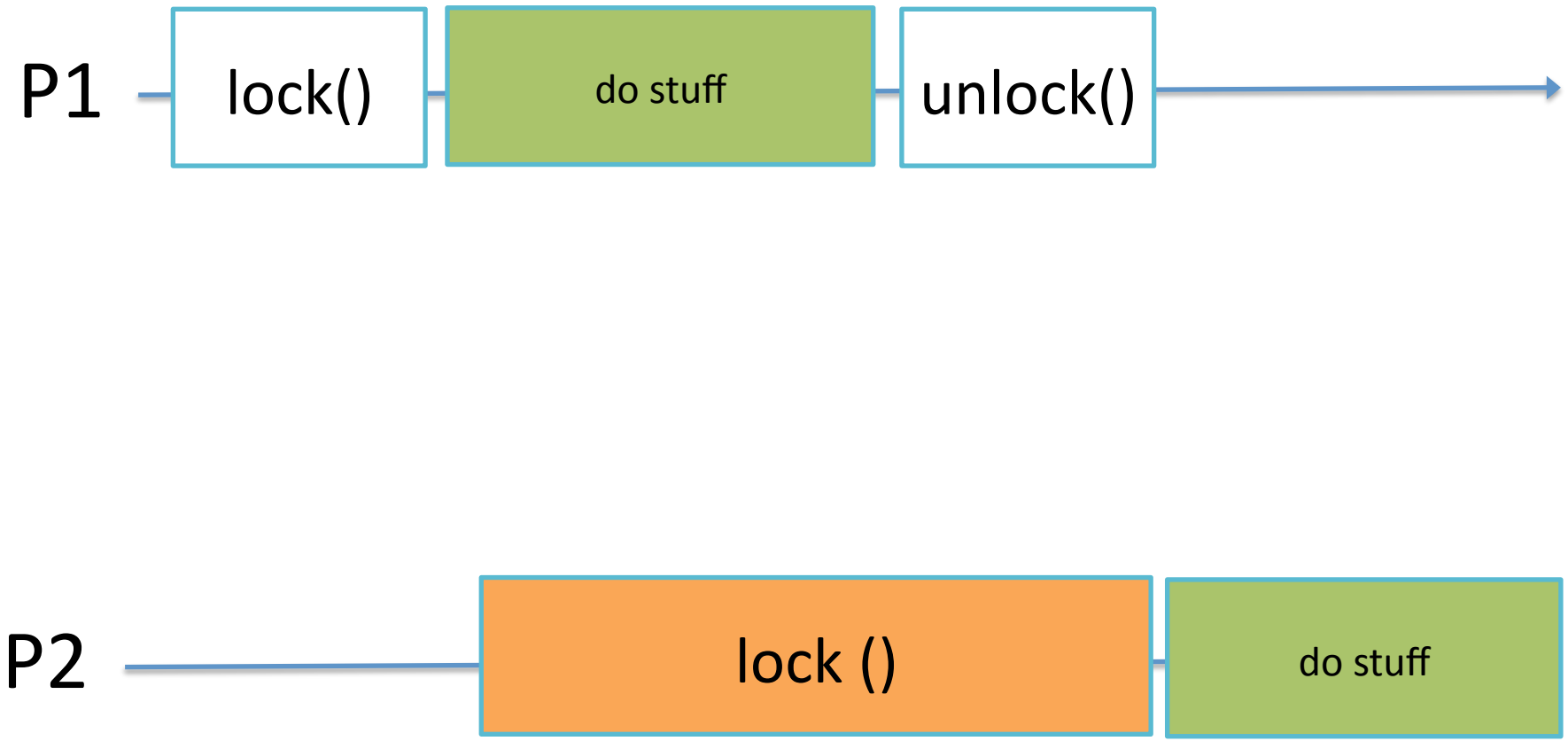
  lock (); ← **may be forced to block until released!**

  do stuff on shared data;

  unlock ();

  more non synchronized code;

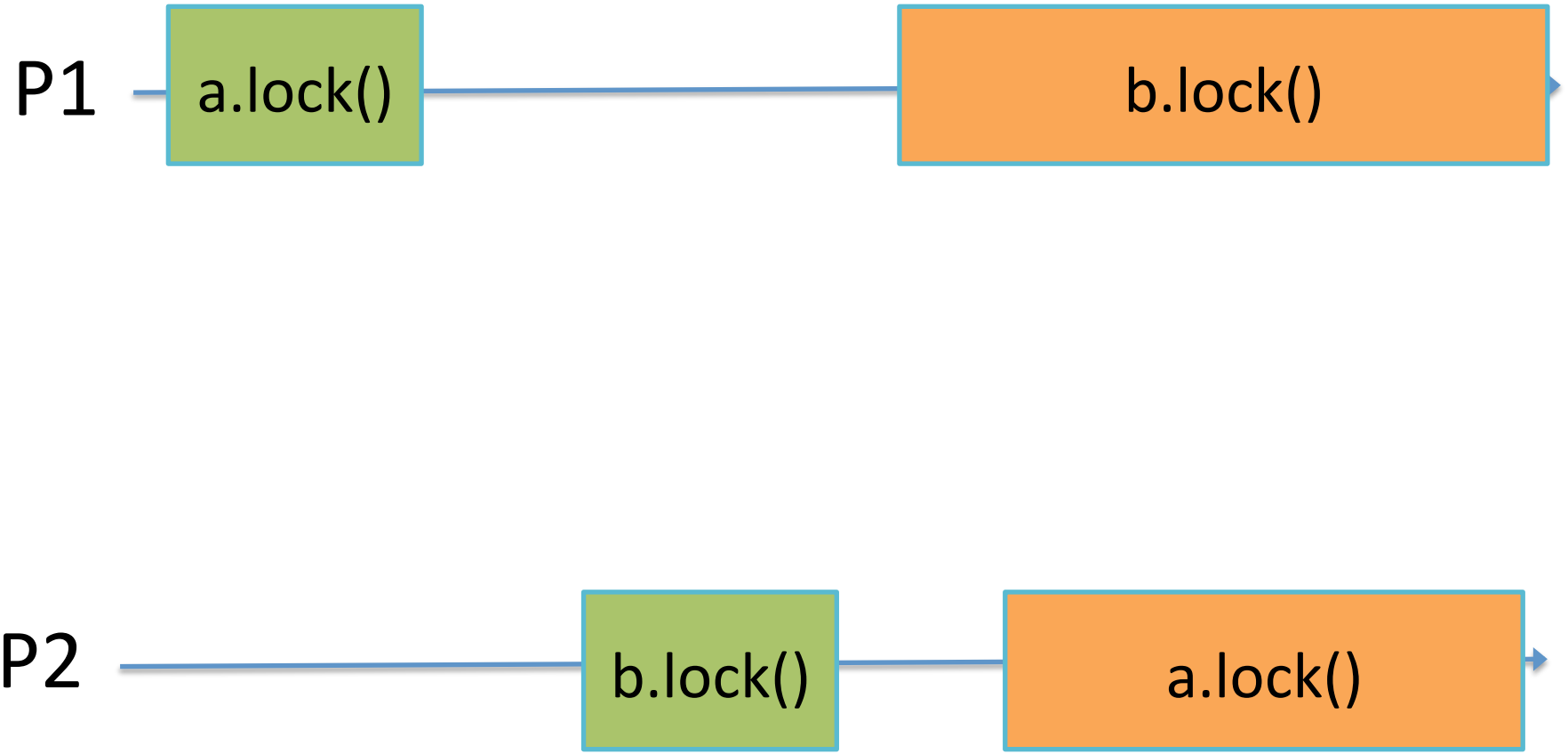
Locks in action



Locks are broken

- Deadlock: locks acquired in “wrong” order.
- Races: due to forgotten locks
- Error recovery tricky: need to restore invariants and release locks in exception handlers
- Simplicity vs scalability?

Deadlocks in action



Locks do not compose

- You cannot build a big working program from small working pieces

```
Class Account {
    Euros amount;
    public deposit(Euros d) {
                amount = amount+d;
            }
    public withdraw(Euros w) {
                if (amount>w) amount = amount–w;
            }
    public Euros read () {
                return amount;
    }
```

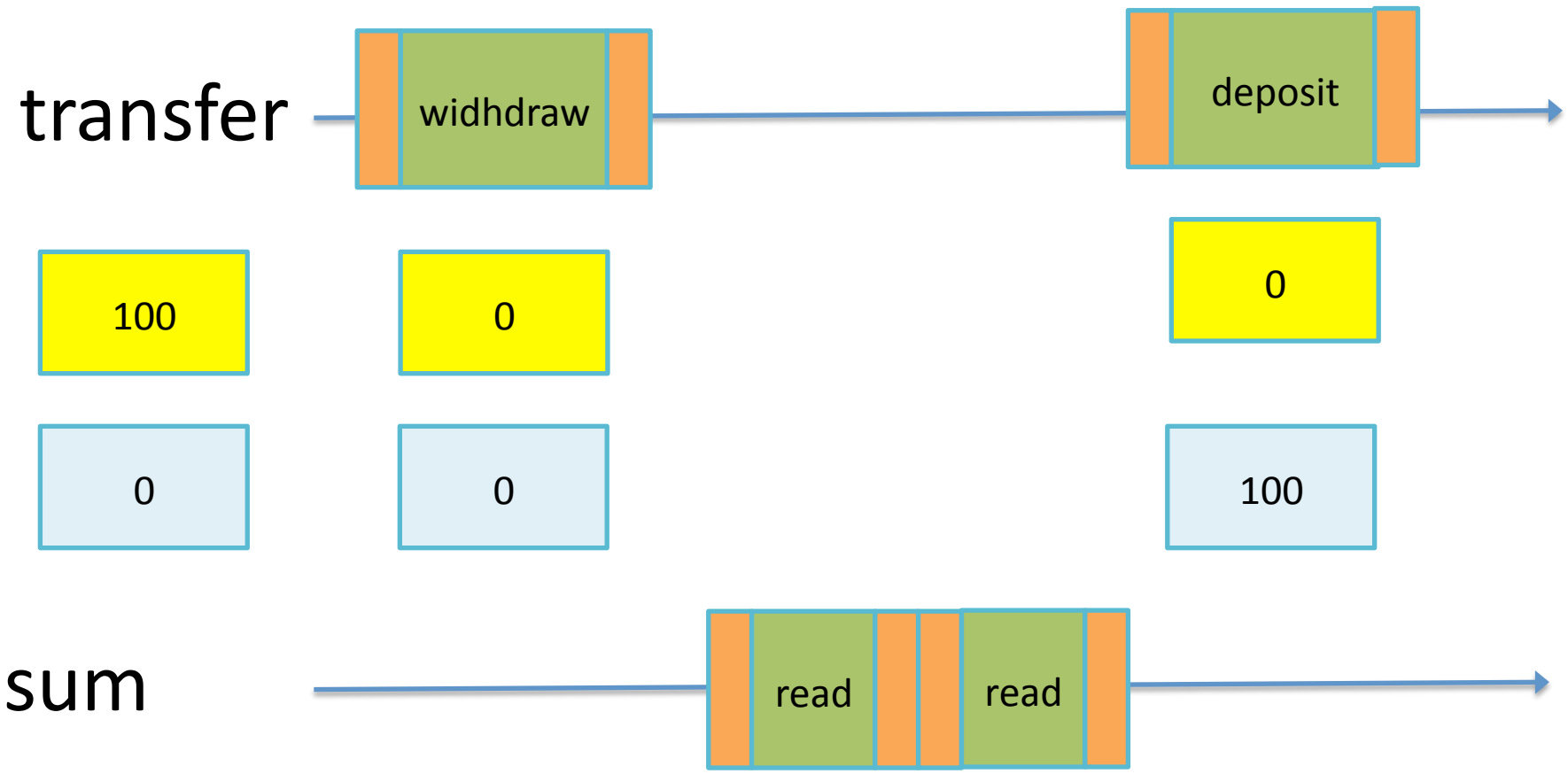
Locks do not compose

```
Class Account {
    Euros amount;
    Lock mutex;
    public deposit(Euros d) {
                mutex.lock();
                amount = amount+d;
                mutex.unlock();
    }
    public withdraw(Euros w) {
                mutex.lock();
                if (amount>w) amount = amount–w;
                mutex.unlock ();
     }
```

Locks do not compose

```
void transfer (Account from, Account to, Euros value){
        from.withdraw (value);
        to.deposit(value);
}

Euros sum (Account a1, Account a2) {
    return a1.read() + a2.read();
}
```

Locks in action



Fine grained parallelism?
easier to say than to do

- Simple grained locking is a **very complex:**
  - need to reason about deadlocks, livelocks, priority inversions:
    - complex/undocumented lock acquistion protocols
    - scarce composability of existing software modules

  ... and a **verification nightmare:**
    - subtle bugs that are extremely hard to reproduce

- Make parallel programming **accessible to the masses!**

Concurrent programming without locks?

- Lock-free algorithms.
- These algorithms exist but are very hard to design and to prove correct.
- Only for very specialized applications.
- Designed and implemented by top experts.
  - See conferences such as PODC, DISC or SPAA.

How to make

parallel programming

accessible to the masses?

Abstractions for simplifying concurrent programming…

Atomic Transactions

Atomic Transactions

access object 1;

access object 2;

Atomic Transactions

```
atomic {
    access object 1;
    access object 2;
}
```

Atomic Transactions

```
atomic {
    access object 1;
    access object 2;
}
atomic {
    access object 3;
    access object 4;
}
```

Atomic Transactions

```
atomic {
        atomic {
            access object 1;
            access object 2;
        }
        atomic {
            access object 3;
            access object 4;
        }
}
```

Transactional memories

- Key idea:
  - hide away synchronization issues from the programmer
  - replace locks with atomic transactions.
- Advantages:
  - avoid deadlocks, priority inversions, convoying
  - simpler to reason about, verify, compose

Historical Perspective

- Early work in the context of database systems (76)
- Formalization of properties (79)
- Integration in languages for distributed computing (82)
- Suggested integration in hardware (93)
- Software implementations (95)
- More flexible software implementations (03)

Historical Perspective

Today among the most relevant research topics in the areas of:

- Computer architecture
- Programming Languages
- Operating Systems
- Distributed Computing

STRONG INTERDISCIPLINARITY

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 unsatisfaction
- 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

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 (strict two phase locking)

- 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.

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

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

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?

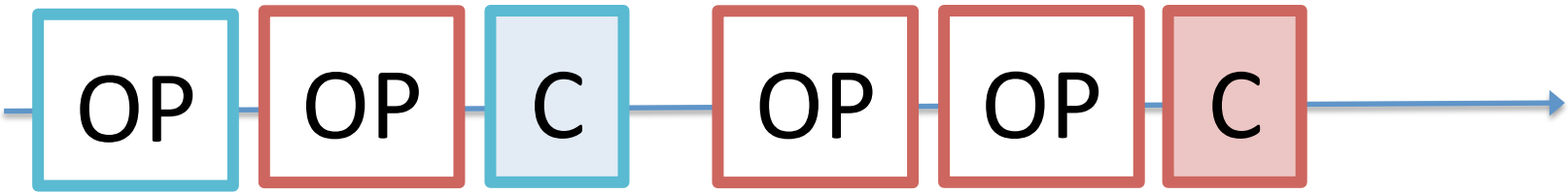
Classic atomicity property

- 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

Histories

- 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

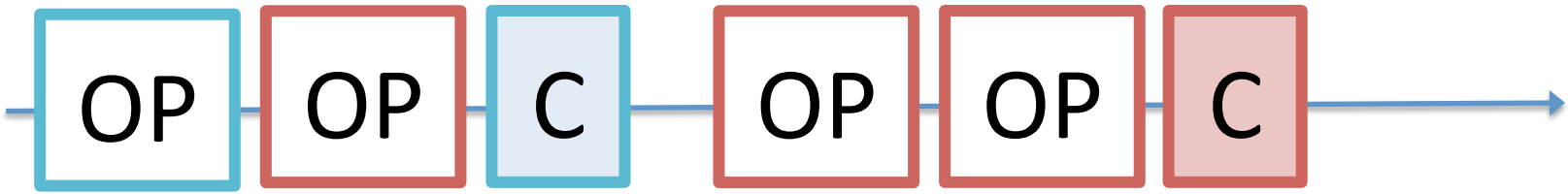


Histories

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

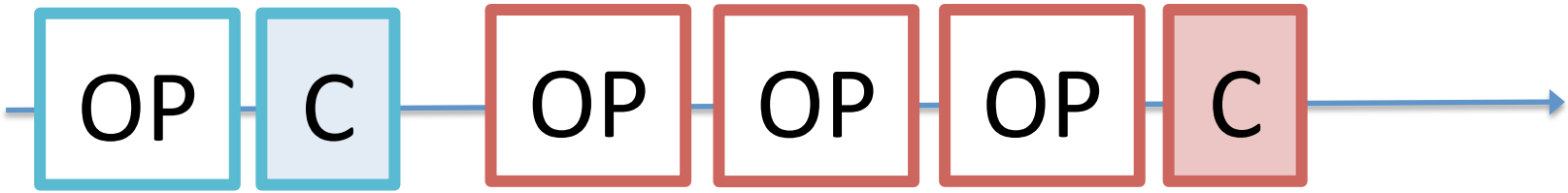
Histories

- 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:

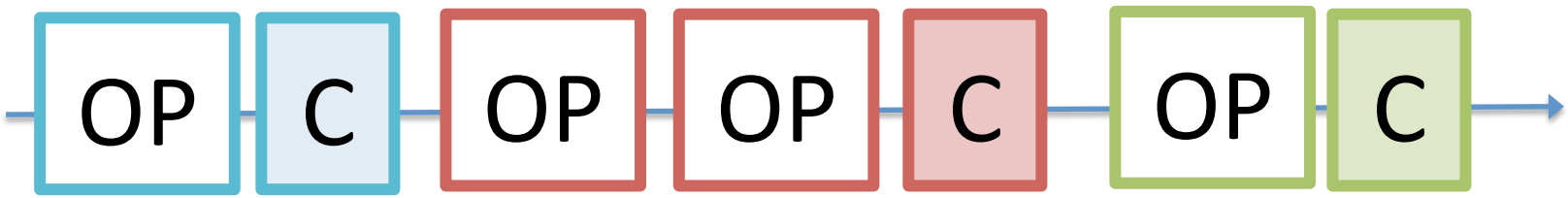


Histories

- 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:



Histories

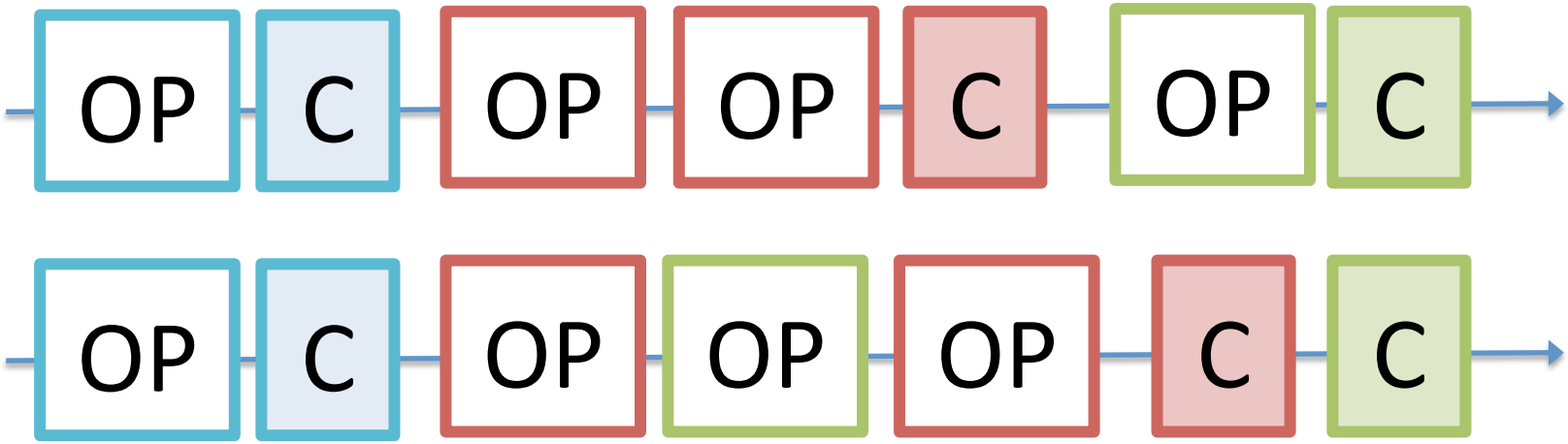
- A history is sequential if it has only sequential transactions; it is concurrent otherwise.
- Non-sequential:

Histories

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

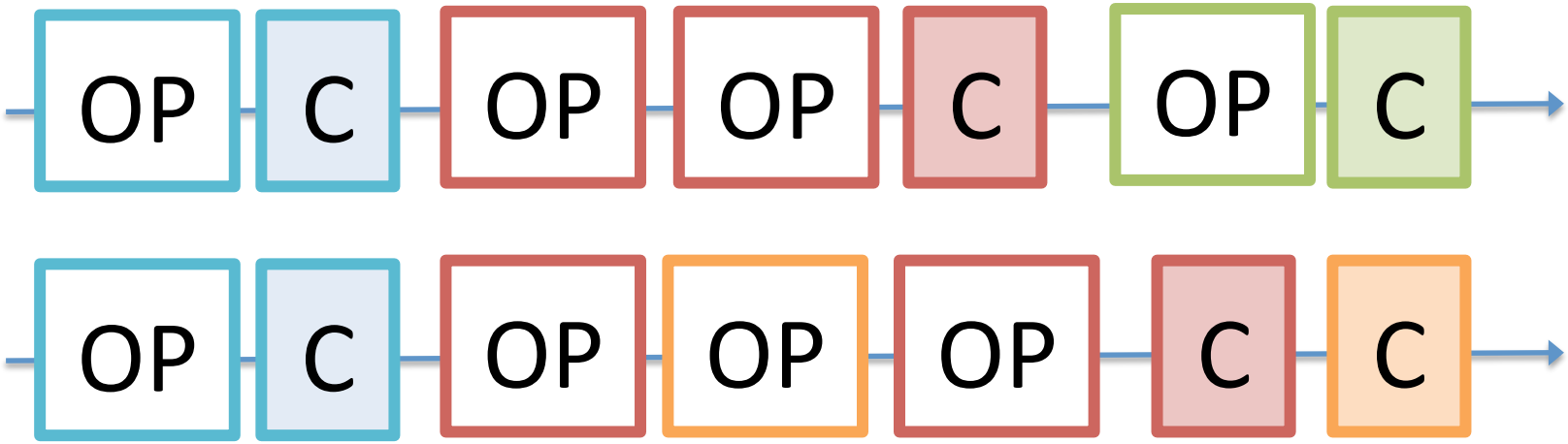
Histories

- Two histories are **equivalent** if they have the same transactions
- **Equivalent**:



Histories

- Two histories are **equivalent** if they have the same transactions
- **Non-equivalent:**



What the programmer wants?

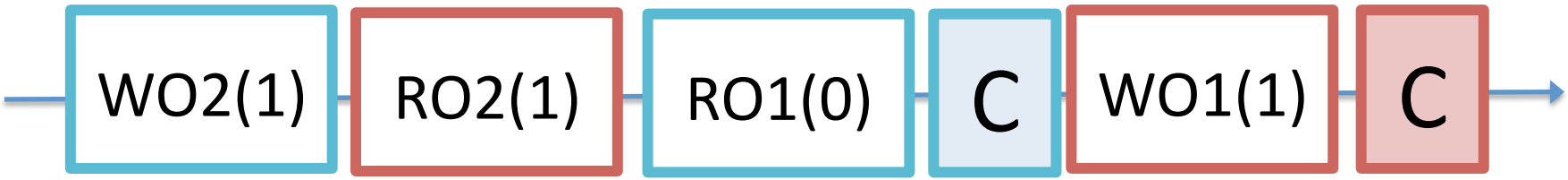
- Programmer does not want to be concerned about concurrency issues.
- Execute transactions “as if” they are serial
- No need to be “in serial” as long as results are the same

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**

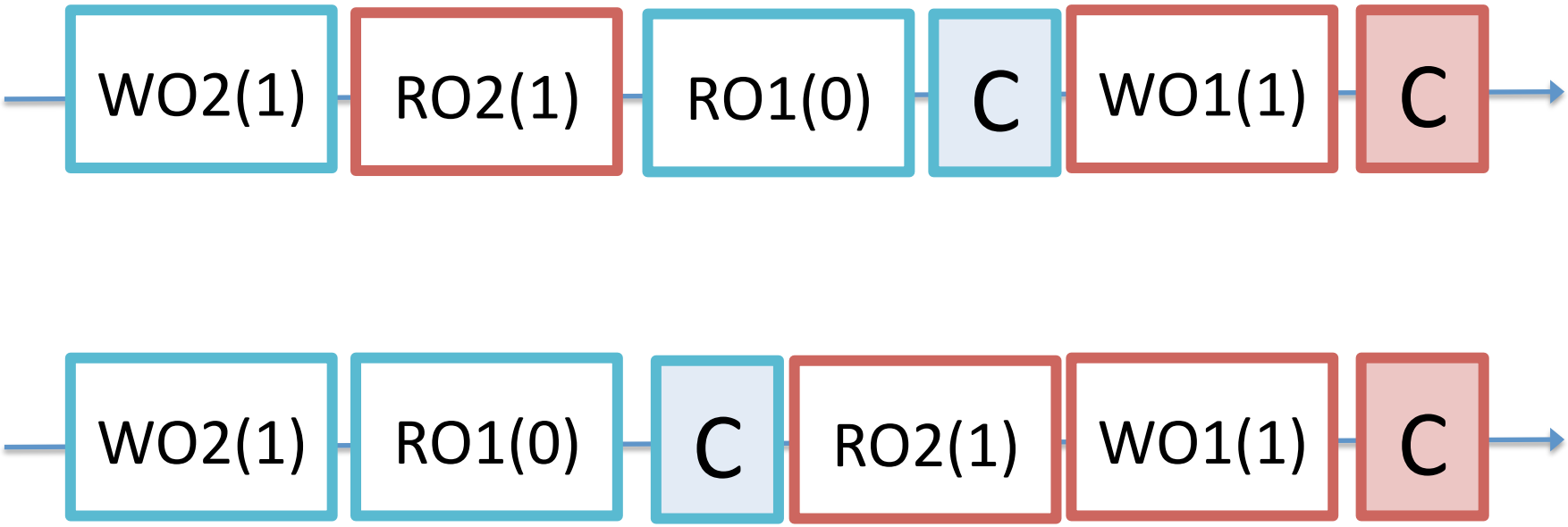
Serializability

- Serializable?



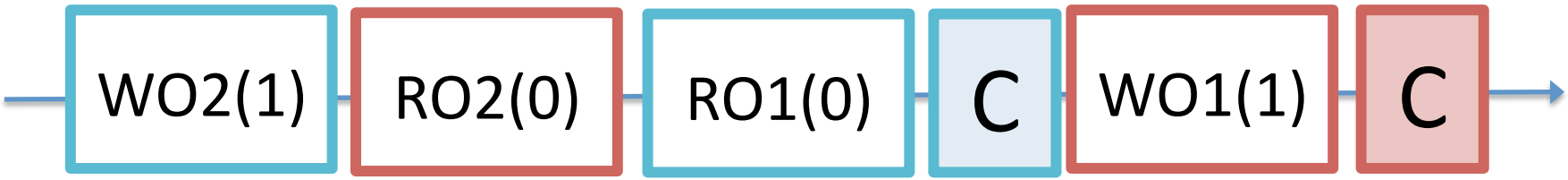
Serializability

- Serializable:



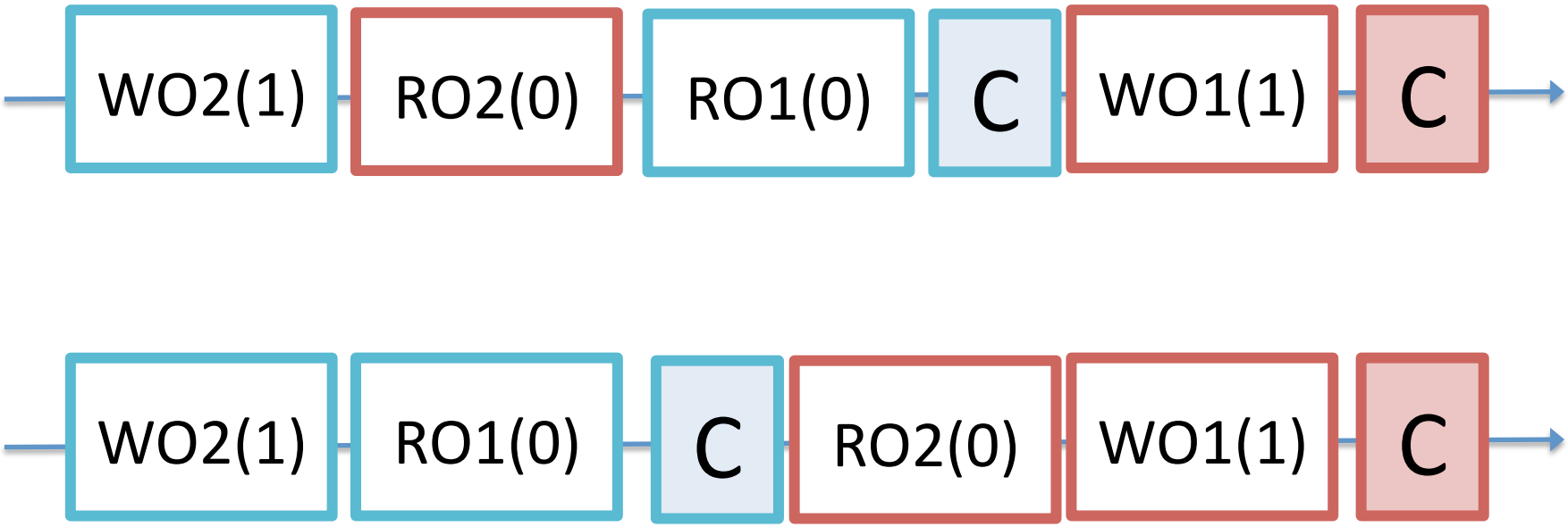
Serializability

- Non-serializable:



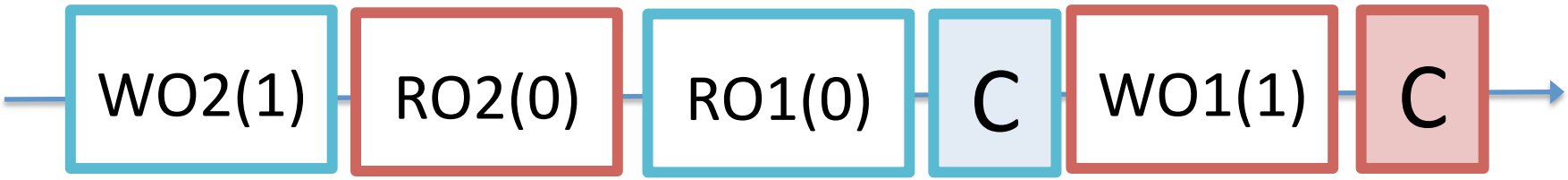
Serializability

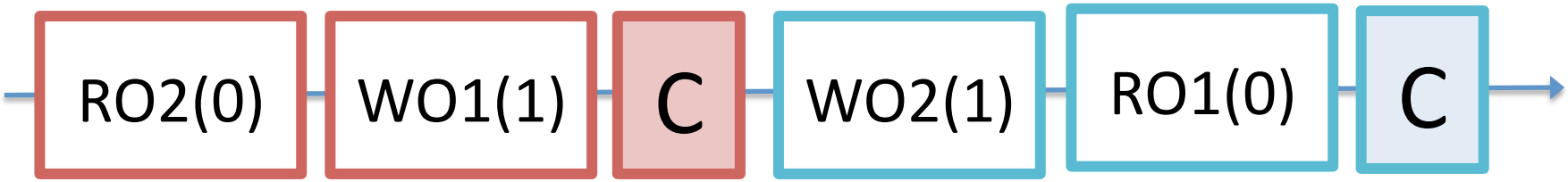
- Non-serializable (blue before red):



Serializability

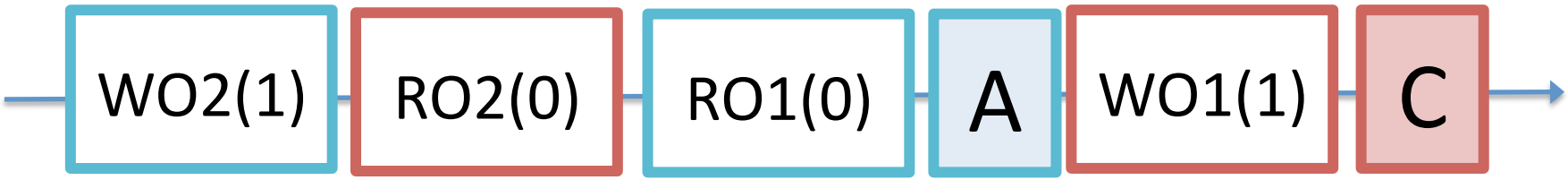
- Non-serializable (red before blue):





Serializability

- Serializable (blue aborts)?

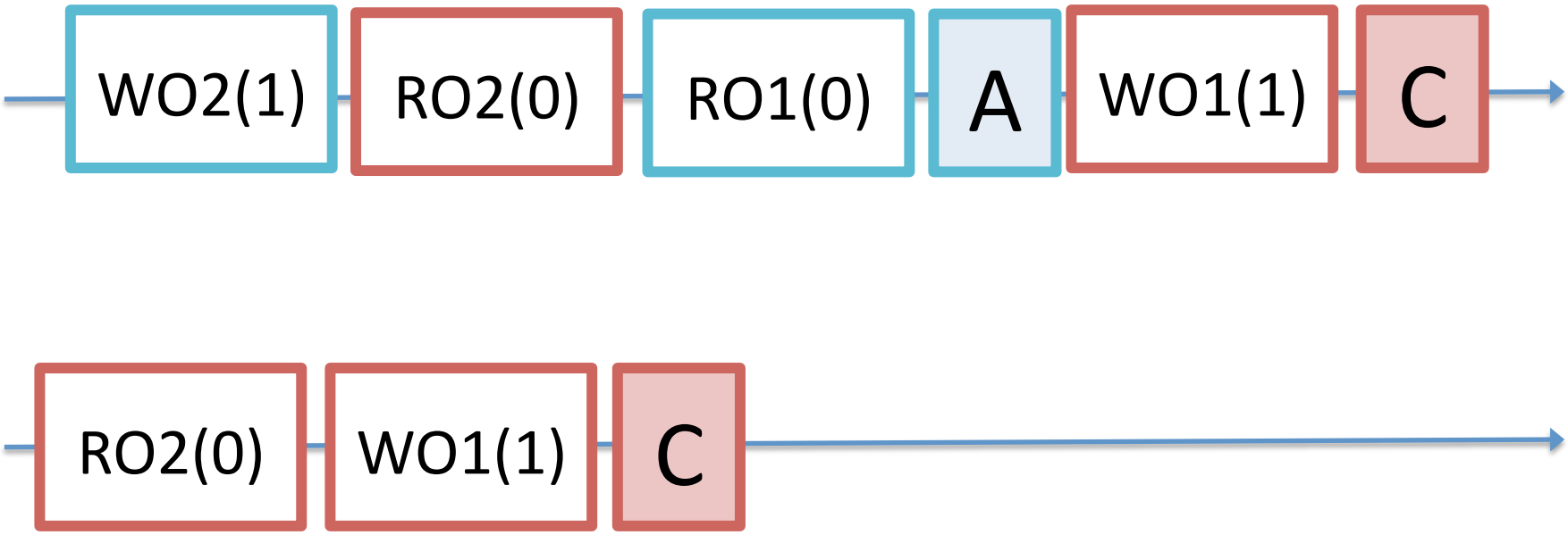


Serializability's definition (Papa79 - View Serializability)

- 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: only committed tx matter!



Serializability: enough for STMs?

- 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!

Example

Initially: x:=1; y:=2

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

If T1 and T2 are atomic, program is correct.

Example

Initially: x:=1; y:=2

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

Otherwise....

Example

Initially: x:=1; y:=2

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

Otherwise....

Example

Initially: x:=2; y:=2

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

Otherwise....

Example

Initially: x:=2; y:=2

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

Otherwise....

Example

Initially: x:=2; y:=2

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

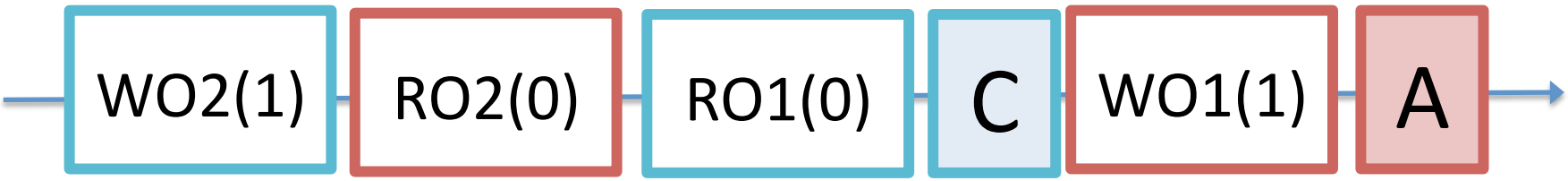
Otherwise....divide by zero!

Opacity
[GK08]

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

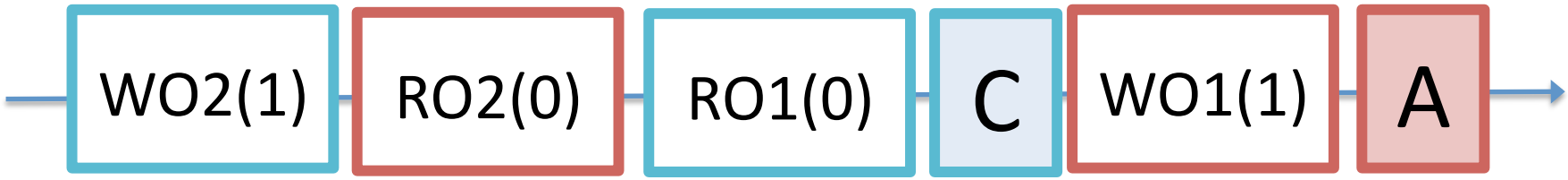
Opacity
[GK08]

- 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 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

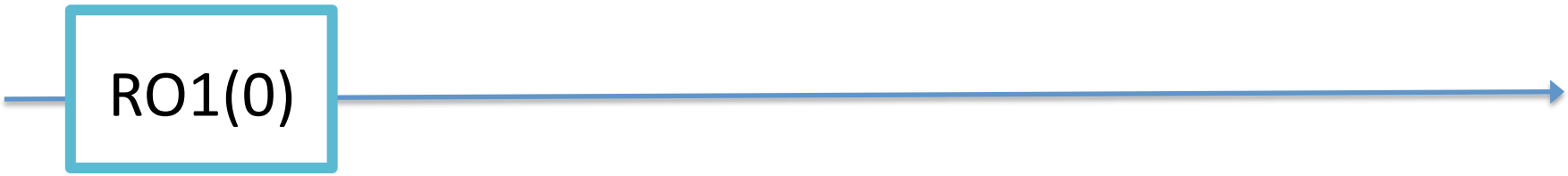
- **Correct** transactions:
  - **commit** is invoked after a finite number of operations
  - either **commit** or perform an infinite number of (low-level) steps
- **Well-formed** histories:
  - every transaction that aborts is immediately repeated until it commits

Desirable: wait-freedom

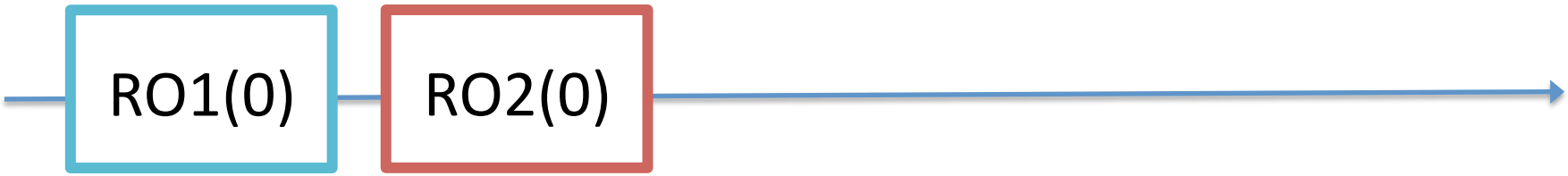
- Every **correct** transaction **eventually commits**
- A transaction may abort a finite number of times as long as it eventually commits

IMPOSSIBLE IN AN ASYNCHRONOUS SYSTEM

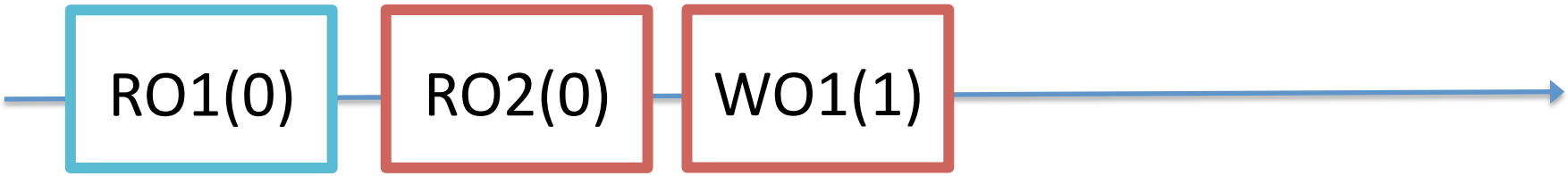
Aborting is unavoidable [Gu09]



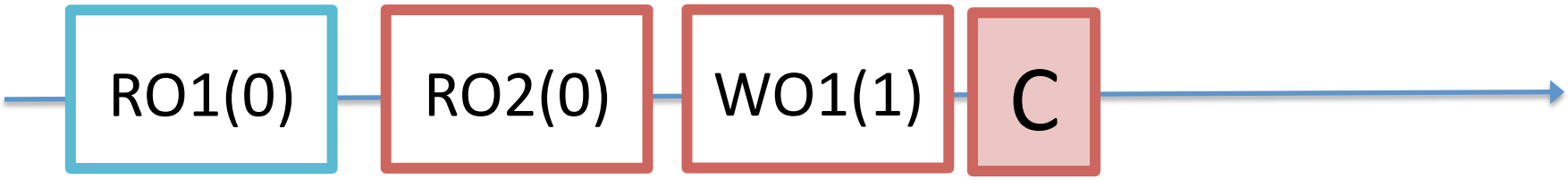
Aborting is unavoidable [Gu09]



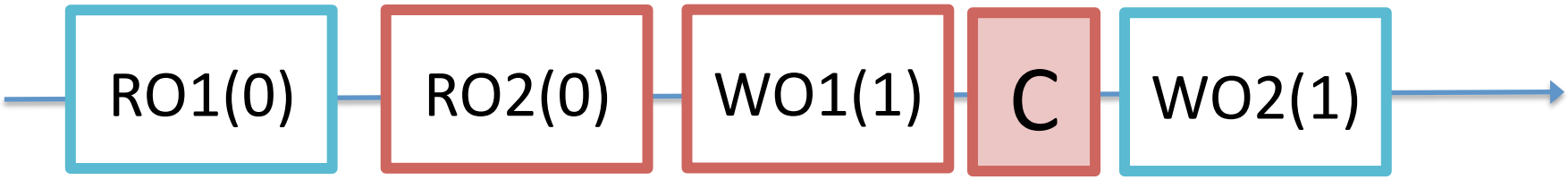
Aborting is unavoidable [Gu09]



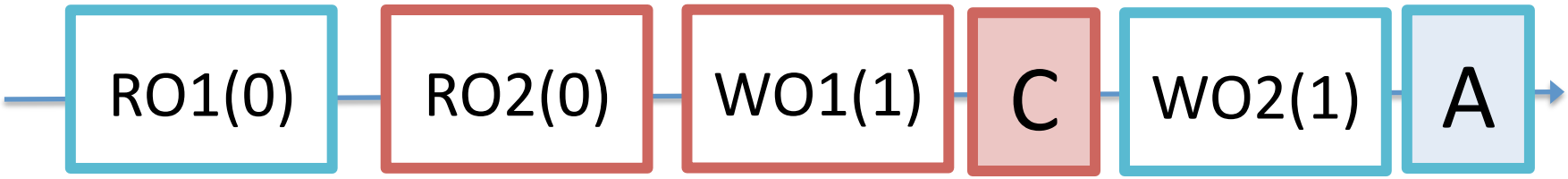
Aborting is unavoidable [Gu09]



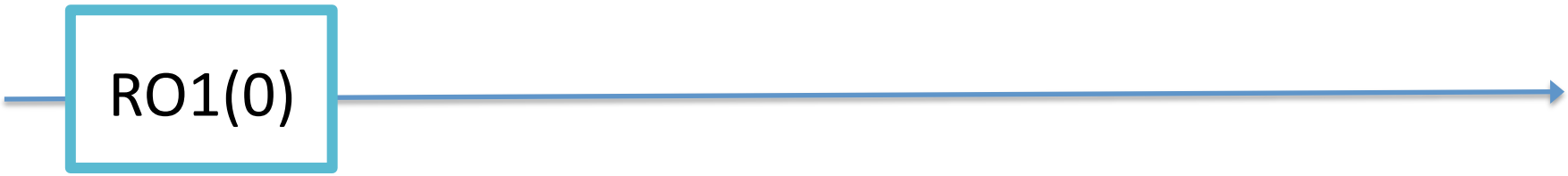
Aborting is unavoidable [Gu09]



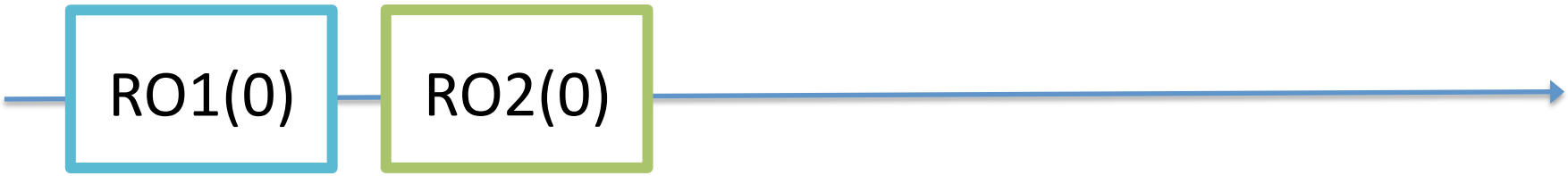
Aborting is unavoidable [Gu09]



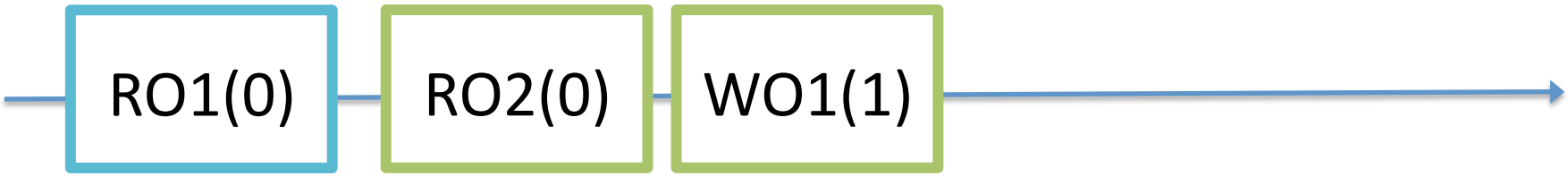
Aborting is unavoidable [Gu09]



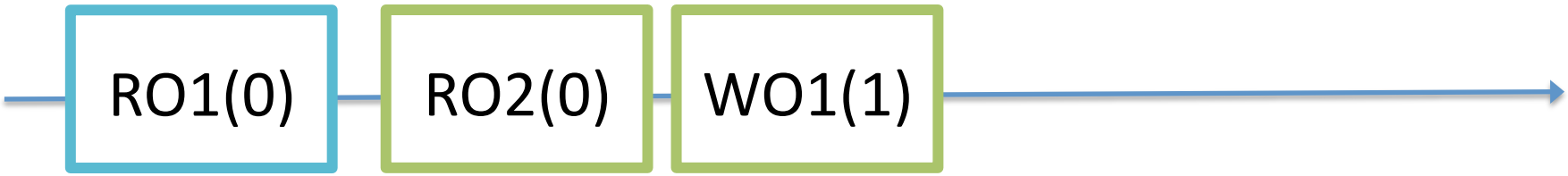
Aborting is unavoidable [Gu09]



Aborting is unavoidable [Gu09]



Aborting is unavoidable [Gu09]



And so on….

Conditional progress: obstruction freedom

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

- **Obstruction-freedom** is possible...
  (examples later)
- …but what to do upon contention?

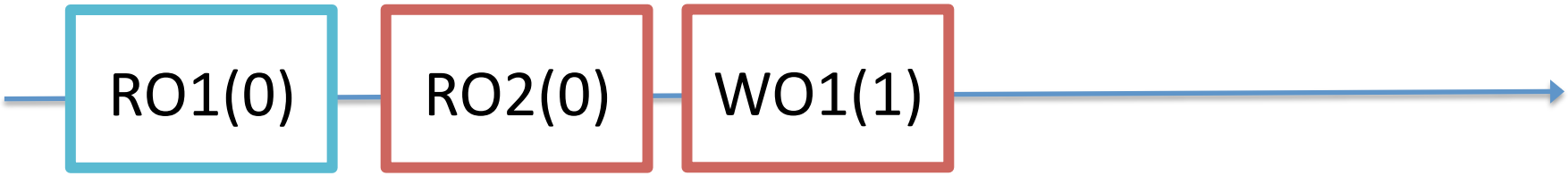
Contention-managers

- Abort is unavoidable
- But want to maximize the number of commits
- Which strategy to use?

Contention-managers encapsulate policies for dealing with contention scenarios.

CM: aggressive

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
- Greedy is O(s)-competitive with off-line clairvoyant scheduler [GHP05, AGHK06]

ToC

PART I

- Non-distributed Transactional Memories
  - Concepts (45 min)
  - **Systems (45 min)**

(break)

PART II

- Distributed Transactional Memories
  - Concepts (45 min)
  - Systems (45 min)

Road map

- Non distributed transactional memory systems
  - DSTM [HLMS03]
  - JVSTM [CS05]
  - TL2 [DSS06]

- Benchmarks
  - micro-benchmarks
  - STMBench7
  - lee-TM
  - Stamp

DSTM [HLMS03]

DSTM

- First STM supporting transactions performing an “unbounded” number of operations (2003)
- Non-blocking algorithm
- Obstruction freedom progress guarantee + out-of-band contention manager

DSTM's algorithm in a nutshell

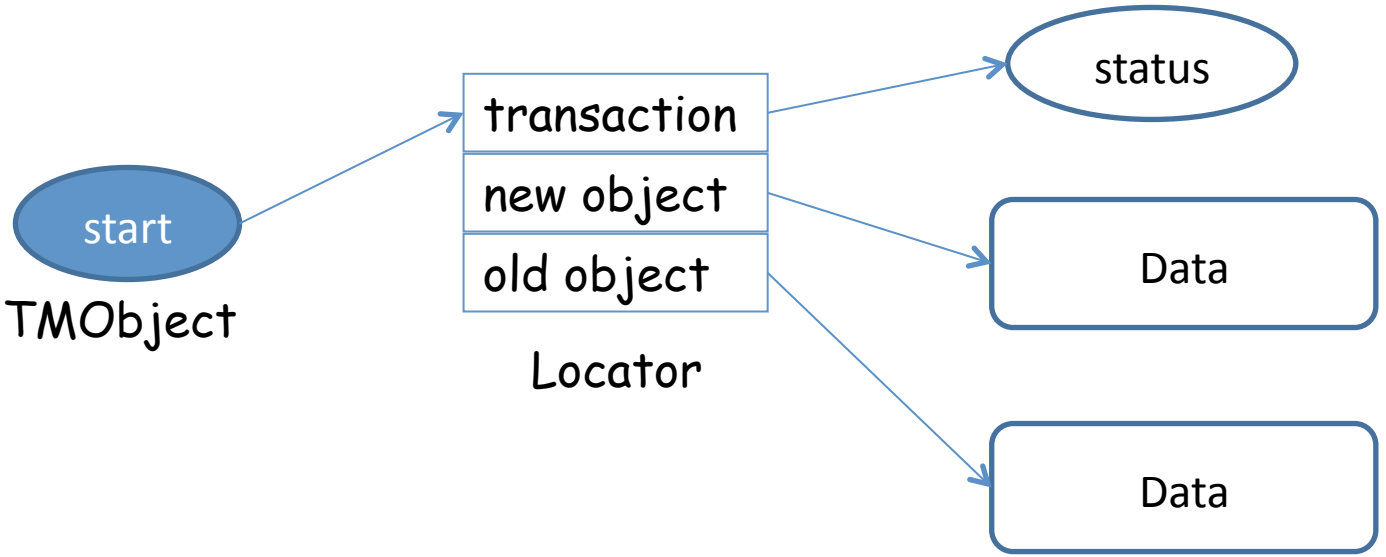
- **Killer write** (ownership)
- **Careful read** (validation)

DSTM's algorithm in a slide

- 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
- 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

DSTM implementation

- Transactional object structure:



DSTM Interface

- A thread that executes transactions must be inherited from TMThread:
  - Each thread can run a single transaction at a time

```
class TMThread : Thread {
    void beginTransaction();
    bool commitTransaction();
    void abortTransaction();
}
```

DSTM Interface

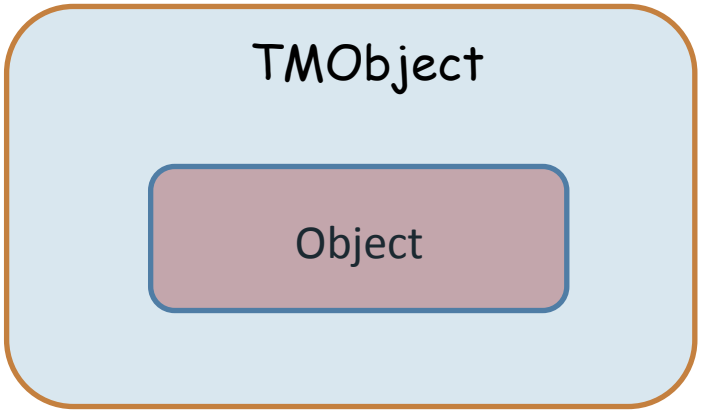
All shared memory objects must implement the TMCloneable interface:

```
inteface TMCloneable {
    Object clone();
}
```

- This method clones the object…
- programmers don't need to handle synchronization issues

DSTM Interface

- In order to make an object transactional, need to wrap it



- TMObject is a container for regular Java objects

DSTM Interface

- Before using a TMObject in a transaction, it must be opened

```
class TMObject {
        TMObject(Object obj);
        enum Mode {READ, WRITE};
        Object open(Mode mode);
}
```

- An object can either be opened for READ or WRITE (and read)

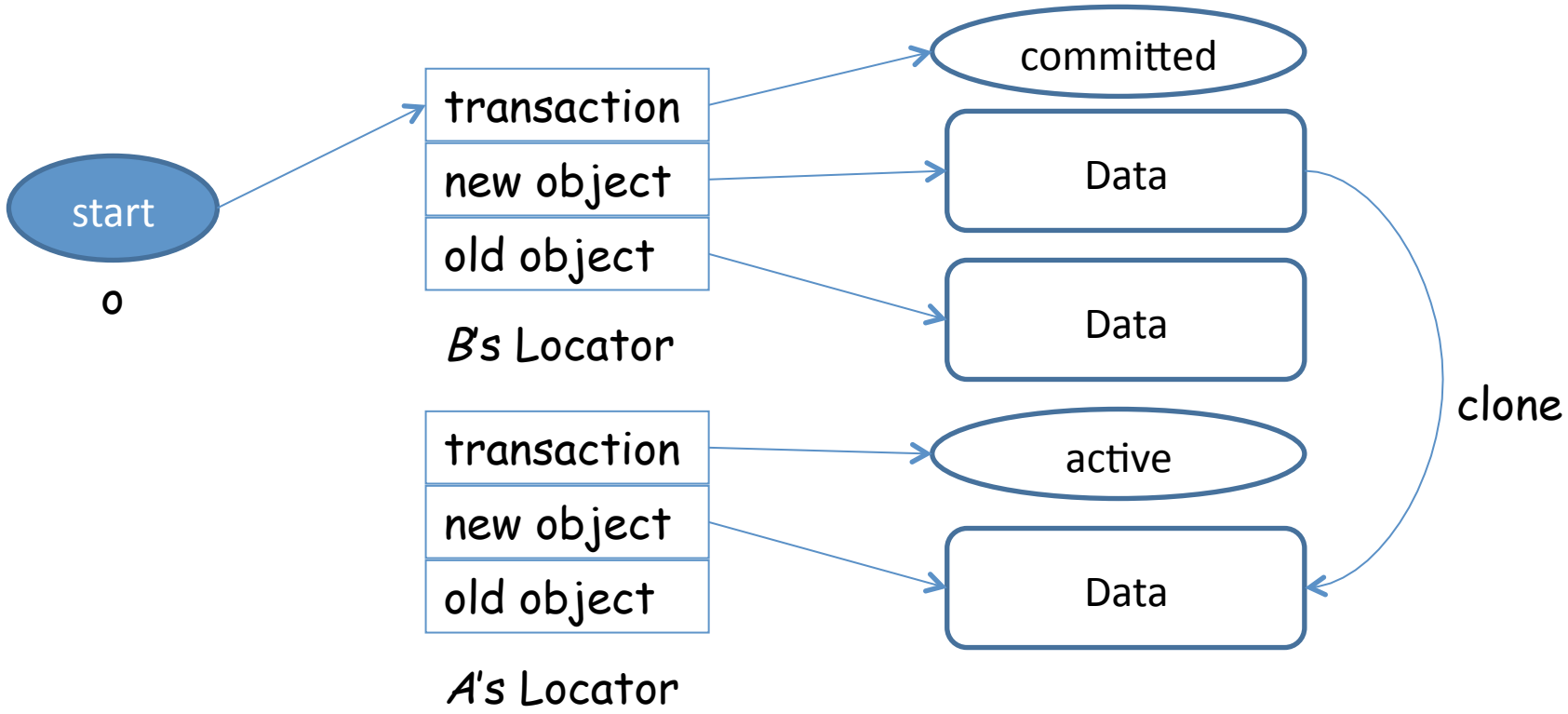
Current object version

- The current object version is determined by the status of the transaction that most recently opened the object in WRITE mode:
  - **committed**: the new object is the current
  - **aborted**: the old object is the current
  - **active**: the old object is the current, and the new is tentative
- The actual version only changes when a commit is successful

Opening an object

- Lets assume transaction *A* opens object *o* in WRITE mode.
- Let transaction *B* be the transaction that most recently opened *o* in WRITE mode.
- We need to distinguish between the following cases:
  - *B* is committed
  - *B* is aborted
  - *B* is active

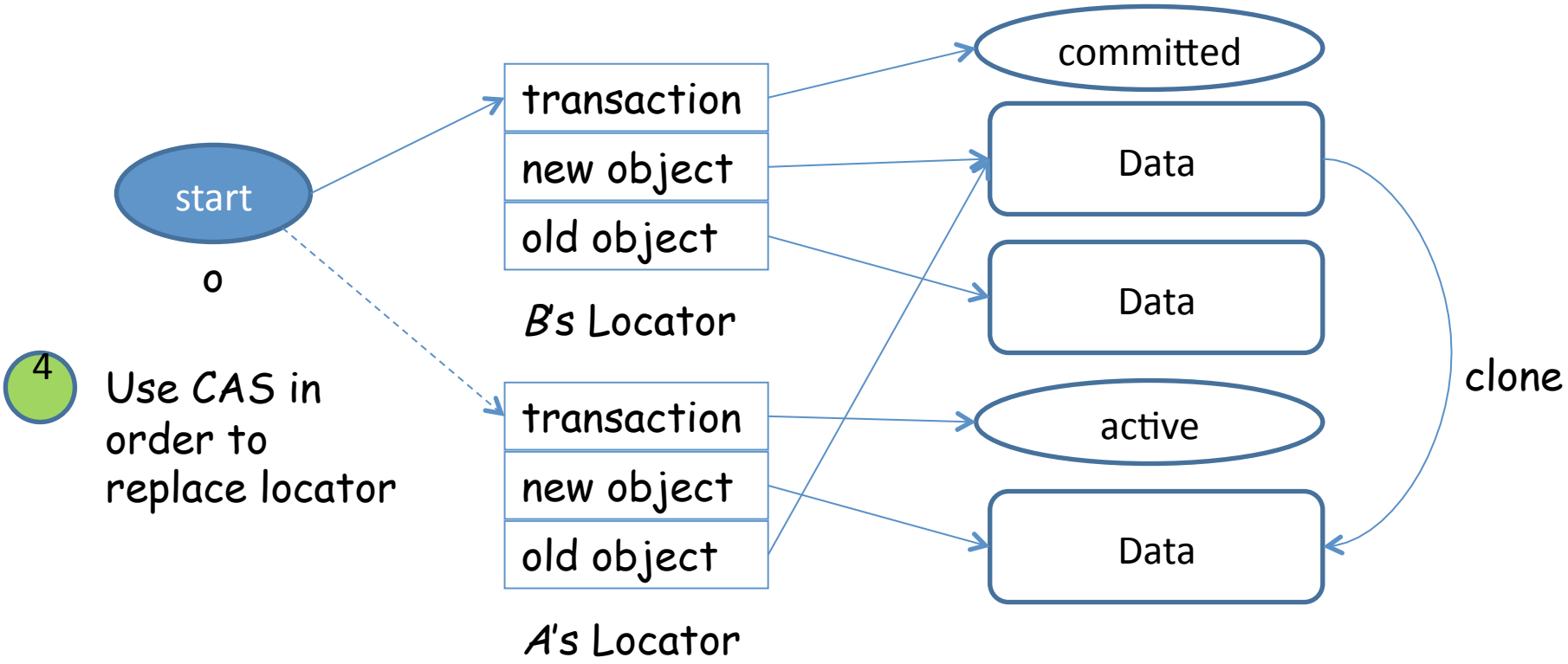
Opening an object – *B* committed



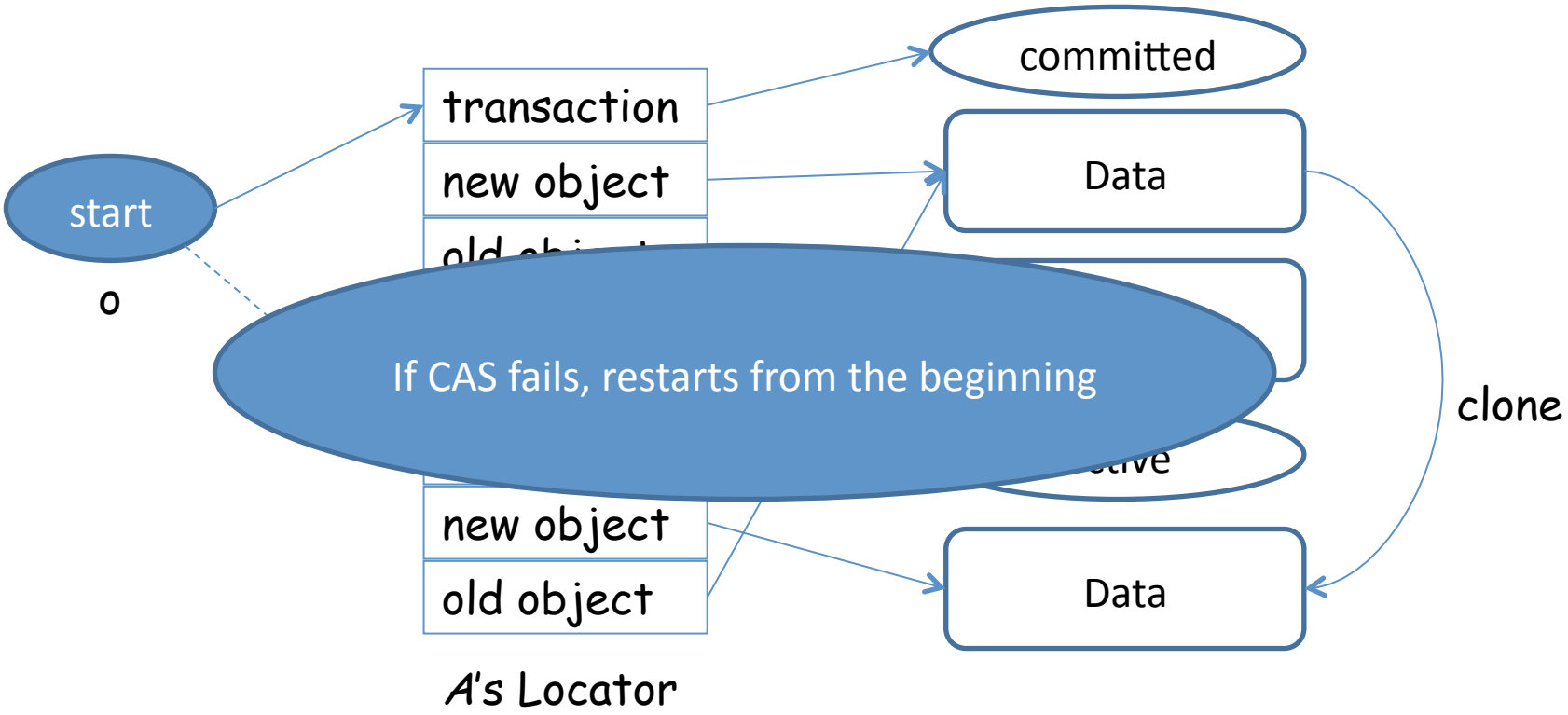
1 *A* creates a new Locator

2 *A* clones the previous new object, and sets new

Opening an object – *B* committed



Opening an object – *B* committed



Opening an object – *B* active

- What if A tries to open an object that is being updated? (*B' locator* is active)
- *A* and *B* are two concurrent, conflicting transactions
- Use **Contention Manager** to decide which should continue and which should abort
- If *B* needs to abort, try to change its status to aborted (using CAS)

Opening an object - READ

- Lets assume transaction *A* opens object *o* in READ mode (i.e. first read operation)
  - validate the whole transaction's readset
  - fetch the current version just as before
  - Add the pair (o, v) to the transaction's readset
- What if the tx had already opened the obj?
  - already read: return the previously read value
  - already written: return the prev. written value

Committing a transaction

- The commit needs to do the following:
  1. Validate the transaction
  2. Change the transaction's status from active to committed (using CAS)

Validating transactions

- What?
  - Validate the objects read by the transaction
- Why?
  - To make sure that the transaction observes a consistent state
- How?
  1. For each pair (o, v) in the readset, verify that v is still the most recently committed version of o
  2. Check that status is still active

Validating transactions

- What?
  - Validate the objects read by the transaction
- Why?
  - To m… …ves a co…
- How?
  1. For each pair (o, v) in the readset, verify that v is still the most recently committed version of o
  2. Check that status is still active

If the validation fails, throw an exception so the user will restart the transaction from the beginning

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?

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

Java Versioned Software Transactional Memory (JVSTM)

JVSTM - overview

- 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)

Java Versioned Software Transactional Memory

```
public class Transaction {
  public static void start();
  public static void abort();
  public static void commit();
}
```

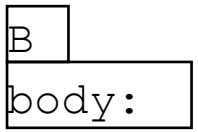
```
public class VBox<E> {
  public VBox(E initial);
  public E get();
  public void put(E newE);
}
```

```
public class Counter {
  private VBox<Long> count = new VBox<Long>(0L);

  public long getCount() {
    return count.get();
  }

  public @Atomic void inc() {
    count.put(getCount() + 1);
  }
}
```

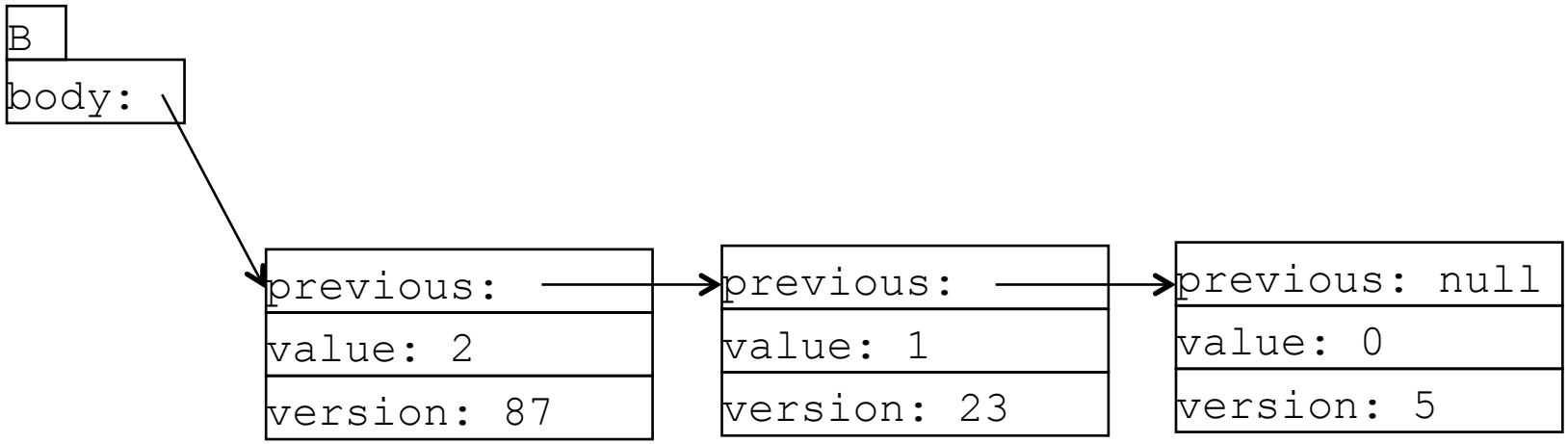
Versioned Boxes

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

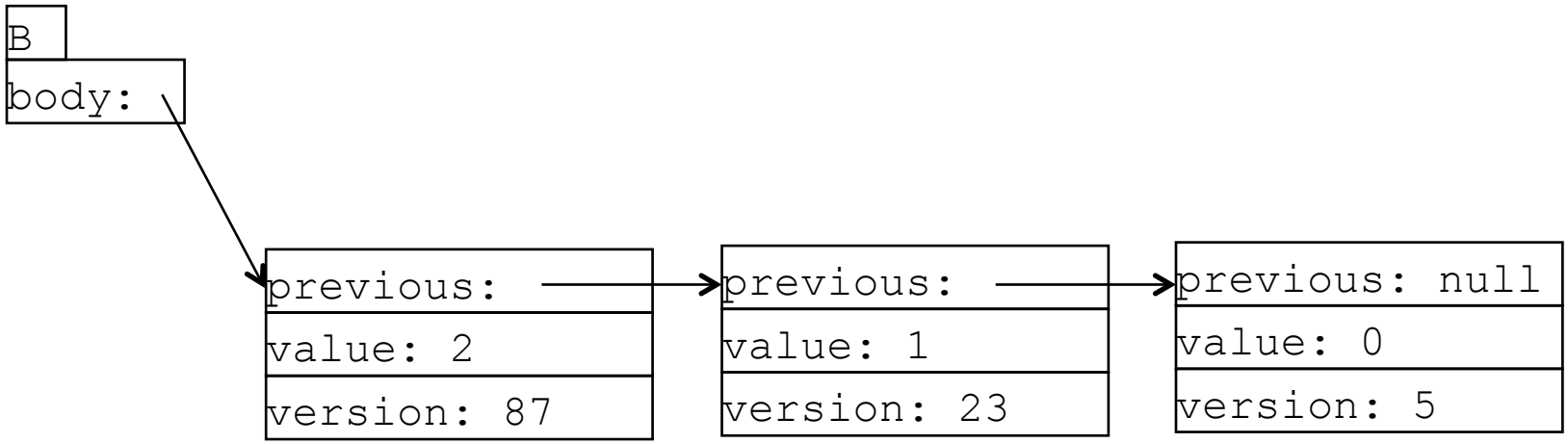
Versioned Boxes



Versioned Boxes



Versioned Boxes




```
VBox<T>

+ T get()
+ put(T value)
```

JVSTM algorithm

- upon begin tx, read GVN and assigned it to tx's snapshot ID (sID)
- upon read on object o:
  - if o is in tx'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 tx' sID"

- if tx is not read-only, add o to readset

JVSTM algorithm

- upon write, just add to the writeset
  - no early conflict detection
  - optimistic approach

- upon commit
  - validate readset:
    - abort if any object changed
  - acquire new sID (atomic increase of GVN)
  - apply writeset:
    - add new version in each written VBox

Commit Phase

```
commit() {
  GLOBAL_LOCK.lock();
  try {




  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Commit Phase

```
commit() {
  GLOBAL_LOCK.lock();
  try {
    if (validate()) {



    }
  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

Commit Phase

```
commit() {
  GLOBAL_LOCK.lock();
  try {
    if (validate()) {
      int newTxNumber = globalCounter + 1;
      writeBack(newTxNumber);
      globalCounter = newTxNumber;
    }
  } finally {
    GLOBAL_LOCK.unlock();
  }
}
```

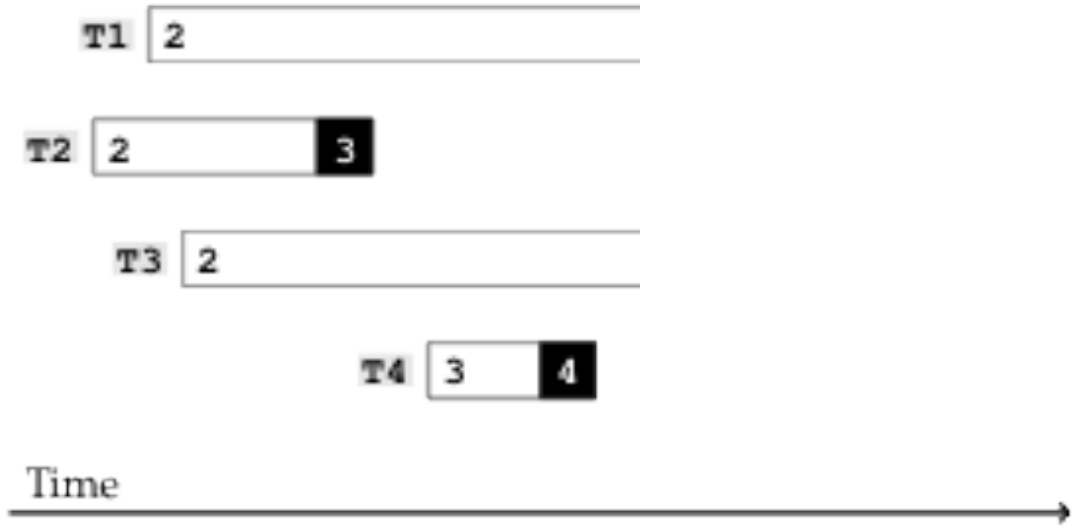
Transactions

Transactions

Transactions

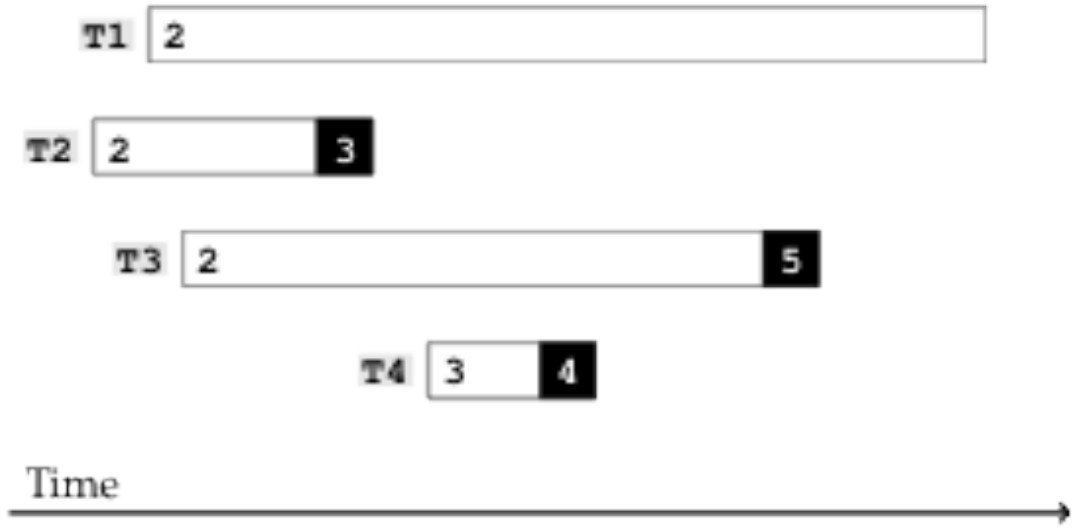
Transactions

Transactions



Transactions

Transactions



Transactional Locking II (TL)

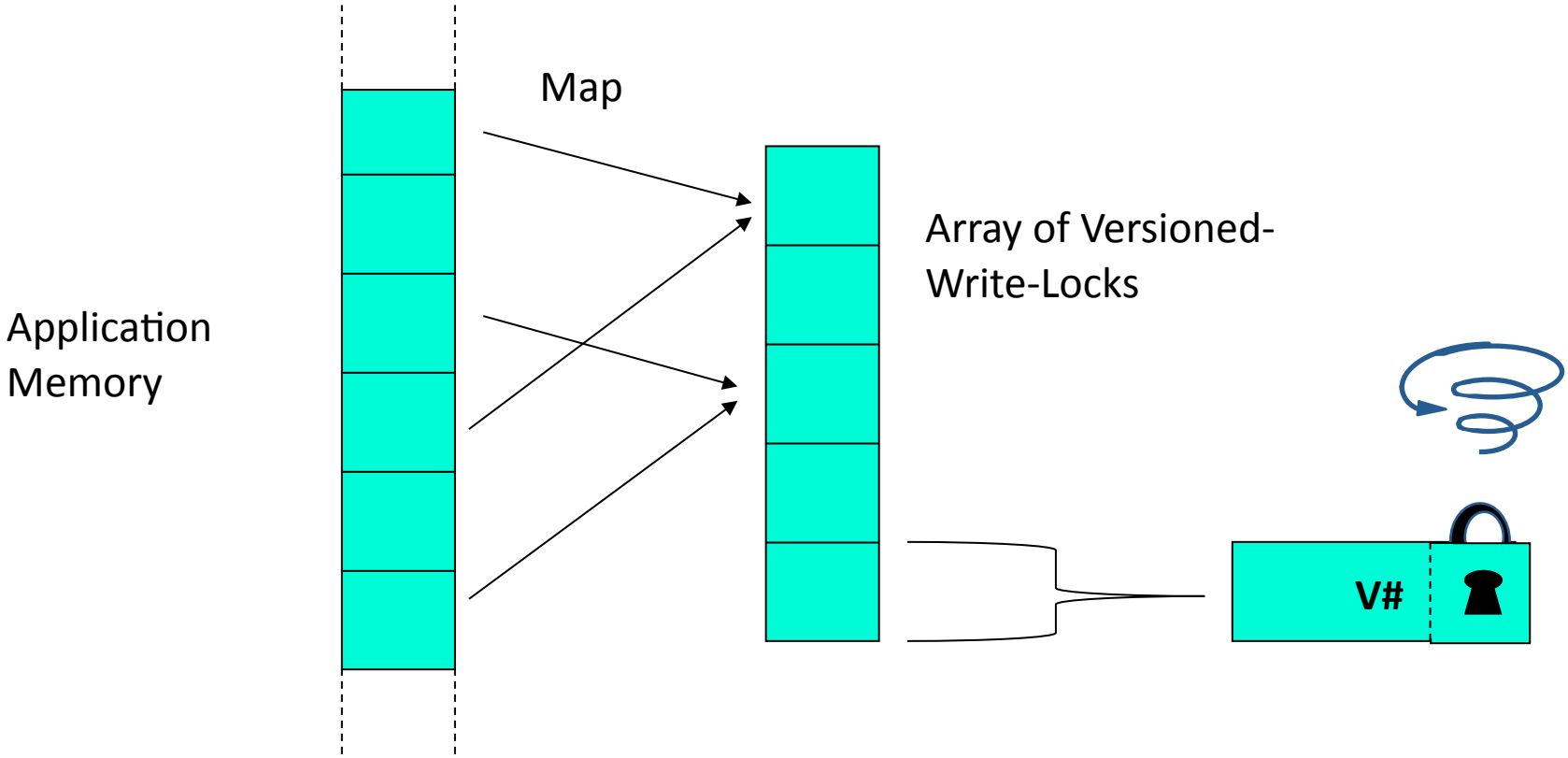
TL2 - Overview

- Single-versioned, word-based STM
- Commit time locking strategy:
  - versioned write locks
- Achieve opacity without re-validating readset at each read:
  - use centralized global version number to define memory snapshots
    - allow to efficiently detect non-opaque schedules
  - several optimizations to reduce contention on GVN

TL2 - Interface

```
…
elem_t* elemPtr;
…
TM_BEGIN();

long pop      = (long)TM_READ(queuePtr->pop);
long push     = (long)TM_READ(queuePtr->push);
long capacity = (long)TM_READ(queuePtr->capacity);

long newPop = (pop + 1) % capacity;
if (newPop == push) {
   elemPtr = NULL;
   } else {
      void** elements = (void**)TM_READ_P(queuePtr-> elements);
      elemPtr = (pair_t*) TM_READ_P(elements[newPop]);
      TM_WRITE(queuePtr->pop, newPop);
}

TM_END();
…
…
```

Locking Design Choices for Word Based STMs



PS = Lock per Stripe (separate array of locks)

PO = Lock per Object (embedded in object)

TL2 - Algorithm

- begin:
  - sample GVC and assign it to the tx's read version number (*vn*)
- read:
  - check if data is in writeset (Bloom Filter)
  - load corresponding versioned write lock (*vwl*)
  - if busy or vwl version > rv abort
  - load data
  - check if vwl has changed : abort
  - store address of data in read-set (list)

TL2 - Algorithm

- write:
  - just add (address,value) to write-set
- commit:
  - acquire write locks
    - unordered = deadlock chance = chance tx aborts
  - increment gvc (via CAS)
  - validate readset, abort upon:
    - new value present (wvl's version > rv)
    - busy wvl
  - writeback data + wvl,
  - release all locks

TL2 – Read-only transactions

- In case transactions are pre-declared as read-only it can be avoided to build a readset

- upon read:
  - load corresponding versioned write lock (*vwl*)
  - if busy or vwl version > rv abort
  - load data
  - check if vwl has changed => abort

Benchmarks

Benchmarks

- First generation:
  - micro-benchmarks
  - simple data structures:
    - list, skiplist, RBTree
  - trivial business logic:
    - serie of put, get, delete
- DSTM2 Microbenchmark's suite

Benchmarks

- II generation:
  - complex, realistic applications
  - heterogeneous workloads

- STMBench7
- Lee Benchmark
- Stamp

STMBench7 [JKV07]

- Inherits most of application logic from OO7:
  - complex benchmark for OO databases
  - CAD/CAM/CASE applications

STMBench7 [JKV07]

- Very large, complex data structures
  - Large tree with graph in each leaf
  - 6 indexes
  - Can be traversed in any direction
  - highly dynamic data strucutre
  - mix of short and long transactions:
    - long and short traversals
    - short operations
    - structural modifications

STMBench7 [JKV07]

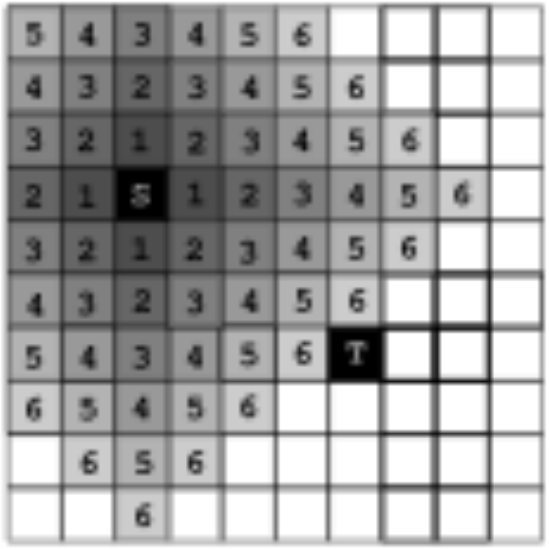
- Baseline comparison with different grains of locking:
  - Coarse:
    - single RW lock
  - Medium:
    - one RW lock per level
    - global RW lock for structure modifications

Lee-TM [AKWKLJ08]

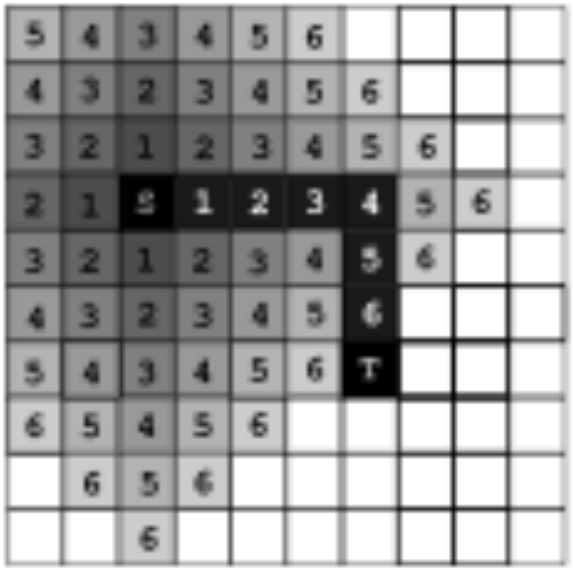
- Based on the Lee's algorithm for circuit routing
- Concurrent laying routes between two end-points on a grid:
  1. Expansion phase
  2. Backtracking phase

The Lee algorithm

Expansion Phase



Backtracking Phase



Lee-TM

- Heterogeneous workload:
  - distance between end-points affects transaction length

- Complex application logic:
  - hard to define fine-grained locking schemes
  - baseline comparison is with coarse/medium strategies for the locking of the grid

Stamp [MCKO08]

- Suite of eight multi-threaded applications:
  - bayes: Bayesian network learning
  - genome: gene sequencing
  - intruder: network intrusion detection
  - kmeans: K-means clustering
  - labyrinth: maze routing
  - ssca2: graph kernels
  - vacation: client/server travel reservation system
  - yada: Delaunay mesh refinement (Ruppert's algorithm)

Part II

Distributed
Transactional Memories

ToC

PART I

- Non-Distributed Transactional Memories
  - Concepts (45 min)
  - Systems (45 min)

(break)

PART II

- Distributed Transactional Memories
  - **Concepts (45 min)**
  - Systems (45 min)

An obvious evolution

- Real, complex STM based applications are starting to appear:
  - Apache Web Server
  - FenixEDU
  - Circuit Routing
  - …
- …and are being faced with classic production environment's challenges:
  - scalability
  - high-availability
  - fault-tolerance

Distributed STMs

Distributed STMs

- At the convergence of two main areas:

Distributed Shared Memory **Distributed Databases**

Distributed STMs

- At the convergence of two main areas:

Distributed Shared Memory **Distributed Databases**

= Similar goals & abstraction:

= Hide distribution via single system image

≠ Explicit lock based synchronization

- Strongly consistent DSMs:
  - ↑ Very easy to program
  - ↓ Bad performance due to too frequent remote synchronizations
- Relaxed consistency DSMs:
  - ↑ Reduced # sync. = better performance
  - ↓ Complex to program and reason about

Transactions allow to:

1. Transparently deal with remote critical races
2. Boost performance by batching any remote synchronization during the commit phase

Distributed STMs

- At the convergence of two main areas:

Distributed Shared Memory

= Similar goals & abstraction:

- = Hide distribution via single system image
- ≠ Explicit lock based synchronization

- Strongly consistent DSMs:
  - ↑ Very easy to program
  - ↓ Bad performance due to too frequent remote synchronizations
- Relaxed consistency DSMs:
  - ↑ Reduced # sync. = better performance
  - ↓ Complex to program and reason about

Distributed Databases

≈ Relied on ACID transactions for decades

- ≠ Durability regarded as optional in STMs

≠ Much heavier programming interface

- ≠ SQL vs direct access to in-memory variables

Transactions allow to:

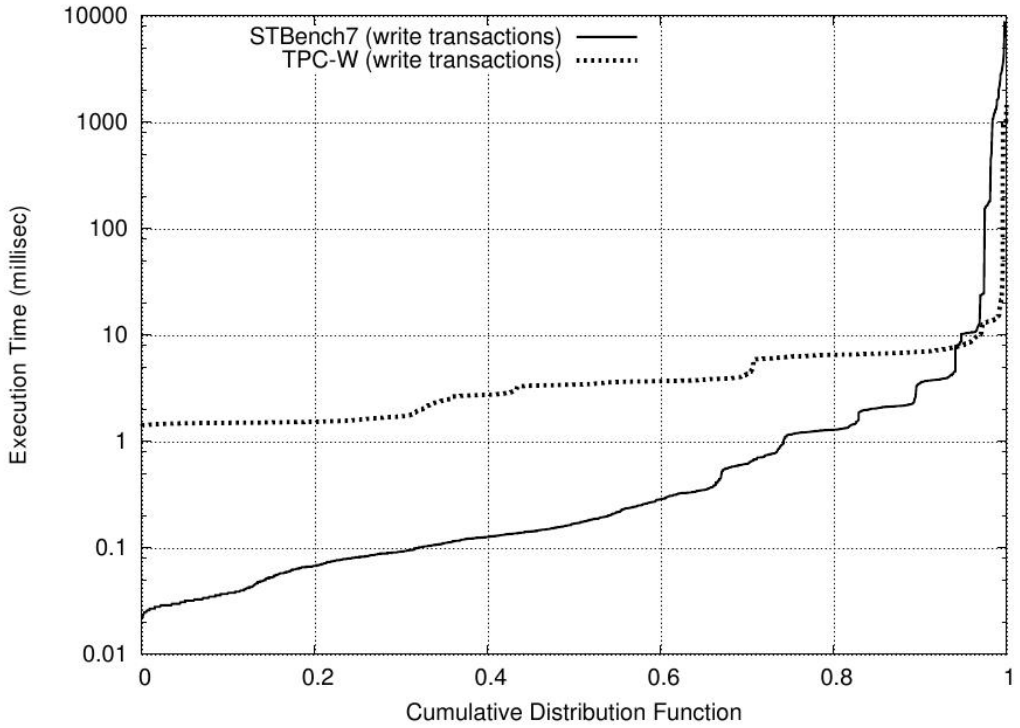
1. Transparently deal with remote critical races
2. Boost performance by batching any remote synchronization during the commit phase

Distributed STMs

- At the convergence of two main areas:

>70% xacts are 10-100 times shorter:

- larger impact of coordination



2. Boost performance by batching any remote synchronization during the commit phase

Distributed STMs

- At the convergence of two main areas:

Distributed Shared Memory

= Similar goals & abstraction:

- = Hide distribution via single system image
- ≠ Explicit lock based synchronization

- Strongly consistent DSMs:
  - ↑ Very easy to program
  - ↓ Bad performance due to too frequent remote synchronizations
- Relaxed consistency DSMs:
  - ↑ Reduced # sync. = better performance
  - ↓ Complex to program and reason about

Transactions allow to:

1. Transparently deal with remote critical races
2. Boost performance by batching any remote synchronization during the commit phase

Distributed Databases

≈ Relied on ACID transactions for decades

- ≠ Durability regarded as optional in STMs

≠ Much heavier programming interface

- ≠ SQL vs direct access to in-memory variables

≠ DBs are "sandboxed" environments:

- ≠ Inconsistent DB transactions return stale data
- ≠ Inconsistent STM transactions can be way more harmful (memory wipe-out, infinite cycles...)

Natural source of inspiration for DSTMs...
though DSTMs have
unique, challenging requirements!

Existing Distributed STMs

- Very recent research area....
- Only a handful of existing prototypes, e.g.:
  - DMV [MMA06]
  - DiSTM [KAJLKW08]
  - ClusterSTM [BAC08]
  - D^2STM [CRRC09]
  - ALC [CRR10]

Programming models for DSTMs

What API should a DSTM expose?

Single System Image

vs

Partitionable Global Address Space

Single System Image

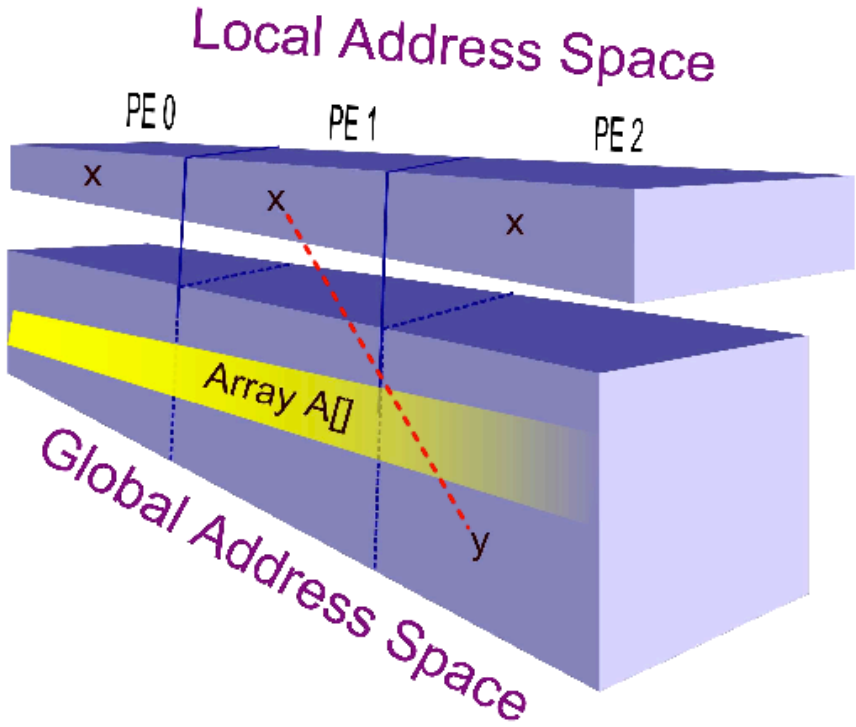
- Data distribution totally hidden to the programmer
- API identical to non-distributed STMs

Single System Image

- Pros:
  - simple programming model
  - easy to port existing applications
- Cons:
  - no control over data/code locality
  - may lead to poor performance
  - existing SSI systems target small scale clusters

Partitionable Global Address Space

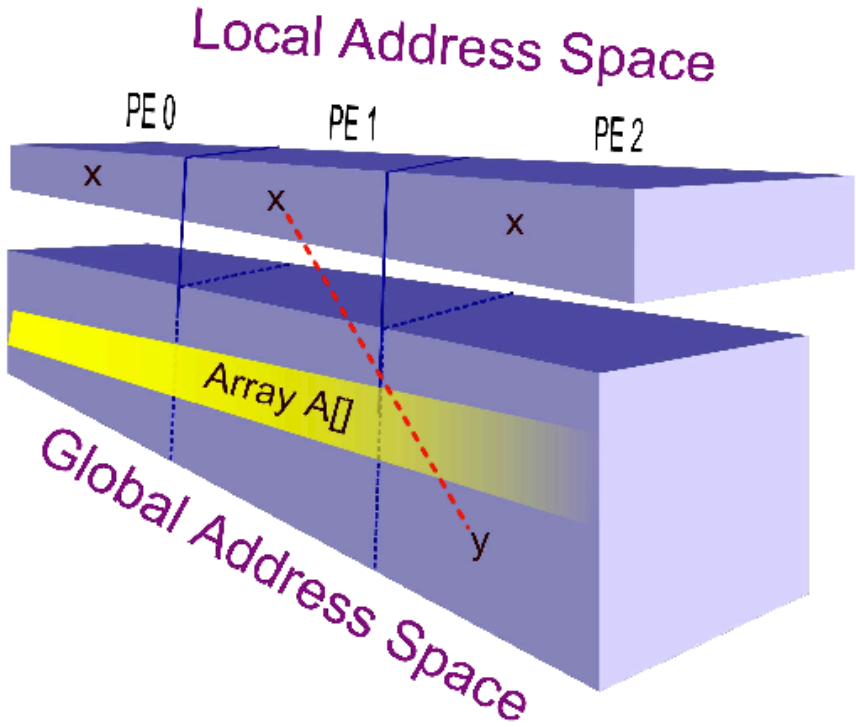
- Explicit distinction between local and remote data partitions:
  - private variables
  - shared variables residing on some node
  - shared data structures distributed across multiple nodes
- Allow controlling the node on which code must execute



Partitionable Global Address Space

Pros:

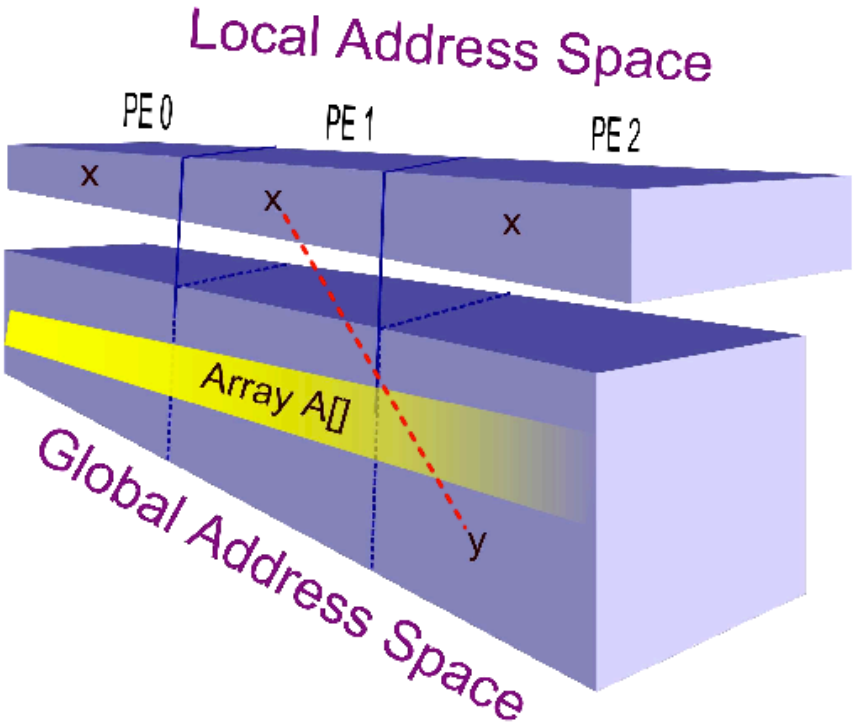
- allow distributed/parallel computing specialists to carefully optimize for data/execution locality:
  - model born in HPC community
  - simpler than MPI, more efficient than DSMs



Partitionable Global Address Space

Cons:

- more complex programming model than SSI
- contradicts one of the key motivations of STMs:
  - simplifying parallel programming



Programming models for DTMs

Control Flow

vs

Data Flow

Control flow model

- data is **statically** assigned to a **home node** and does not change over time
- data manipulation is either carried out:
  - at the node where data resides (RPC style) or
  - a copy of the data is:
    1. fetched on the node where the transaction is originated
    2. written back at the home node

Distributed update must be atomic!

Data Flow model

- transactional **code is immobile**
- **objects move** from node to node depending on data access patterns:
  - upon **write** on object *o* by processor *p*, *p* must:
    - **locate** the current position of *o*
    - acquire **ownership** of *o*
  - upon **read** on object *o* by processor *p*, *p* must:
    - **locate** the current position of *o*
    - acquire a **read-only copy** of *o*

Data Flow model

- Conflicting accesses detected locally during transaction execution:
  - conflict handling delegated to local contention manager
- Avoids distributed coordination
- Locating objects can be very expensive:
  - how to efficiently track dynamic position of objects?

Control Flow vs Data Flow

Pros

- rely on fast (typically O (1)) data location mechanism
- allow easy integration of caching schemes:
  - reduce further misses costs

Pros

- Maximizes benefits from data locality in large scale systems:
  - allows moving data close to sharing clients
  - autonomic data relocation based on access pattern

Control Flow vs Data Flow

Cons

- static data placement may lead to poor data locality, e.g.:
  - changing data access patterns

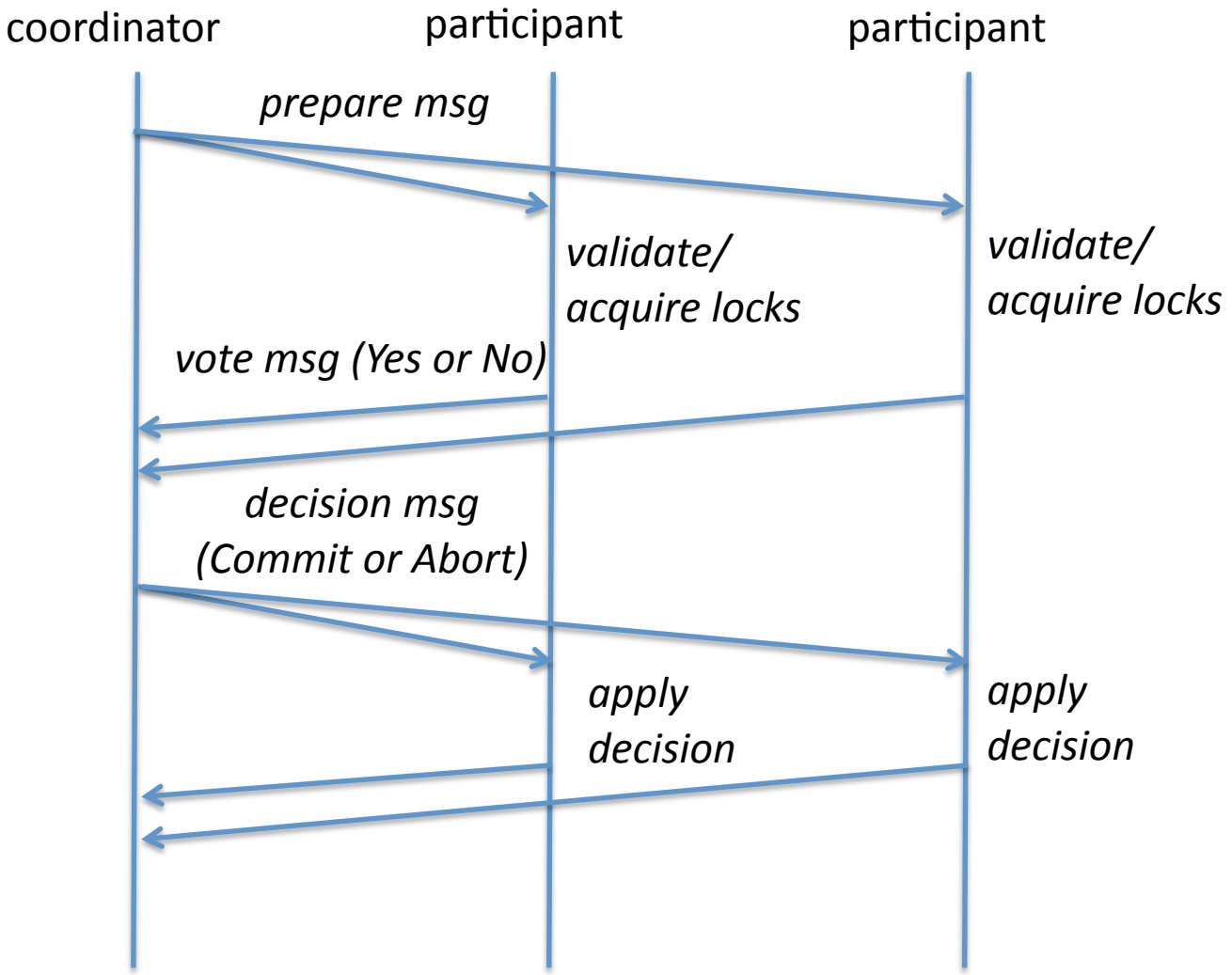
Cons

- locating data owner is costly
- performance is highly sensitive to application locality:
  - hard to integrate caching schemes to reduce misses
  - distributed coordination
  - no existing prototype, only theoretical papers...

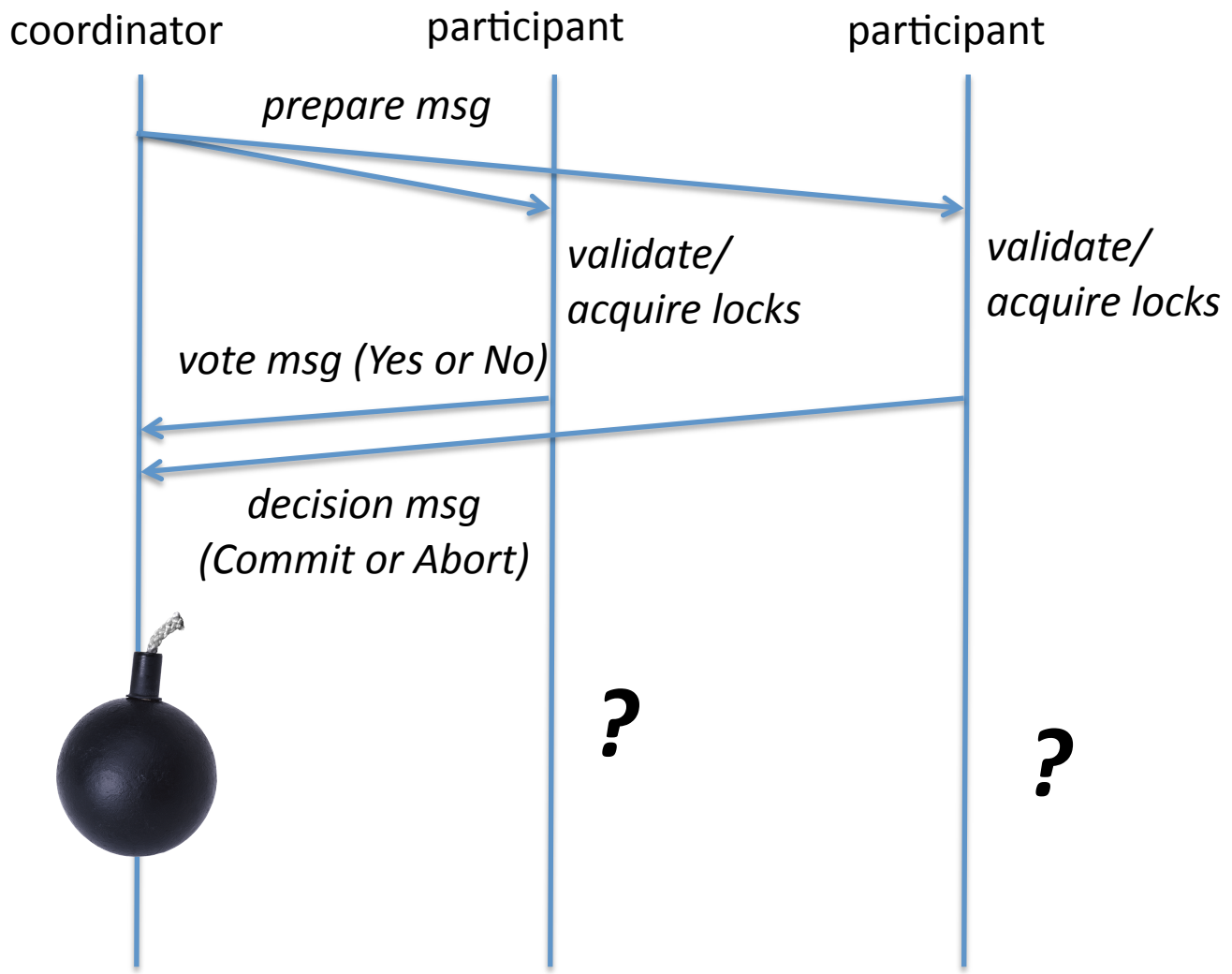
Distributed coordination (required in control flow)

- Ensure atomicity of distributed transaction
  - either all nodes commit the transactions
  - or none of them does
- Two Phase Commit protocol:
  - Prepare phase
    - coordinator gathers participants' votes
  - Decision phase
    - commit decision only if all participants voted affirmatively

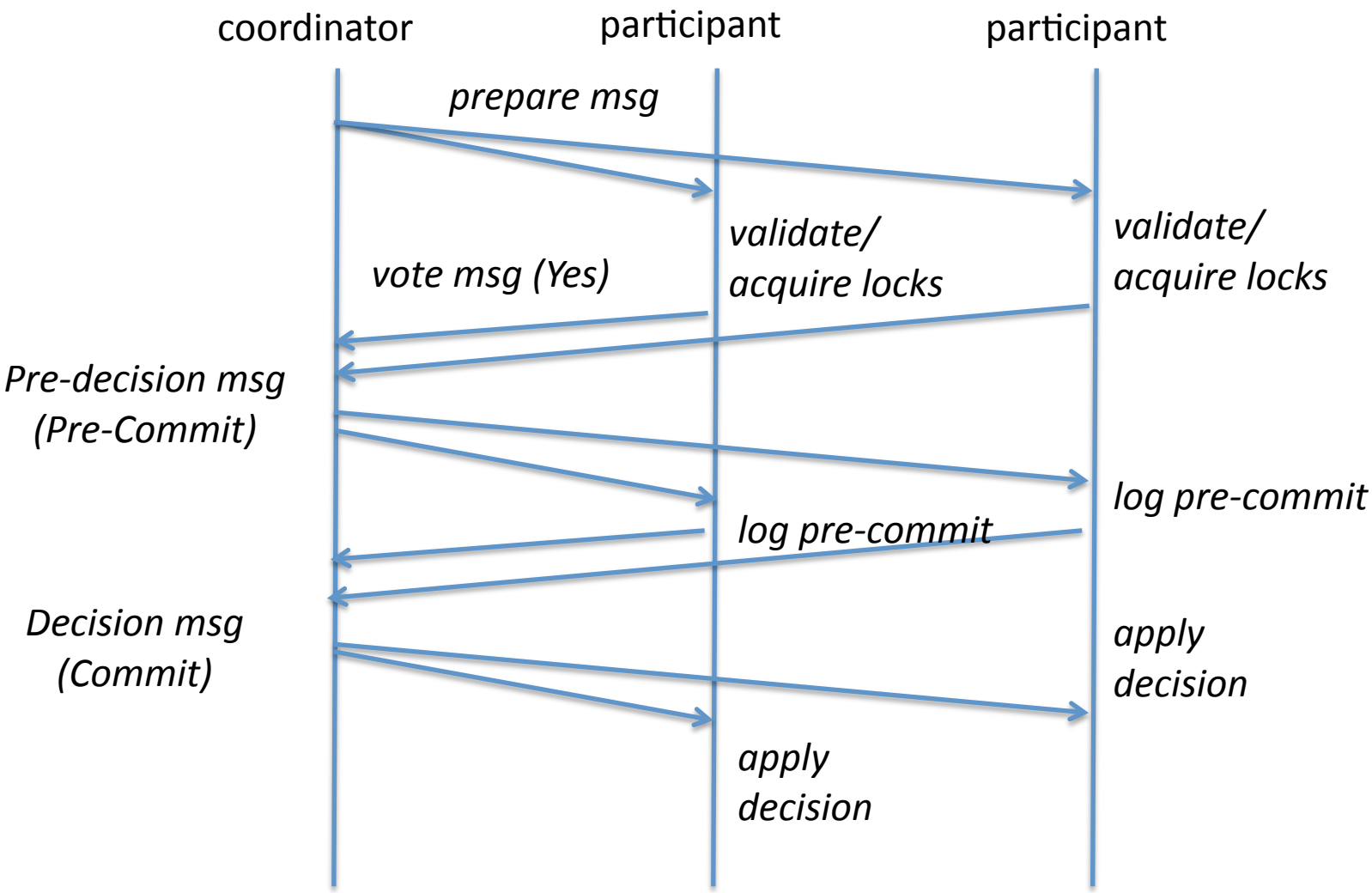
2PC


coordinator
participant
participant
prepare msg
validate/
acquire locks
validate/
acquire locks
vote msg (Yes or No)
decision msg
(Commit or Abort)
apply
decision
apply
decision

2PC: blocking



3PC


coordinator
participant
participant
prepare msg
validate/
acquire locks
validate/
acquire locks
vote msg (Yes)
Pre-decision msg
(Pre-Commit)
log pre-commit
log pre-commit
Decision msg
(Commit)
apply
decision
apply
decision

Distributed coordination (required in control flow)

- Three Phase Commit protocol:
  - Non-blocking
  - Much slower

Transactional Data Replication

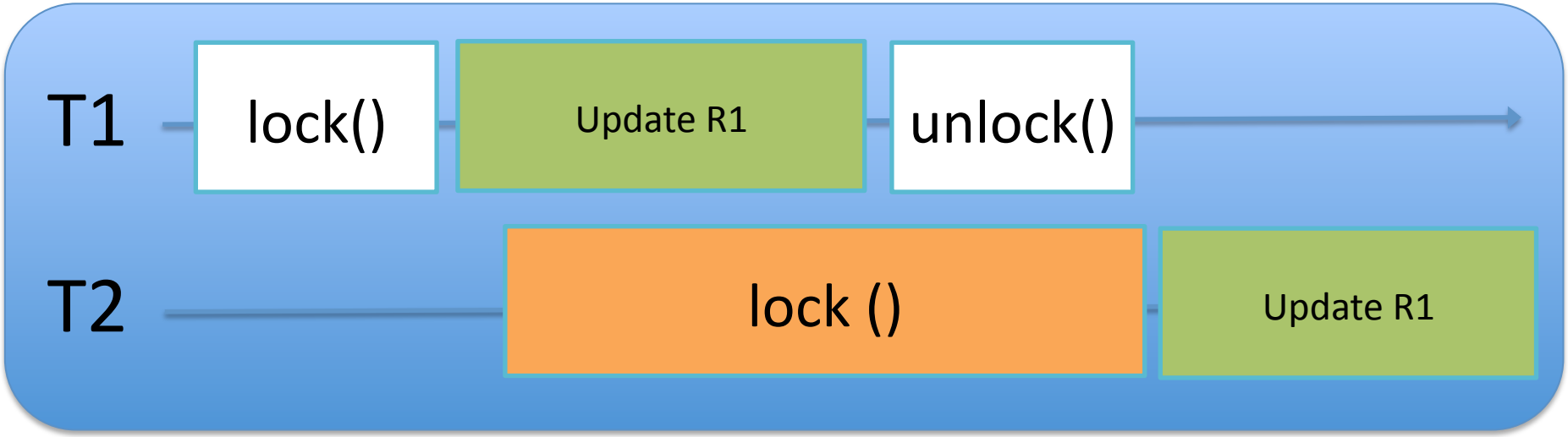
Data replication

- Can be useful for:
  - Performance
  - Fault-tolerance
- Performance
  - Read operations on local data
- Fault-tolerance
  - Even if one node crashes data continues available

Challenge

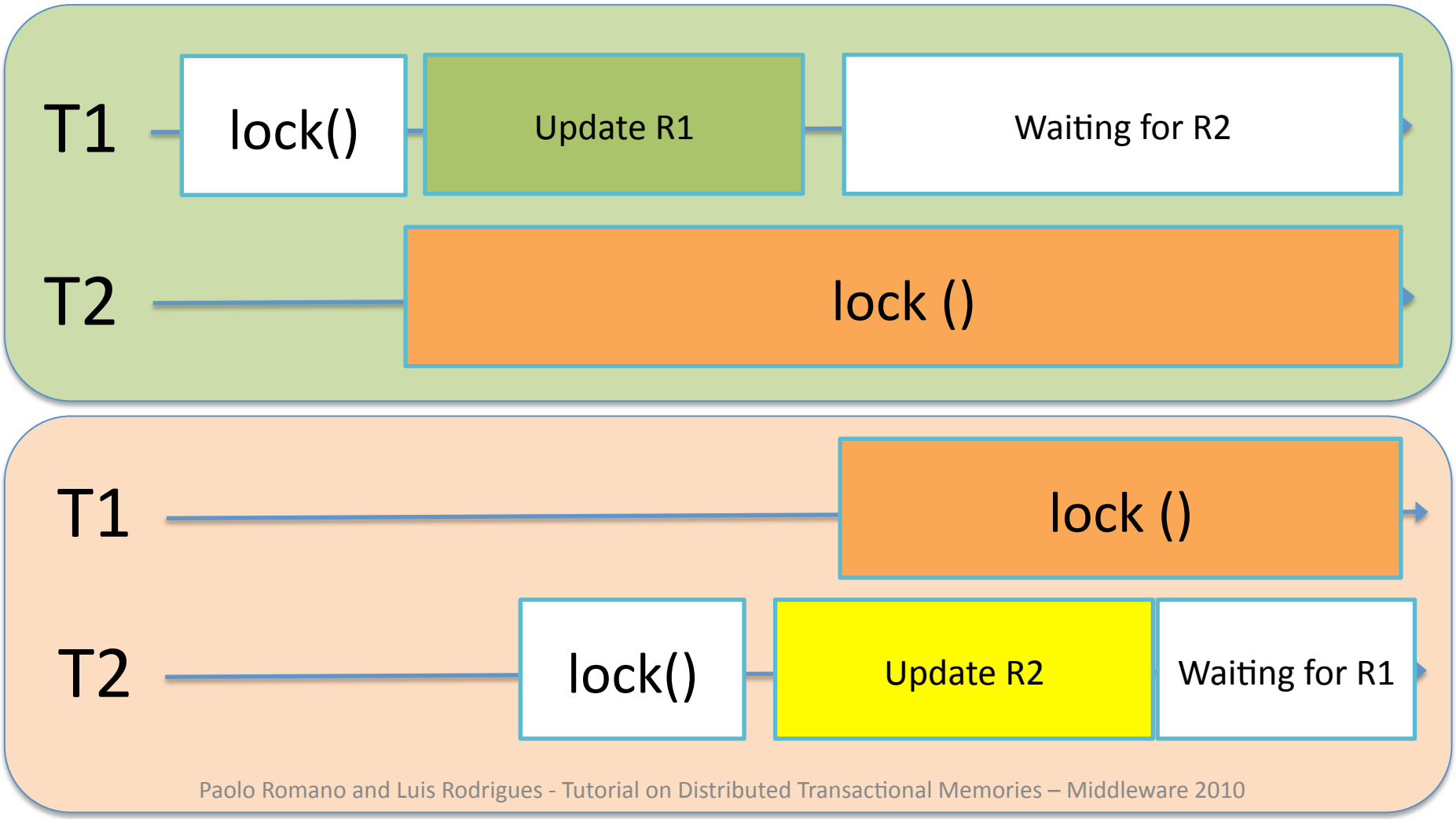
- Distributed coordination when:
  - The transaction commits (all-or-none the copies must be updated)
  - But also for ensuring same serialization order across all replicas!

Replicated transaction: single lock!



In absence of replication, there's no chance to fall into deadlocks with a single lock…
what if we add replication?

Replicated transaction: single lock!



Challenge

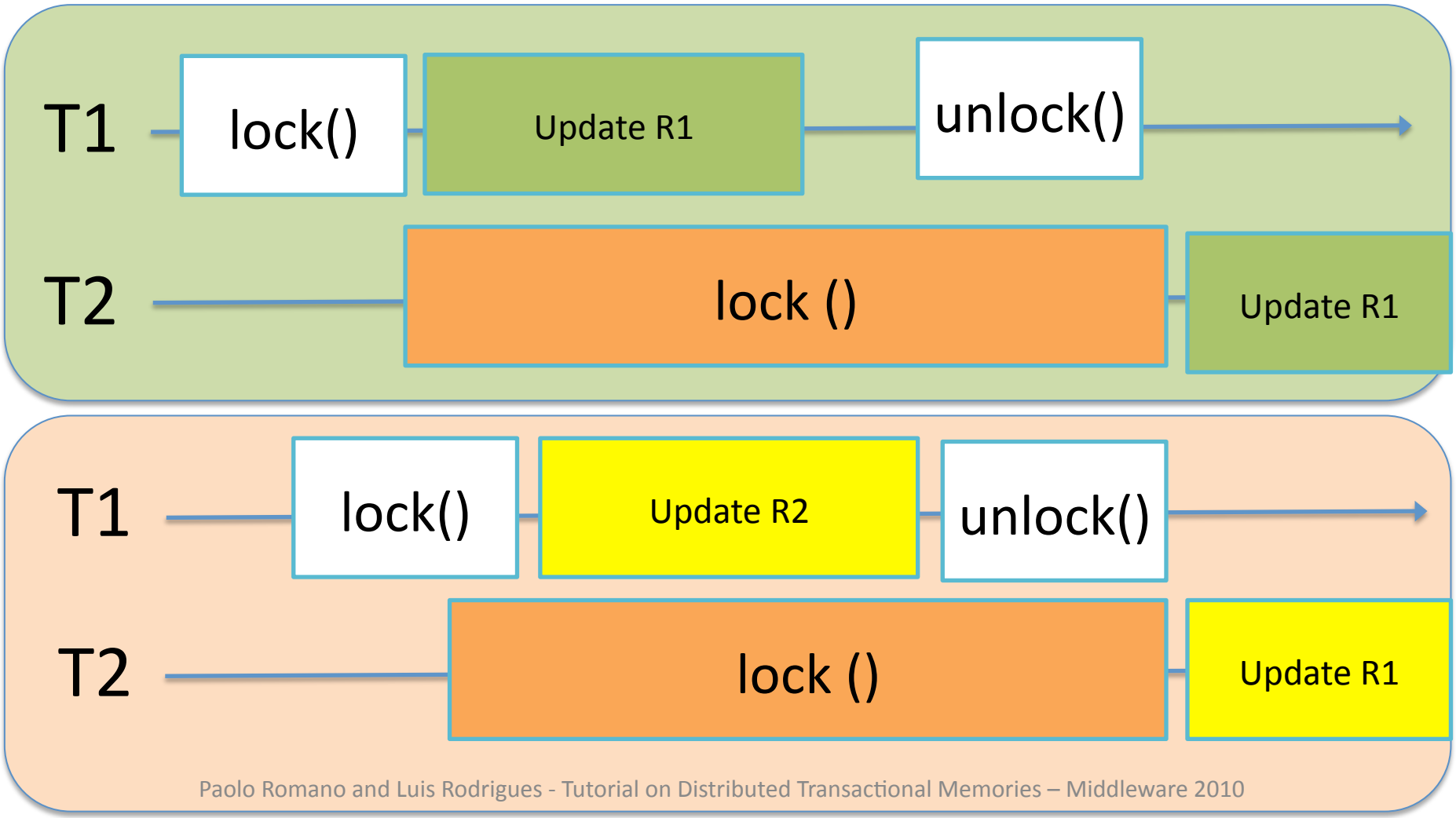
- Even with a single lock we can get deadlocks by introducing replication.
- This can be avoided if both replicas receive the lock requests in the same order.
- How can this be achieved?

Total order broadcast!

Total Order Broadcast

- Communication primitive that offers important properties.
- Reliable delivery:
  - If one replica R1 receives a message *m*, all correct replicas receive *m*.
- Total order:
  - If replica R1 receives *m1* before *m2*, any other replica Ri also receives *m1* before *m2*

Replicated transaction - single lock: total order!



Drawback of previous approach

- Coordination among replicas needs to be executed at every lock operation.
- Total order is an expensive primitive.
- The system becomes too slow.

Solution: limit the coordination among replicas to a single phase, at init or commit time.

Single coordination phase schemes

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

Single coordination phase schemes

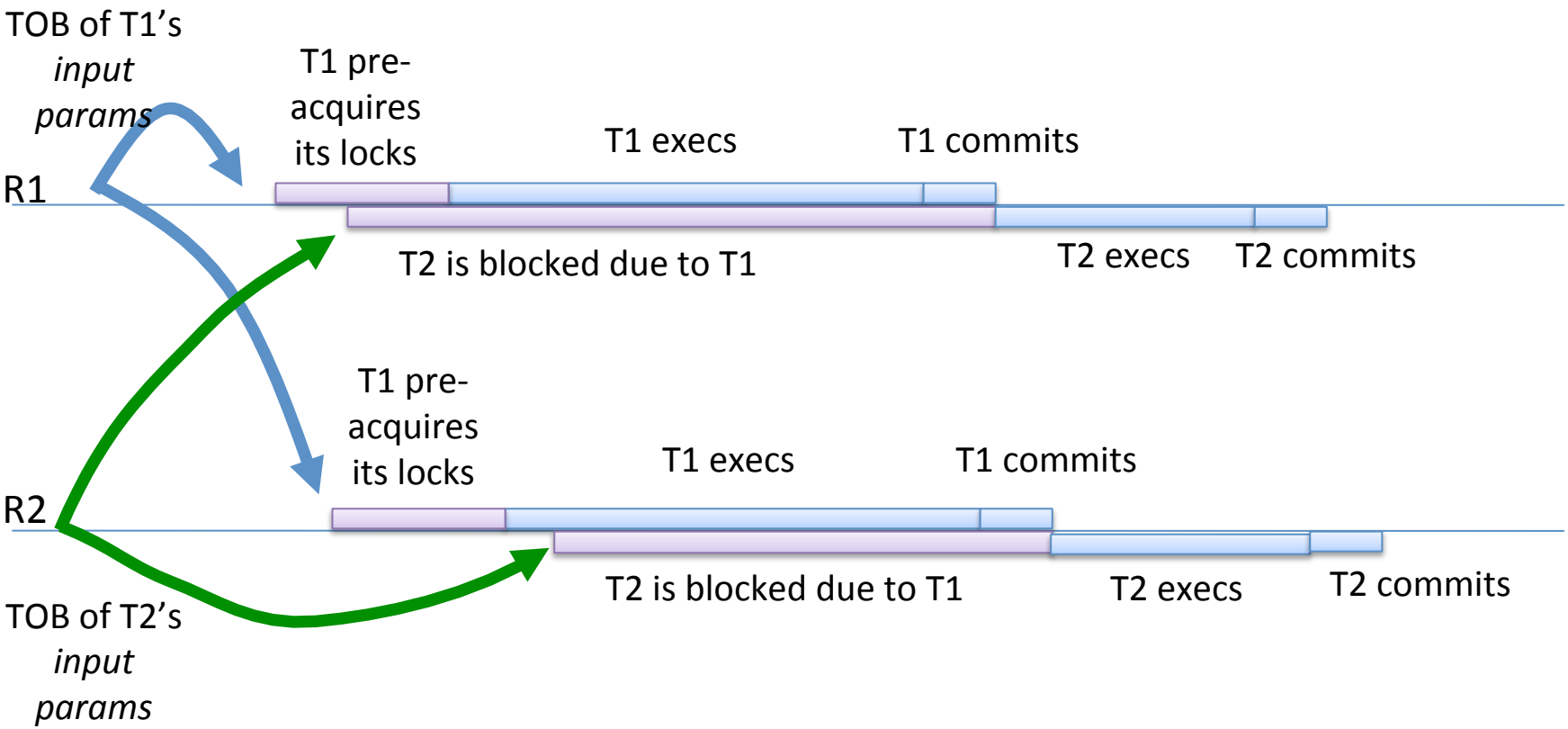
- State machine replication
- Single master (primary-backup)
- Multiple master (certification)

State machine replication

- All replicas execute the same set of transactions, in the same order.
- Transactions are shipped to all replicas using total order broadcast.
- Replicas receive transactions in the same order.
- Replicas execute transaction by that order.

Transactions need to be deterministic!

State Machine replication



+ **avoids transmitting readset& writeset**
+ **transactions never abort, good at high conflict rate**
- **requires deterministic execution of transactions**
- **all replicas process the whole transaction**
- **requires a priori knowledge of tx' read-sets and writeset**

Single coordination phase schemes

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

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.

Primary-backup

- Read transactions may be executed on backup replicas.
- Works fine for workloads with very few update transactions.
  - Otherwise the primary becomes a bottleneck.

Single master: synchronous

- Updates are propagated during the commit phase:
  - Data is replicated immediately
  - Read transactions observe up-to-date data
  - Commit must wait for reliable broadcast to terminate

Single master: asynchronous

- The propagation of updates happens in 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

Single coordination phase schemes

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

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

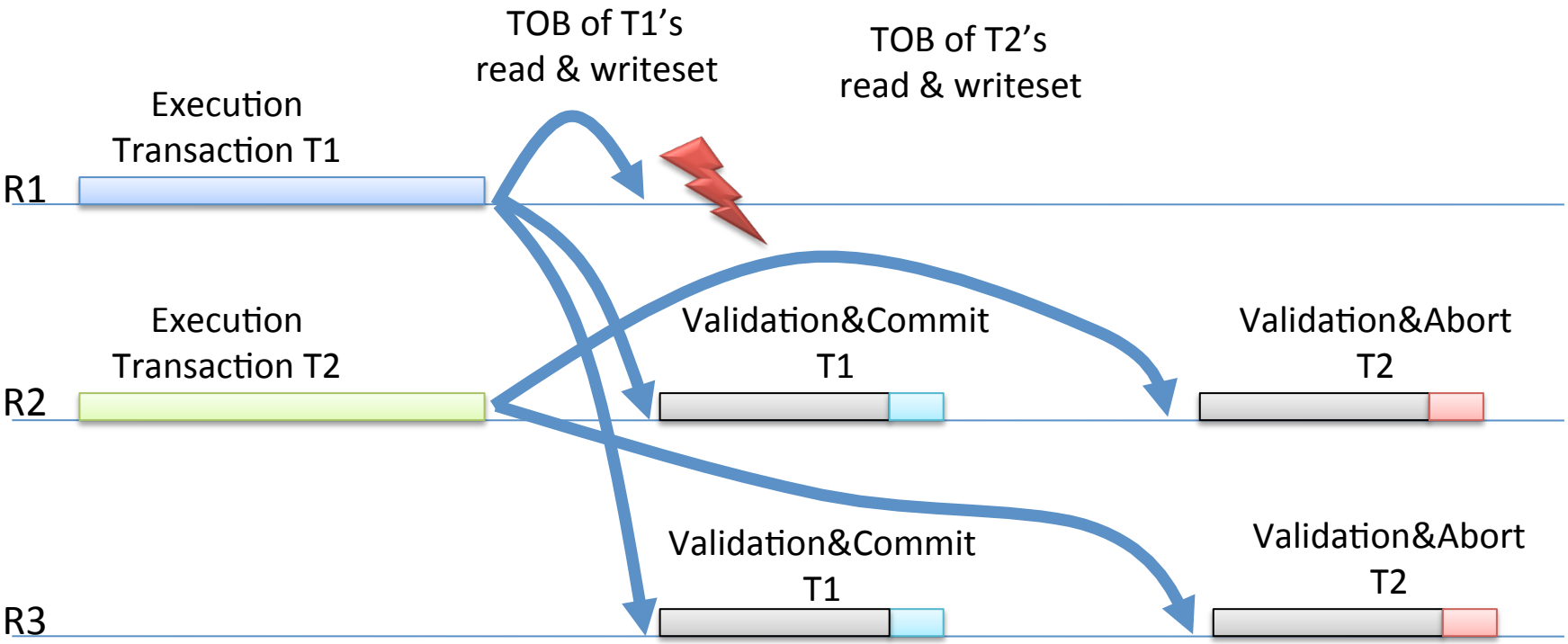
Multi-master

- 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

Non-voting algorithm

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

A Conventional TOB-based Replication Scheme
"Non-voting Certification Protocol"



- No communication overhead during transaction execution:
  - one TOB per transaction
- No distributed deadlocks

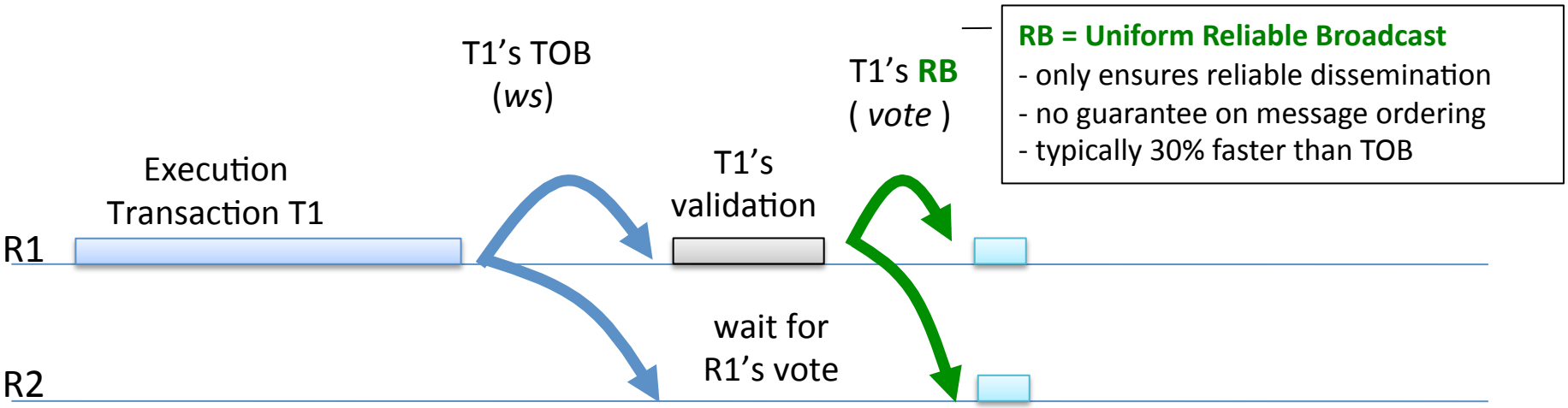
Voting algorithm

- The transaction executes locally at replica R
- When the transaction is ready to commit, **only** the write set is sent to all replicas using total order broadcast
- Transactions' commit requests are processed in total order

Voting algorithm

- 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 Certification scheme



+ **avoids transmitting readset**
- **incurs in an additional broadcast**
- **RB is in the critical path:**
  - **no new transactions are processed as long as the vote is not delivered**
  - **1/Turb is an upper bound on the commit rate, i.e. on throughput**

Summary of algorithms

- State-machine replication.
- Primary-backup:
  - Synchronous
  - Asynchronous
- Multi-master:
  - Non-voting certification
  - Voting certification

Communication and coordination primitives

- Previous protocols rely heavily on several communication and coordination primitives:
  - Total order reliable broadcast
  - Reliable broadcast
  - Atomic commit
- What is the cost of these primitives?
- Where can I get implementations?

Group communication systems

Group Communication System

- Combines, in a integrated form, two fundamental services:
  - Membership service
  - Multicast service with different flavors:
    - Reliable multicast
    - Total order multicast (atomic multicast)
    - Optimistic delivery

Group Communication System

- Membership service
  - Provides information about which members are active and which members have failed.
- Multicast service
  - Provides support for reliable multicast (informally, all group members receive a given multicast message or none does).

Why it is useful for transactional replication

- Provides information about active replicas:
  - The set of replicas is dynamic
  - Replicas may fail: the membership detects the failure and notifies the remaining replicas
  - New replicas may join: old replicas are notified that new replicas exist

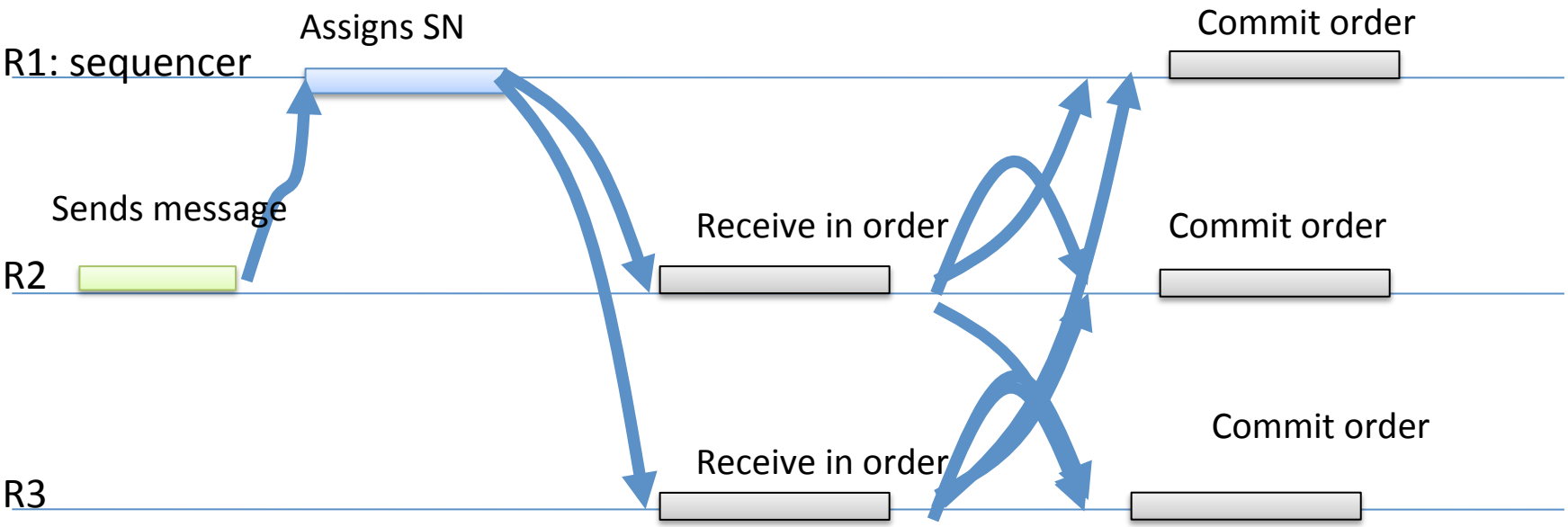
Why it is useful for transactional replication

- Support the communication among replicas
  - Disseminate updates
  - State-transfer to new replicas
- Support the coordination among replicas
  - Order conflicting concurrent transactions
  - Consistent decision on the transactions outcome (commit/abort)

Total Order Broadcast – how expensive?

protocol	resilience	# comm. steps	# msgs.	# forced writes
TOB (i)	Blocking	2	n+1	n
Two Phase Commit	Blocking	3	3n	n
Uniform TOB(ii)	Non-blocking	3/4	4n	n
Three phase commit	Non-blocking	4/5	5n	n

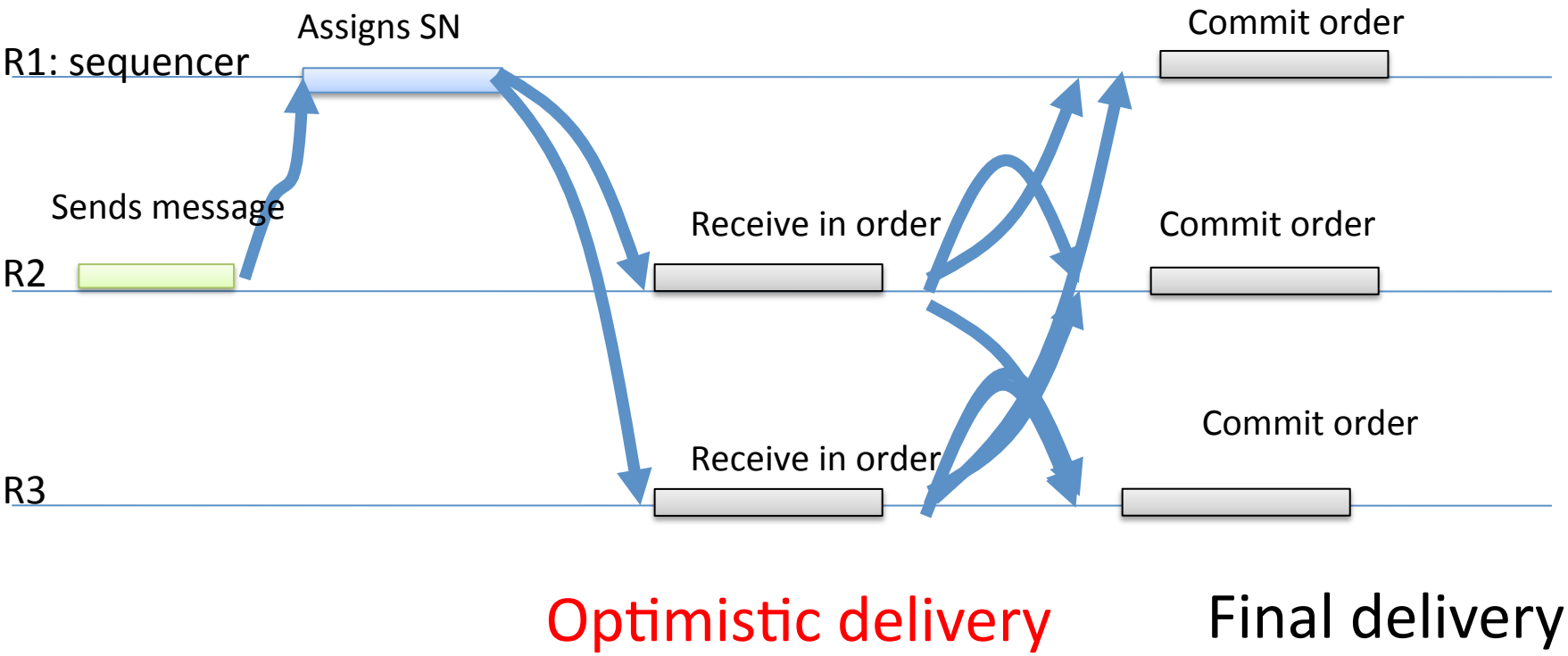
Sequencer based total order



Can we make it faster?

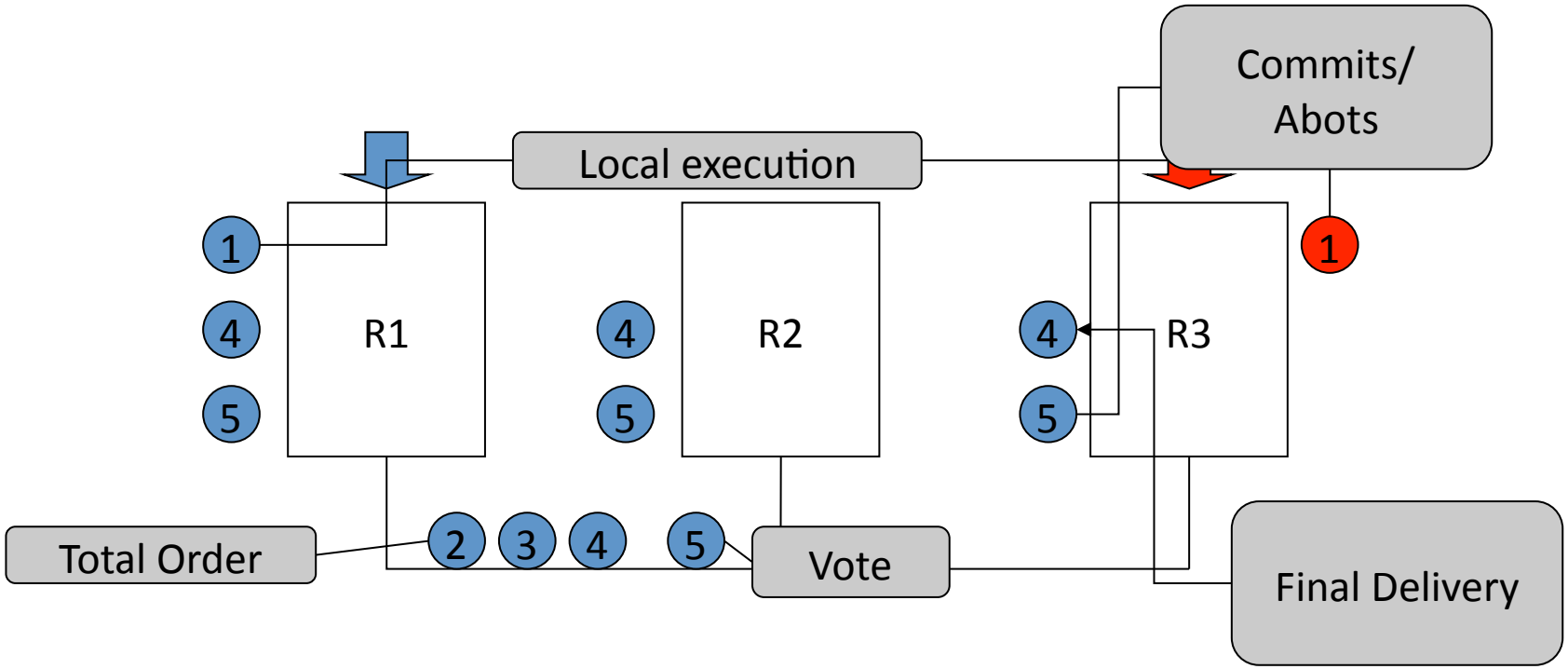
- Total order with optimistic delivery.
- Unless the sequencer node crashes, final uniform total order is the same as regular total order.
- Application may start certificating the transaction locally based on optimistic total order delivery.

Sequencer based total order



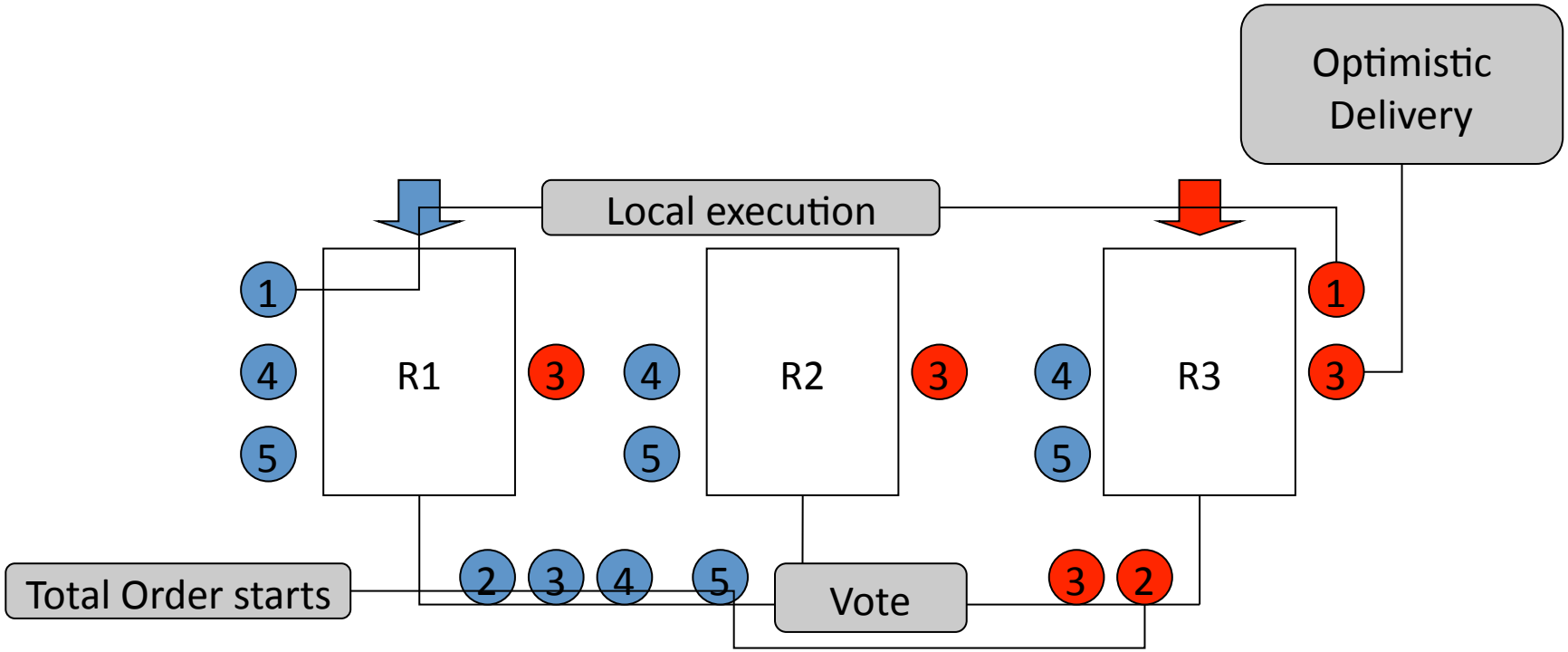
Uniform vs optimistic delivery

- Can save one communication step.



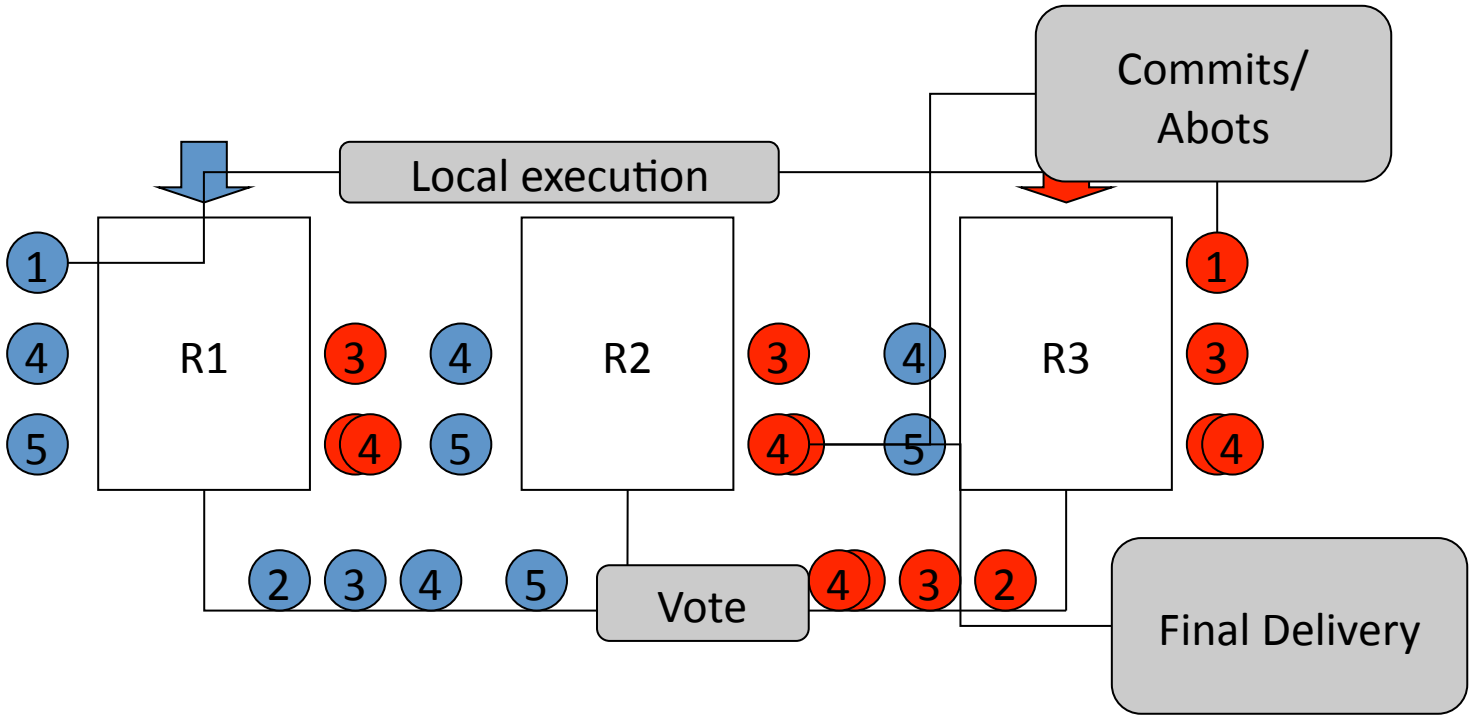
Uniform vs optimistic delivery

- Can save one communication step.



Uniform vs optimistic delivery

- Can save one communication step.



ToC

PART I

- Non-distributed Transactional Memories
  - Concepts (45 min)
  - Systems (45 min)

(break)

PART II

- Distributed Transactional Memories
  - Concepts (45 min)
  - **Systems (45 min)**

Roadmap

- DTMs: Algorithms, platforms
  - *Sinfonia*
  - *Cluster-STM*
  - *D²STM*
  - *ALC*
  - *Speculative Replication of STMs*

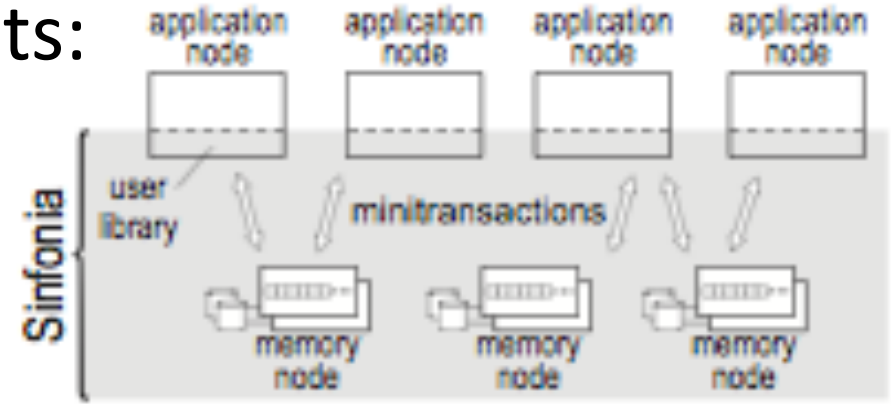
Sinfonia [AMSVK09]

Sinfonia

- Word-based, PGAS
  - linear address space
- Mini-transactions:
  - static: a-priori known data to be accessed
  - allows optimizing communication pattern
- Fault-tolerance via:
  - in-memory replication
  - sync/async persistency

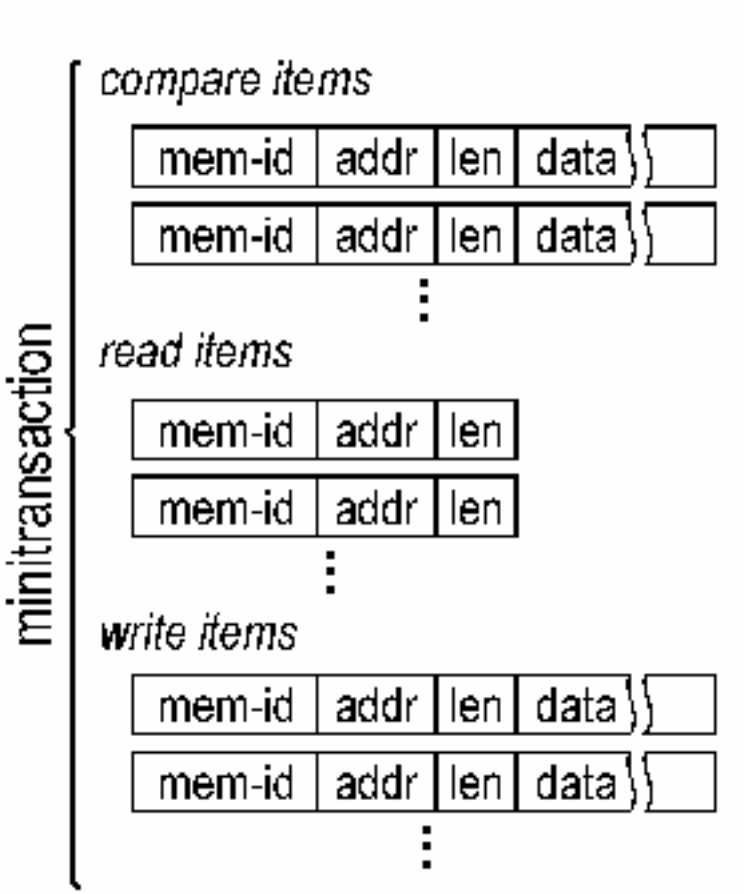
Sinfonia architecture

- Two logical components:
  - application node
  - memory node



- Memory nodes configurable to achieve:
  - synchronous redo logging to disk
  - asynchronous backup image to disk

Minitransactions



semantics

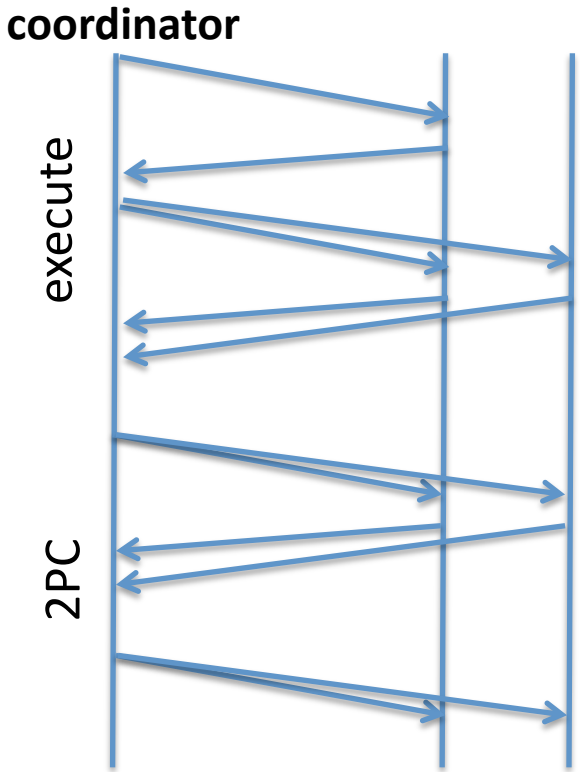
- if all compare items match:
  - retrieve read items
  - modify write items

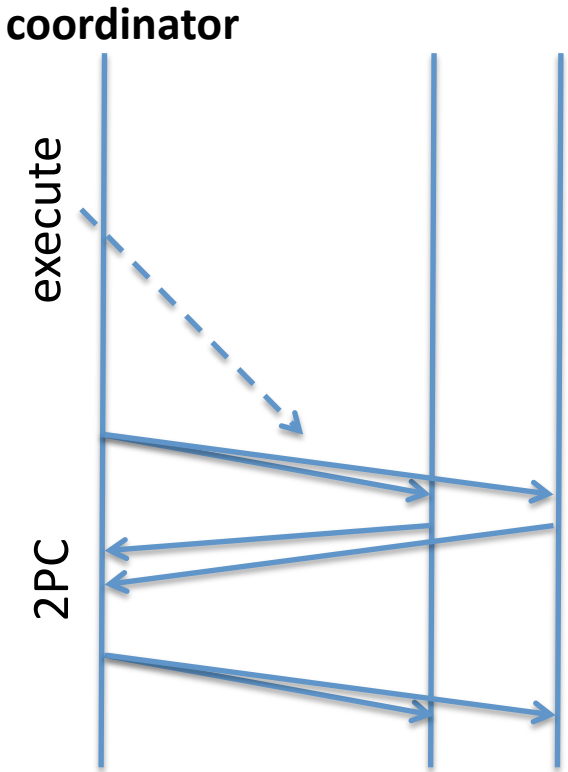
example

```
t=new Minitransaction();
t->cmp(hostX, addrX, len, 1);
t->write(hostY,addrY,len,2);
status=t->exec_and_commit();
```

Sinfonia's execution

- The assumption of static transactions allows piggybacking execution onto 2PC



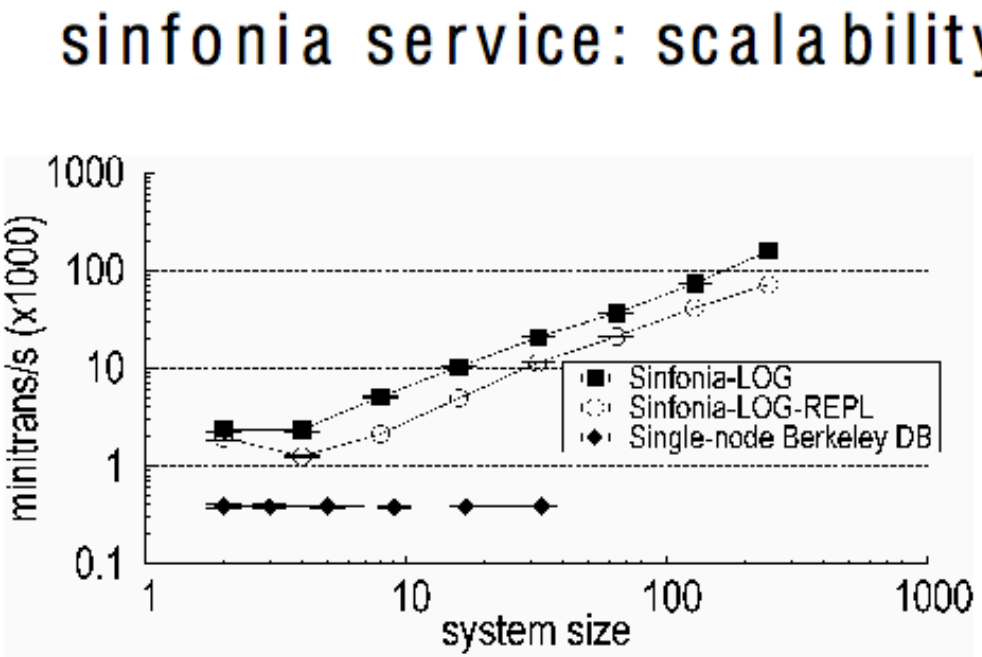


Caching & Replication

- 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

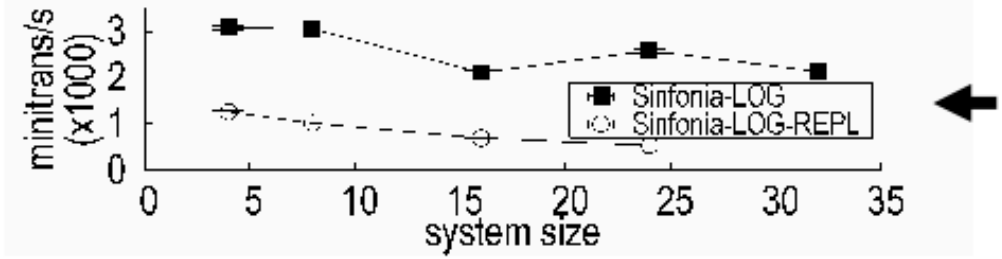
Evaluation – synth. benchmark



minitransaction spread: number of memory nodes in a minitransaction

minitransaction spread=2:

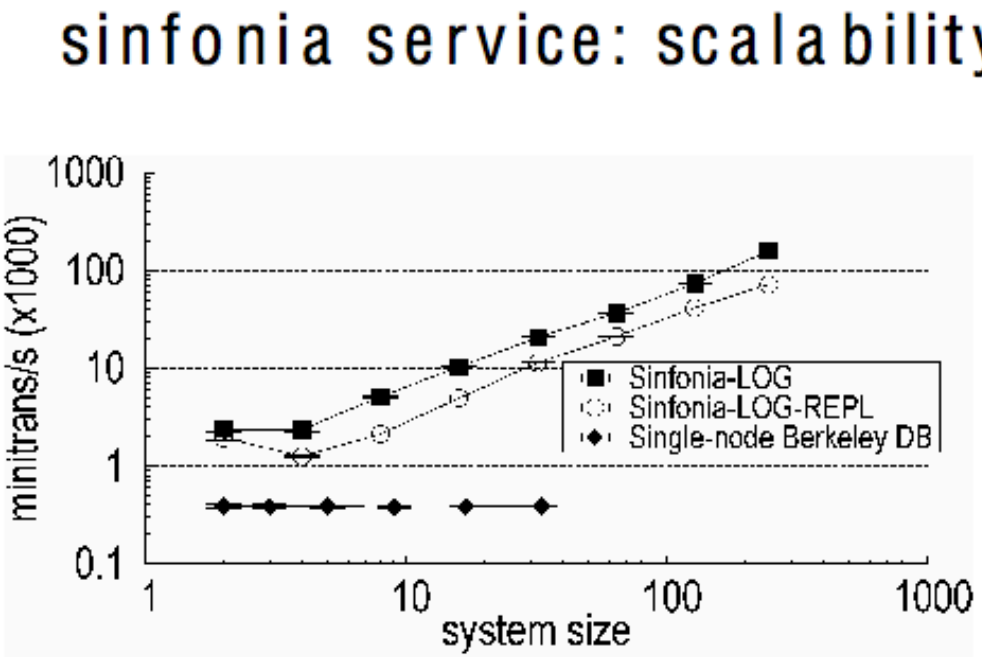
usually within 85% of ideal scalability



minitransaction spread=all memory nodes (increases with system size):

no scalability

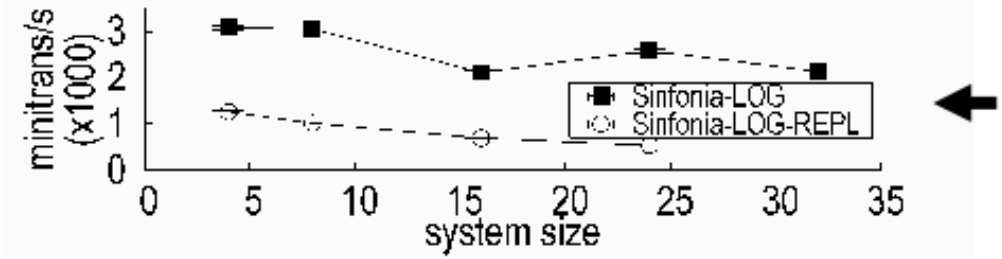
Evaluation – synth. benchmark



minitransaction spread: number of memory nodes in a minitransaction

minitransaction spread=2:

usually within 85% of ideal scalability



minitransaction spread=all memory nodes (increases with system size):

no scalability

Cluster-STM [BAC08]

Cluster-STM

- word based, PGAS
  - language support for remote code execution
- no persistency, no replication, no caching
- supports only single thread per node
- unconstrained transactions
- various lock acquisition schemes + 2PC

Cluster-STM Interface

- Block data movement

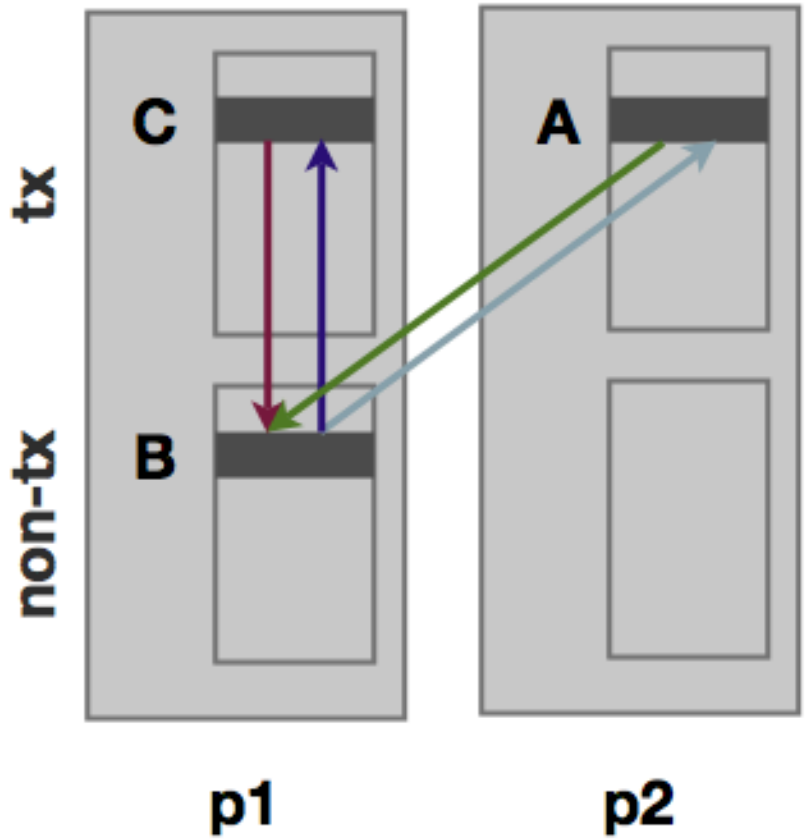
All ops occur on p1

stm_get(work_proc=p2, src=A, dest=B, size=n, ...)

stm_put(work_proc=p2, src=B, dest=A, size=n, ...)

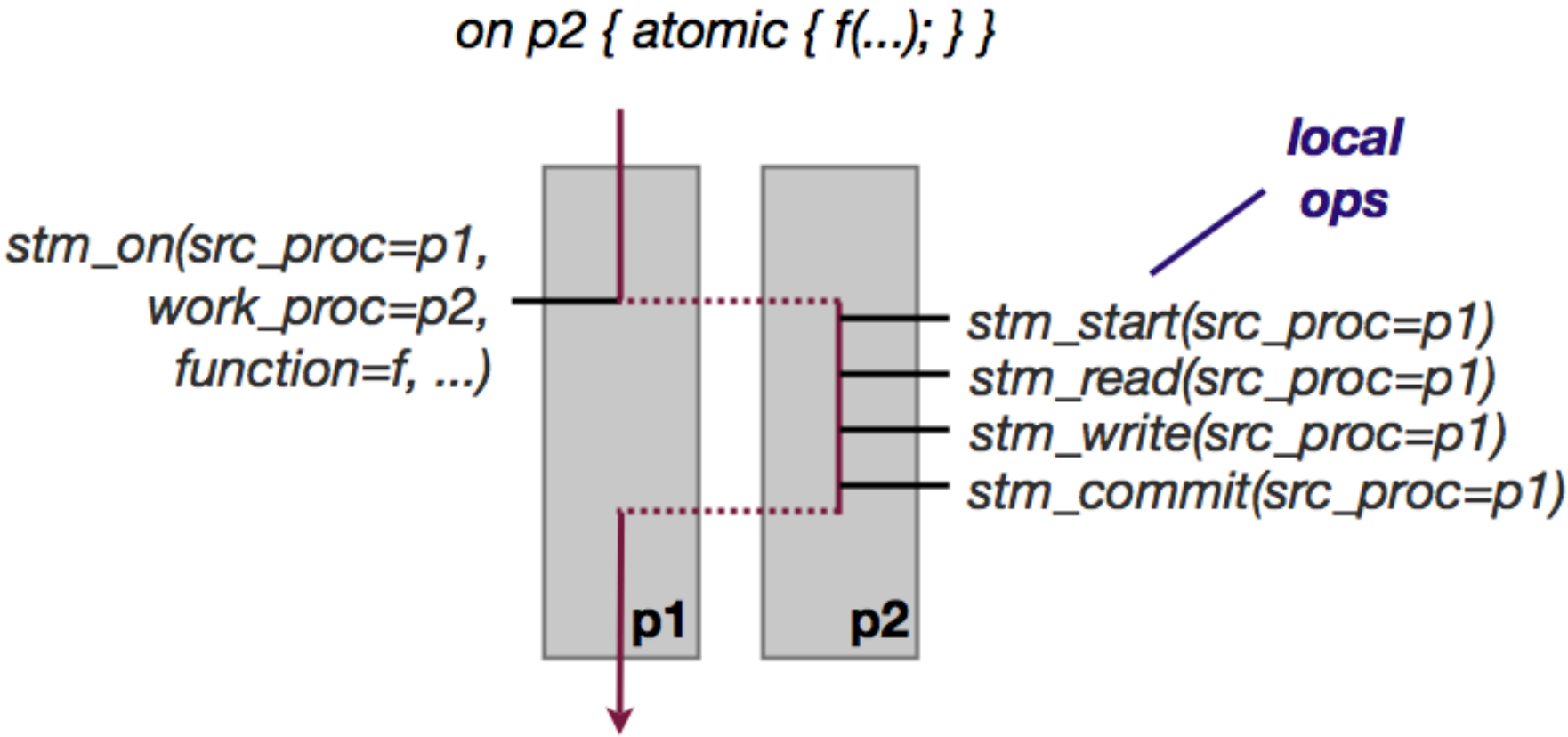
stm_read(src=C, dest=B, size=n, ...)

stm_write(src=B, dest=C, size=n, ...)



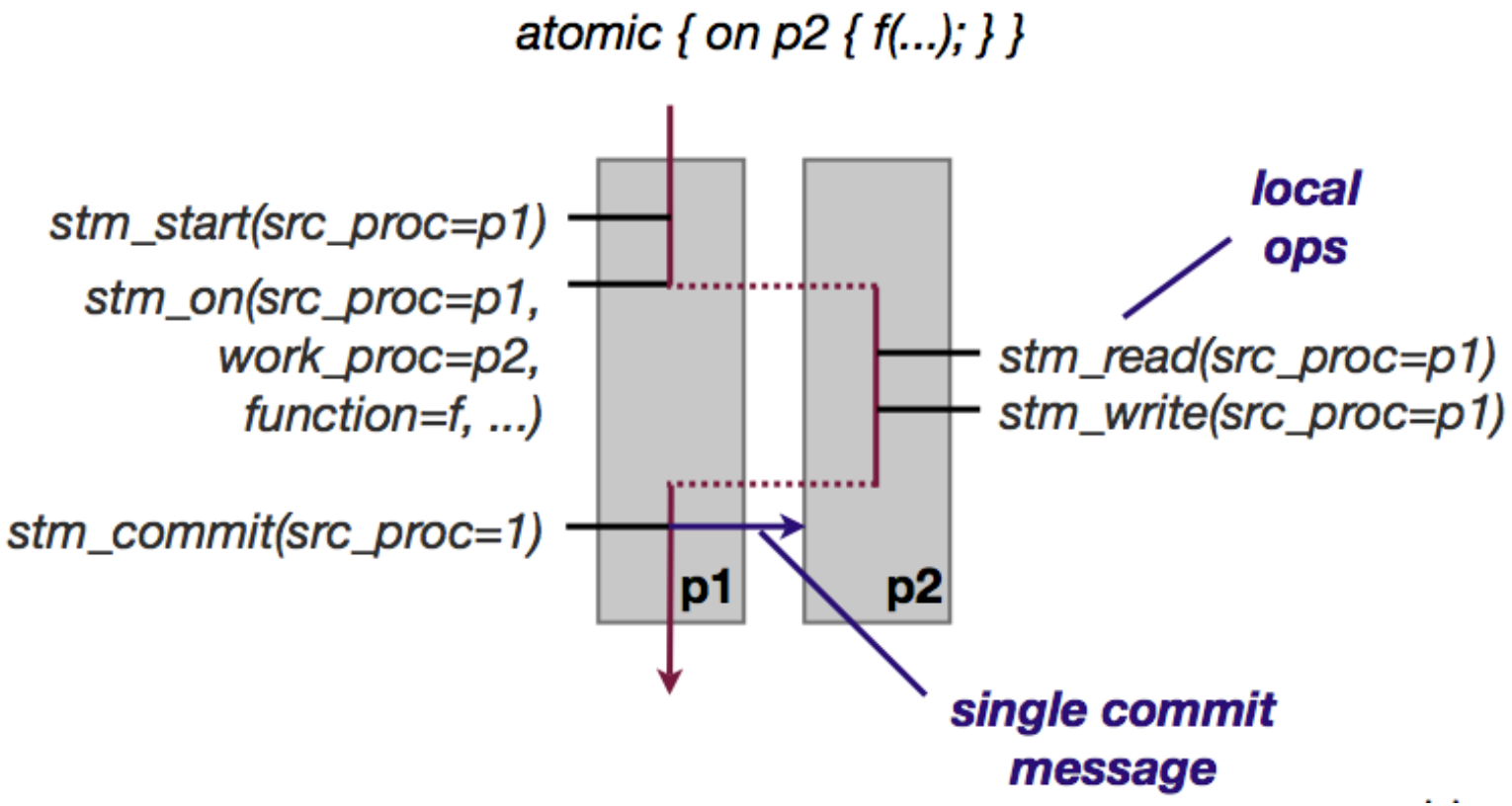
Cluster-STM Interface

- Exploiting data locality via remote execution:
  - transactions inside remote work



Cluster-STM Interface

- Exploiting data locality via remote execution:
  - remote work invoked from within a transaction



Cluster-STM Algorithm

- No caching, no replication

- Evaluated 4 design choices
  - Conflict Detection Unit (CDU) size
  - Read locks (RL) vs. read validation (RV)
  - Undo log (UL) vs write buffer (WB)
  - Early acquire (EA) vs late acquire (LA)

CDU size

- Granularity of conflict detection affects:
  - possibility of false sharing
    - abort rate increase
  - size of metadata:
    - memory footprint

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
- RV:
  - commit time validation (not opaque)
  - validity check requires 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

Undo Log vs Write Buffer

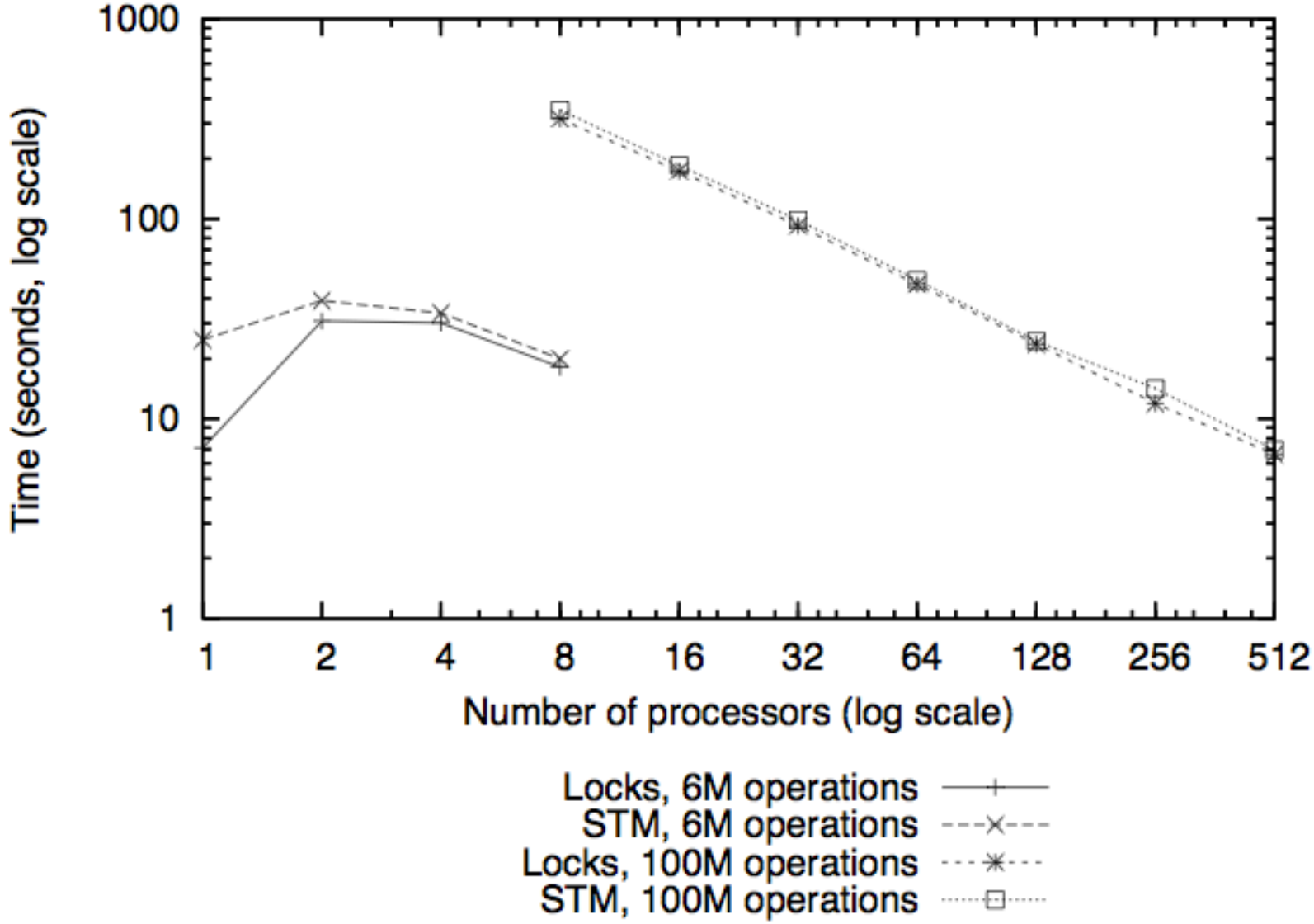
- Undo log:
  - updates are applied in-place during xact's execution
    - reduces the commit cost
    - require write lock acquisition
- Write Buffer
  - updates applied to private copy, written back during commit phase
    - reduces the abort cost

- Choice has a reduced impact in DSTM, where communication costs are largely predominant

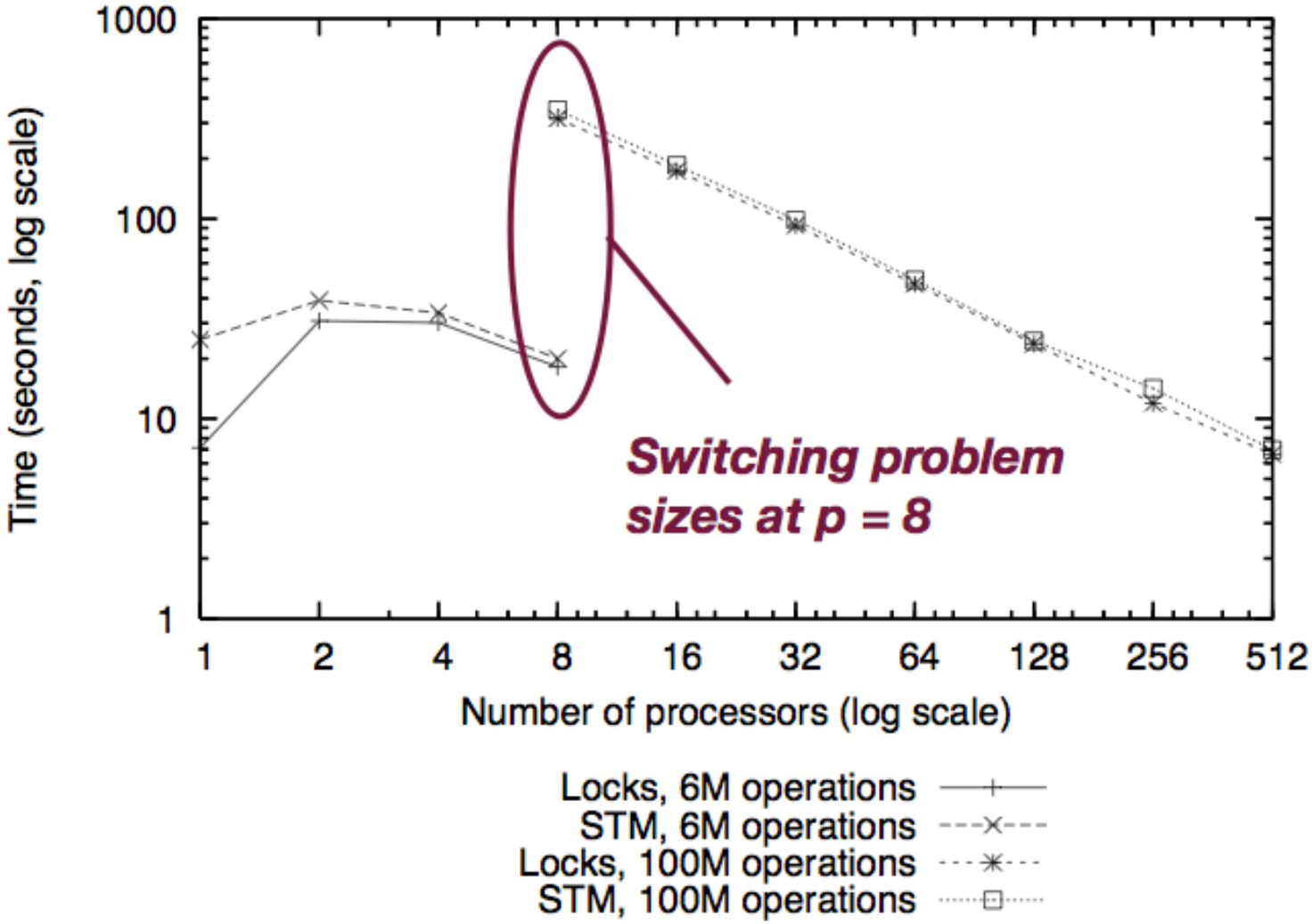
Early Acquire vs Lazy Acquire

- In case Write Buffering is used, write locks can:
  - be acquired at commit time:
    - may allow for more concurrency
    - avoid communications for writes during exec phase
    - requires **additional communication** at commit time
  - as the write is issued:
    - may avoid wasted work by doomed transactions
    - forced sync upon each write

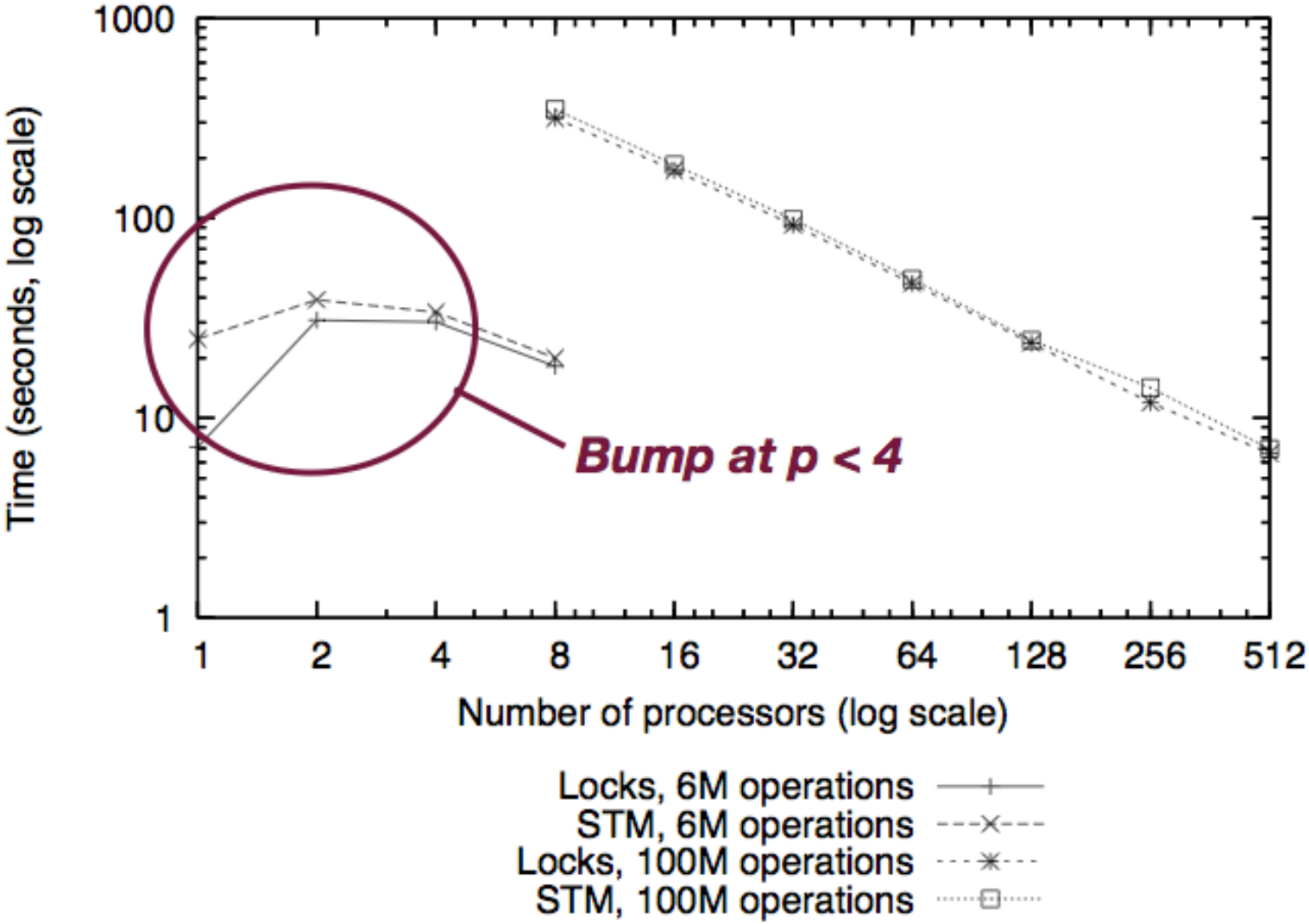
Evaluation (micro-benchmark)



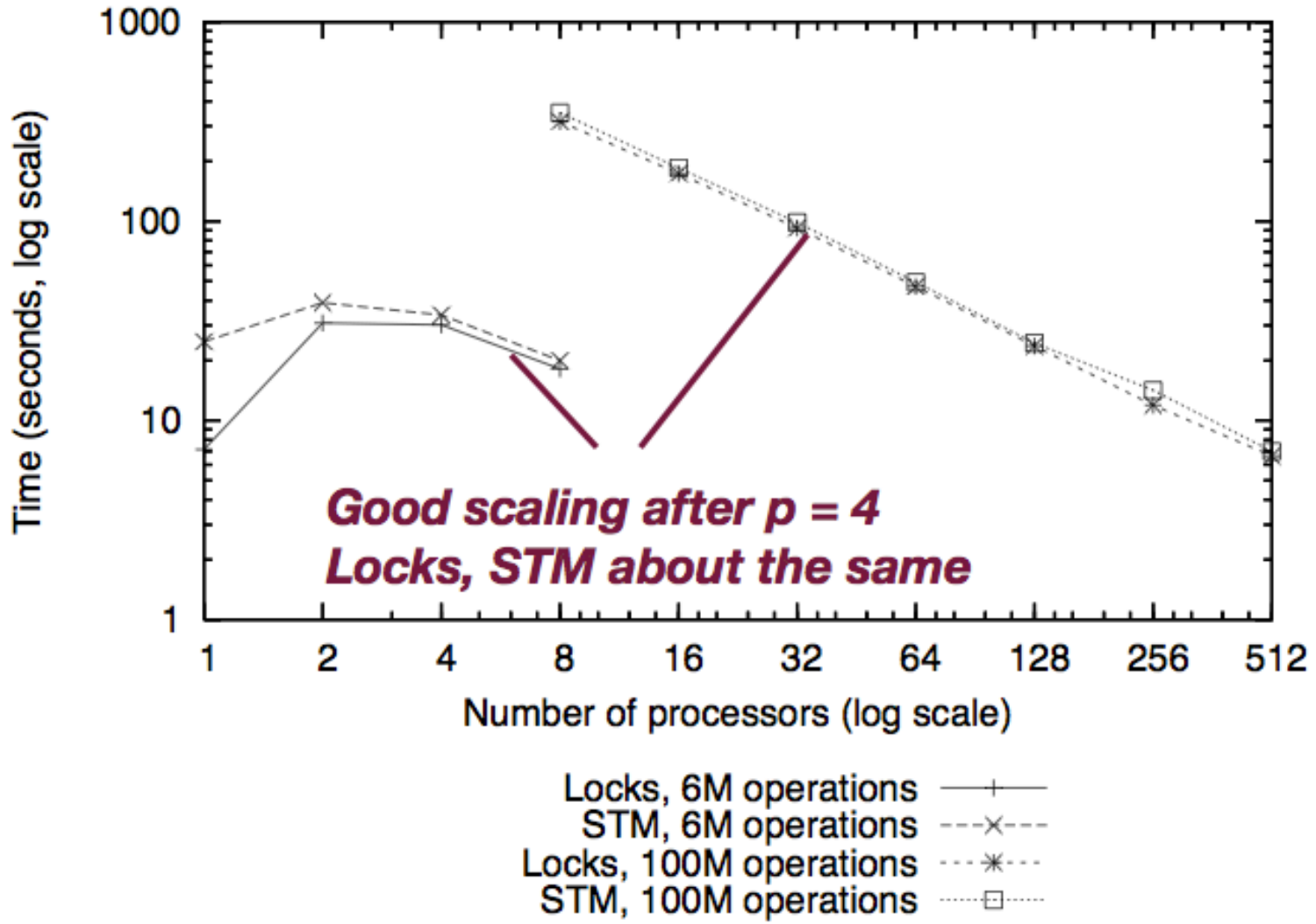
Evaluation (micro-benchmark)



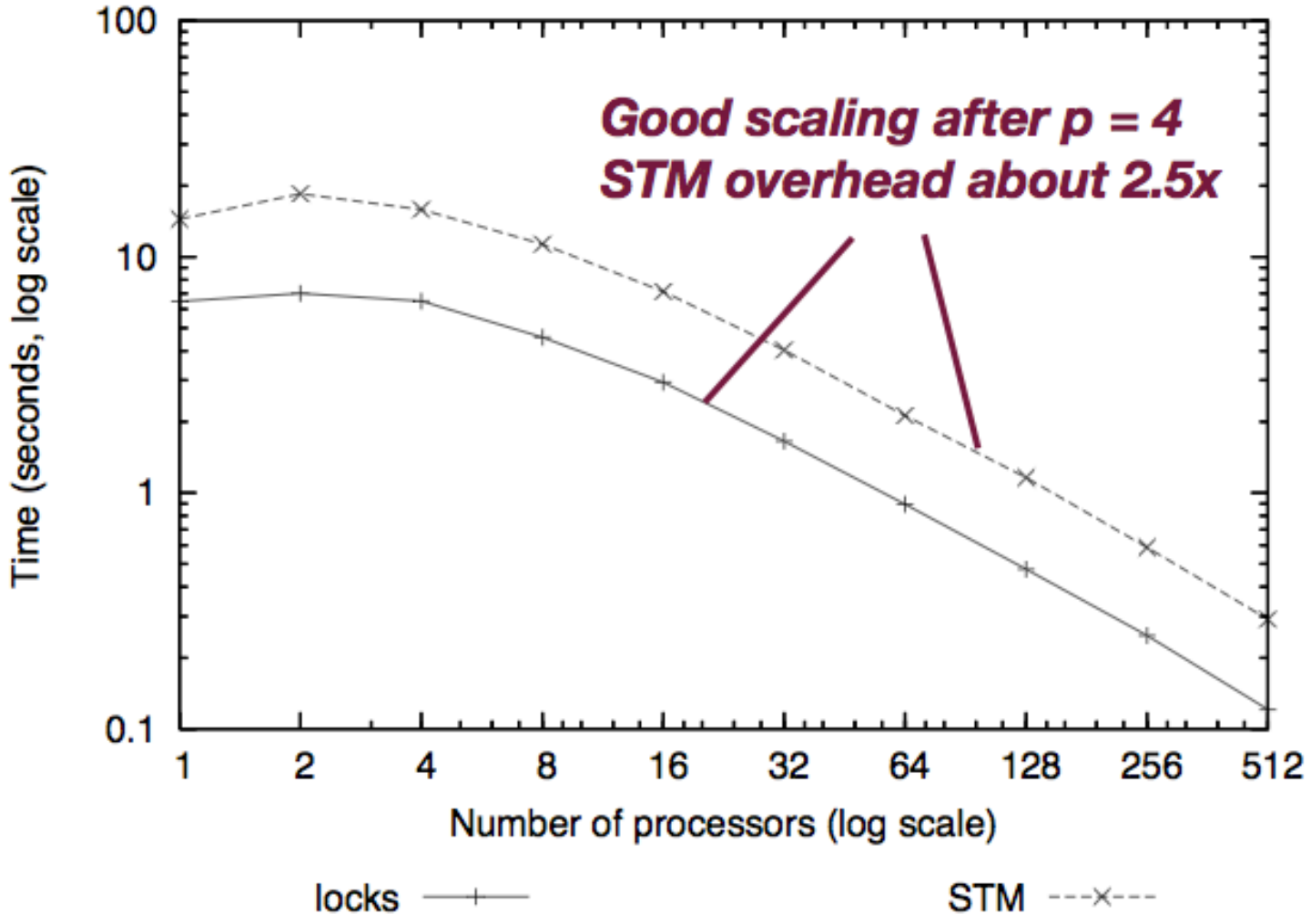
Evaluation (micro-benchmark)



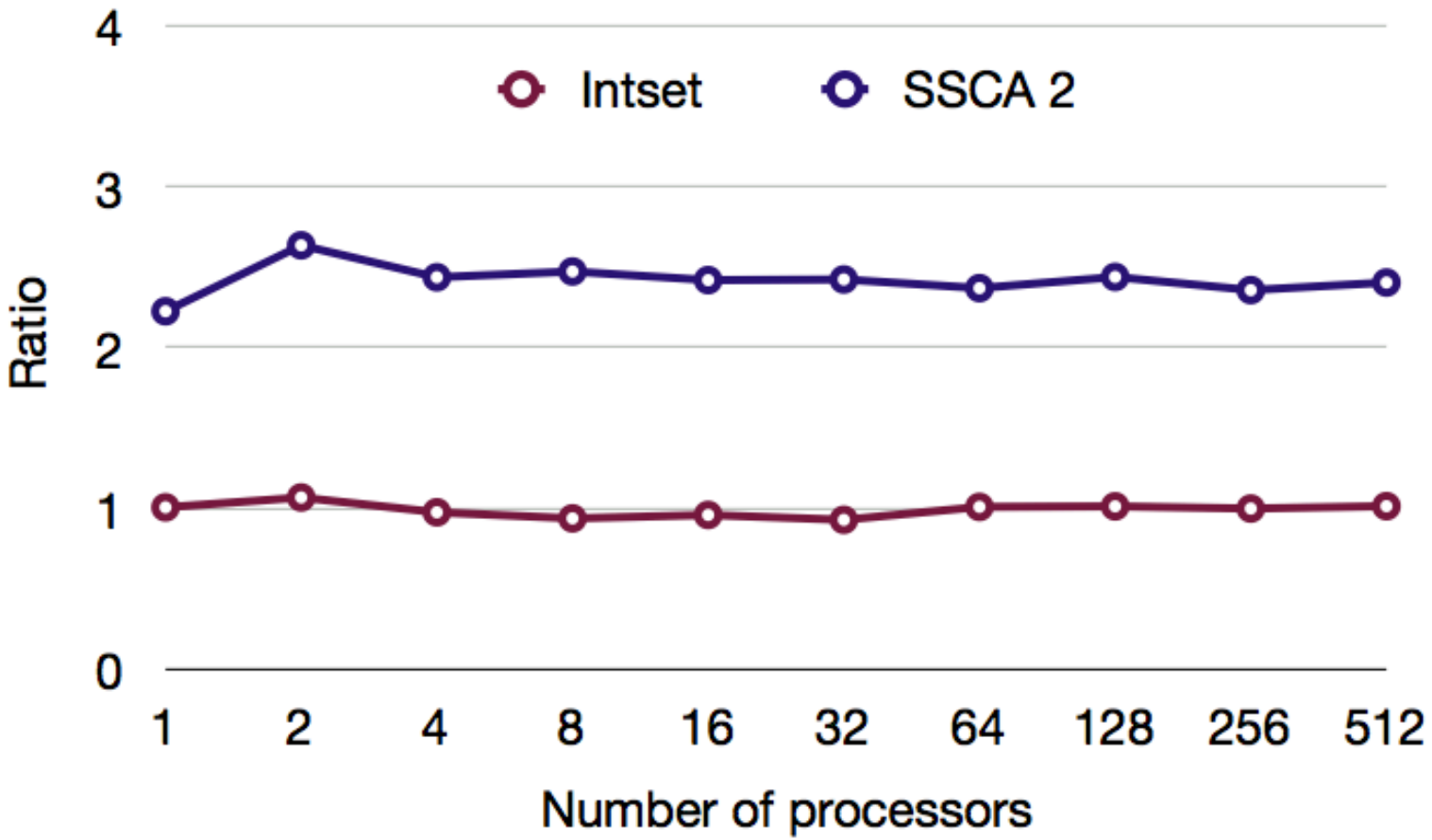
Evaluation (micro-benchmark)



Evaluation
Graph Analysis Benchmark (SSCA2)



Ratio RV to RL execution time

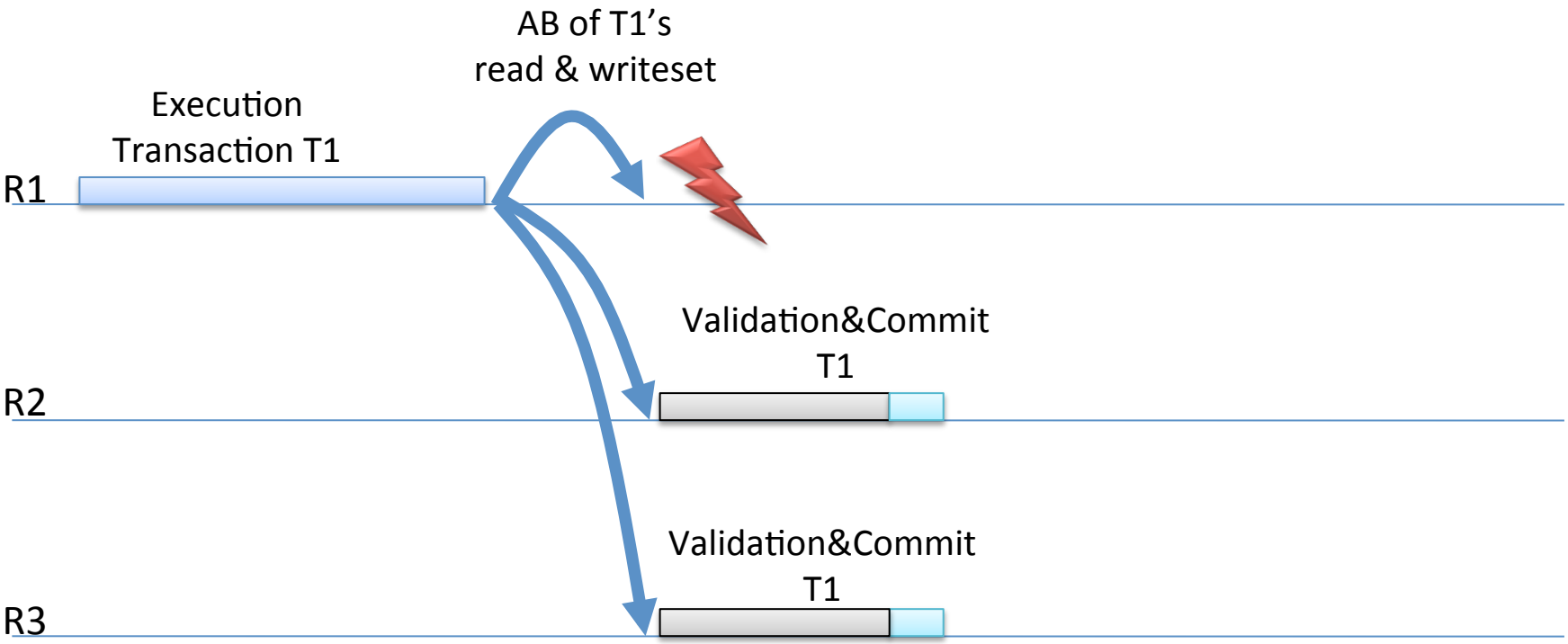


Dependable Distributed STM - D²STM [CRRC09]

D^2STM - overview

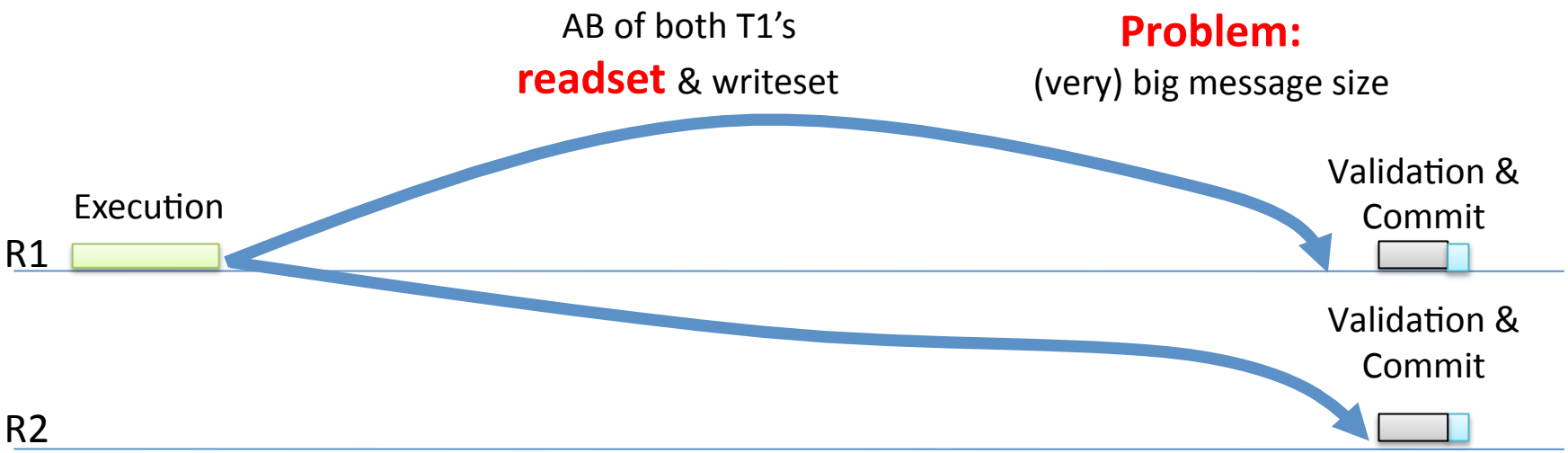
- object oriented
- layered upon a non-distributed STM, JVSTM:
  - single system image
  - full, strongly consistent replication
  - unaltered programming model
- Atomic Broadcast based replication scheme:
  - Bloom-filter Based Certification

A Conventional AB-based Replication Scheme
"Non-voting Certification Protocol"



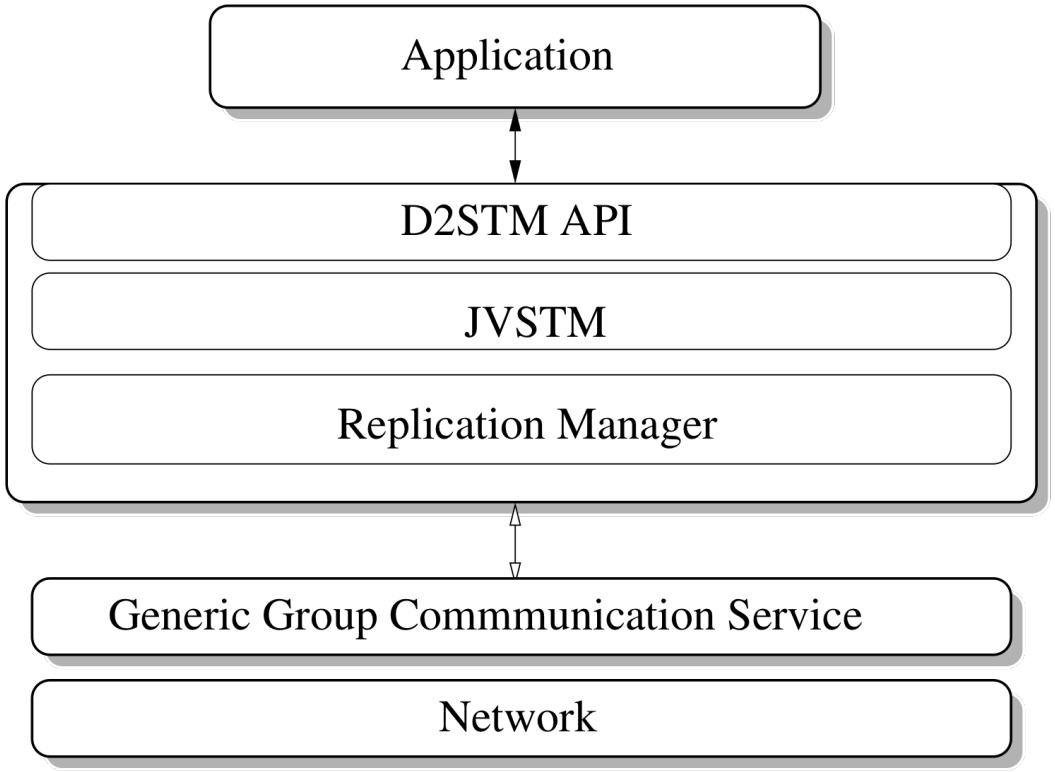
- No communication overhead during xact execution:
  - one AB per xact
- No distributed deadlocks

How it actually looks like in a STM context



- Certification schemes requiring a single AB need to broadcast tx's readset:
  - required by remote nodes to validate the transaction
- In STMs, transaction's exec 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

Software Architecture



Bloom Filters

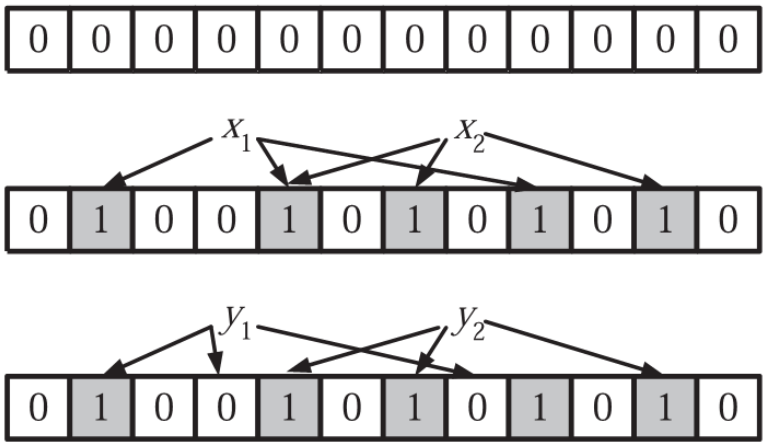


Figure: Bloom filter

- 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

Problem:

- 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*)

Bloom Filter Certification

- Read-only transactions: local execution and commit
- Write transaction T:
  1. Local validation (read set)
  2. If the transaction is not locally aborted, the read set is encoded in a Bloom filter
  3. Atomic broadcast of a message containing:
     1. the Bloom filter enconding of tx readset
     2. the tx write set
     3. the snapshotID of the tx
  4. Upon message delivery: validate tx using Bloom filter's information

Bloom Filter Certification

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

BFC – two key problems

1. BFC requires information on the writeset of committed transactions.

   How to garbage collect them?

2. A false positive in the BF generate an abort.

   How to tune the BF's size to bound the additional abort rate?

Garbage Collection of Tx's writesets

- each process maintains an array *A* storing in pos. i the snapshotID of the oldest tx running on node I
- committing tx piggybacks on its AB the snapshotID of the oldest locally running tx
- let *minSnap* be the minimum snapshotID in A
- the writesets of any Tx with snapshotID < *minSnap* can be safely garbage collected

False Positives

False positives do not lead to inconsistencies in the replicas state, but to an increase in the transactions abort rate

Computing the size of a Bloom filter:

$$m = \left\lceil -n \frac{log_2(1-(1-maxAbortRate)^{\frac{1}{q}})}{ln\ 2} \right\rceil$$

- *q*: number of BF queries, estimated via moving average over recently committed transactions
- *n*: Number of items in the read set (known)
- *maxAbortRate*: User-defined false positive rate (user param)

STMBench7: Results

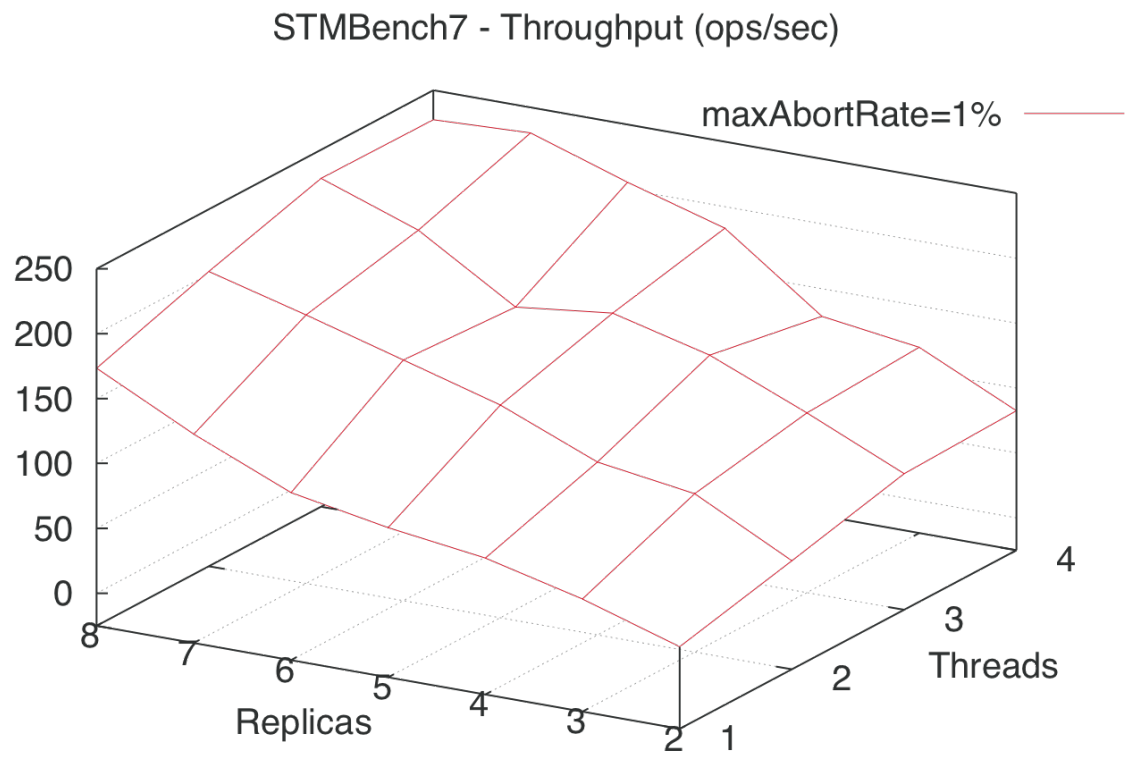


Figure: Throughput

STMBench7: Results

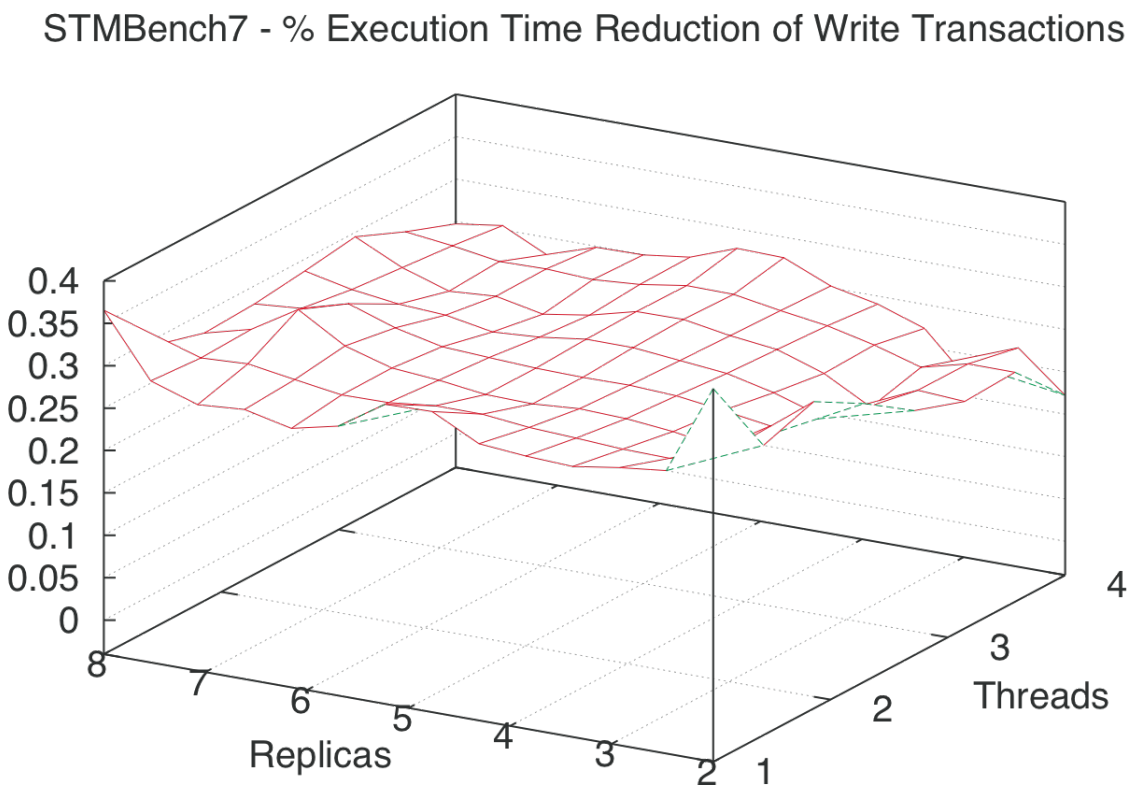


Figure: Execution time reduction (%)

Asynchronous Lease Certification (ALC) [CRR10]

Key intuition

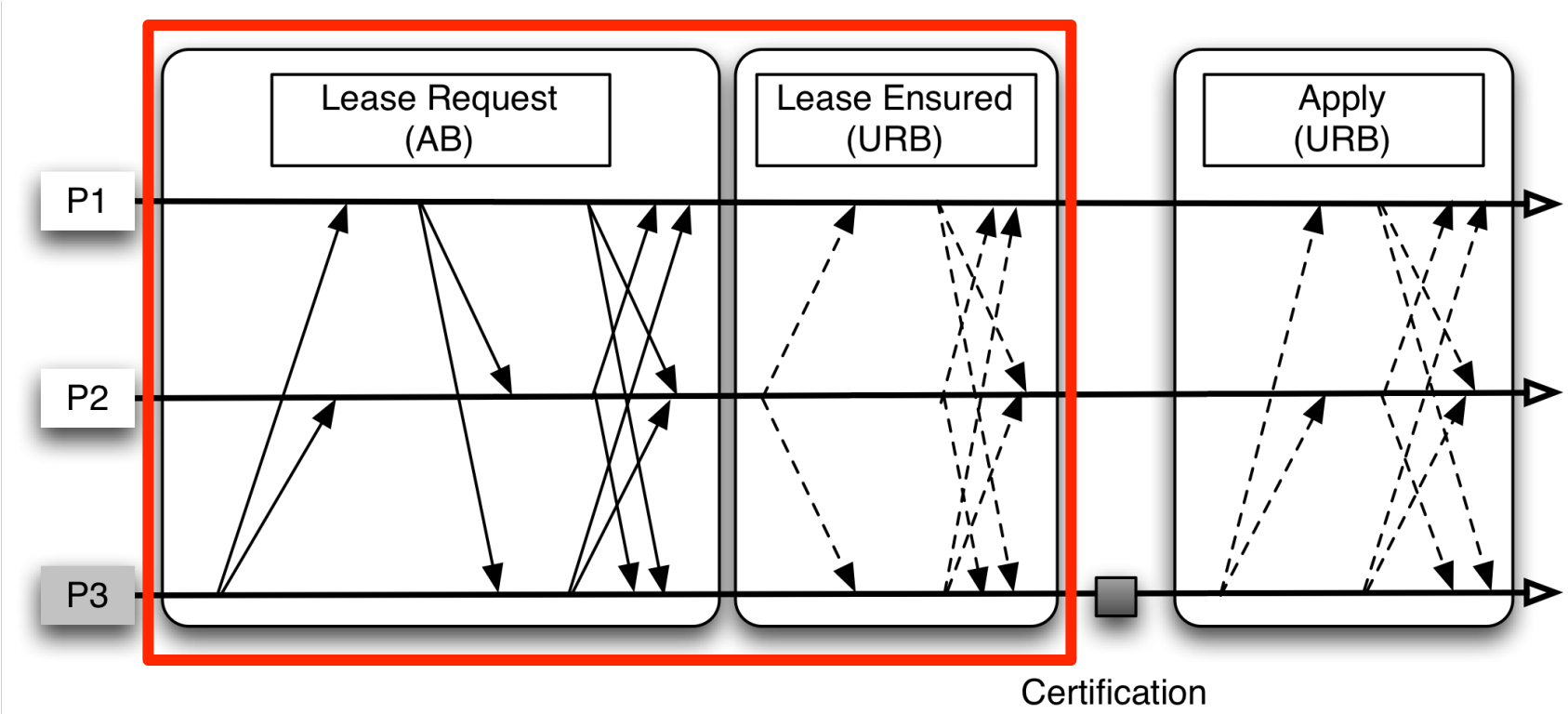
- Exploit data access locality by letting replicas dynamically establish *ownership* of memory regions:
  - replace AB with faster coordination primitives:
    - no need to establish serialization order among non-conflicting transactions
  - shelter transactions from remote conflicts

- 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

Protocol's overview

- Transactions are locally processed
- At commit, replicas checks if a lease on the accessed data is already owned:
  - NO
    1. an Asynchronous Lease is established
    2. the transaction is locally validated
    3. if validation succeeds, its writeset is propagated using Uniform Reliable Broadcast (URB):
       - no ordering guarantee, 30-60% faster than AB
    4. if validation fails, upon re-execution the node holds the lease:
       - xact cannot be aborted due to a remote conflict!
  - YES
    - as above, but from point 2.

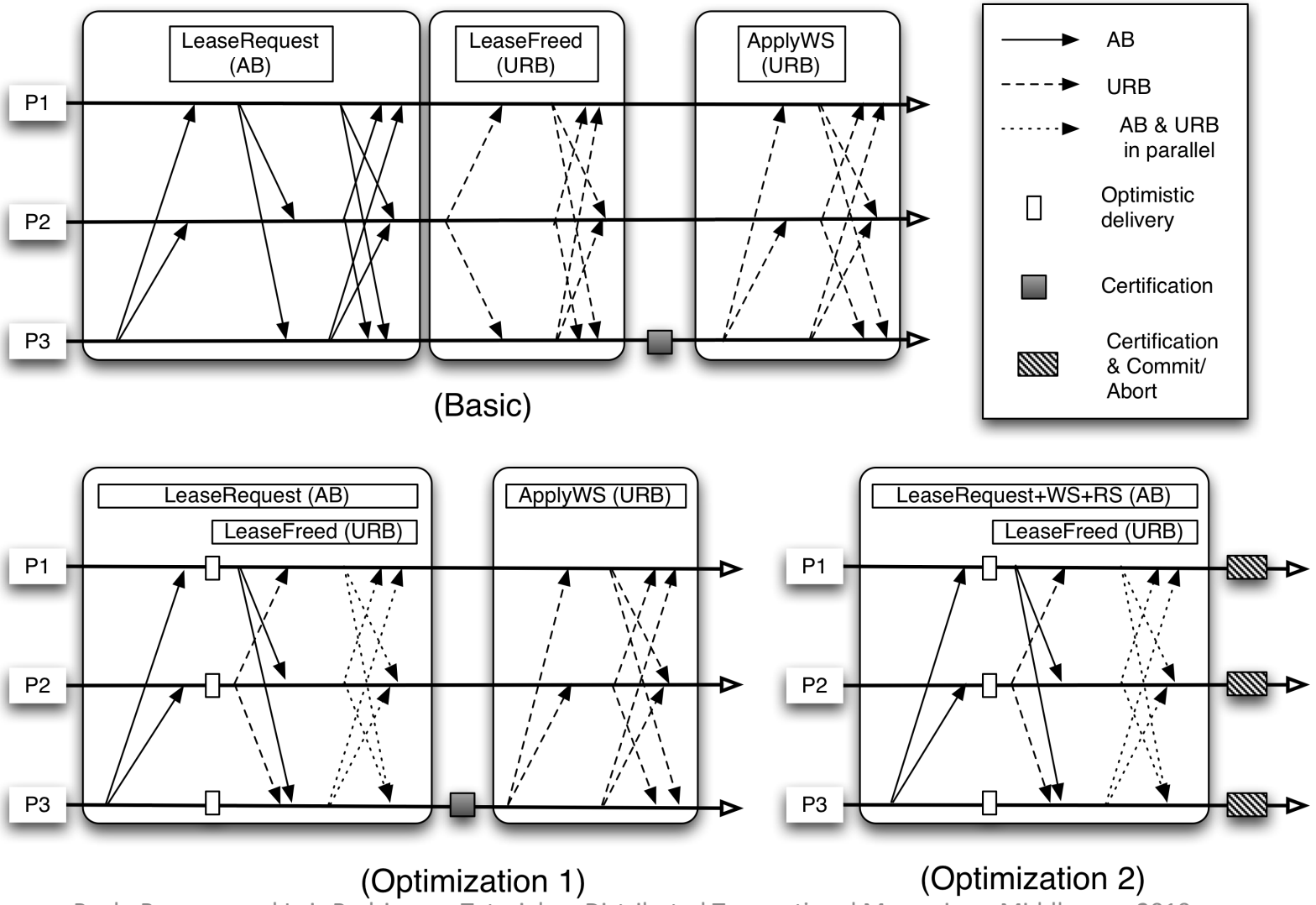
Asynchronous Lease Establishment Basic Protocol



Simple but sloppy:

If a node doesn't own a lease, it incurs in the latency of 1 AB + 2 URB to commit a xact

Asynchronous Lease Establishment Optimized Protocol



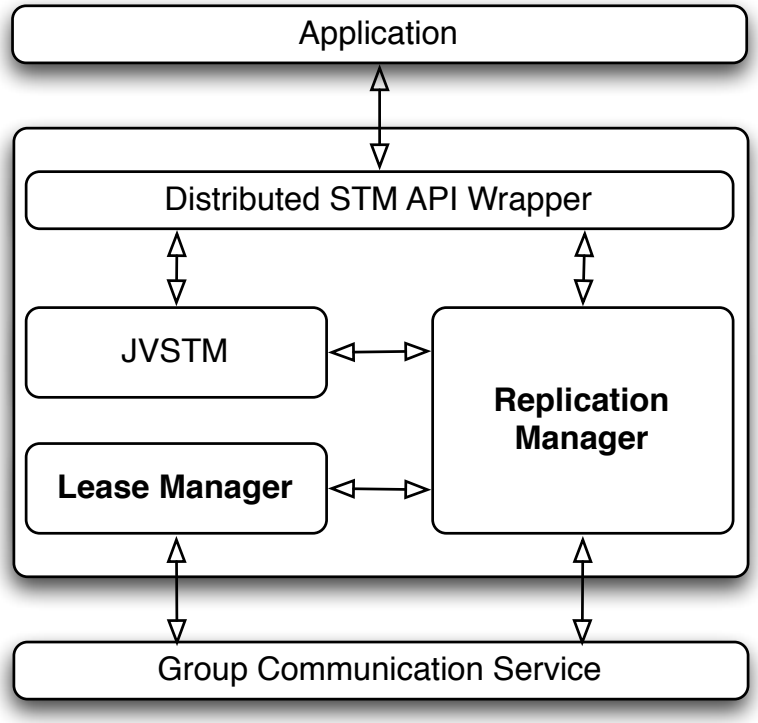
Benefits of ALC

- If applications exhibit some access locality:
  - avoid, or reduce frequency of, AB
  - locality enhanceable 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

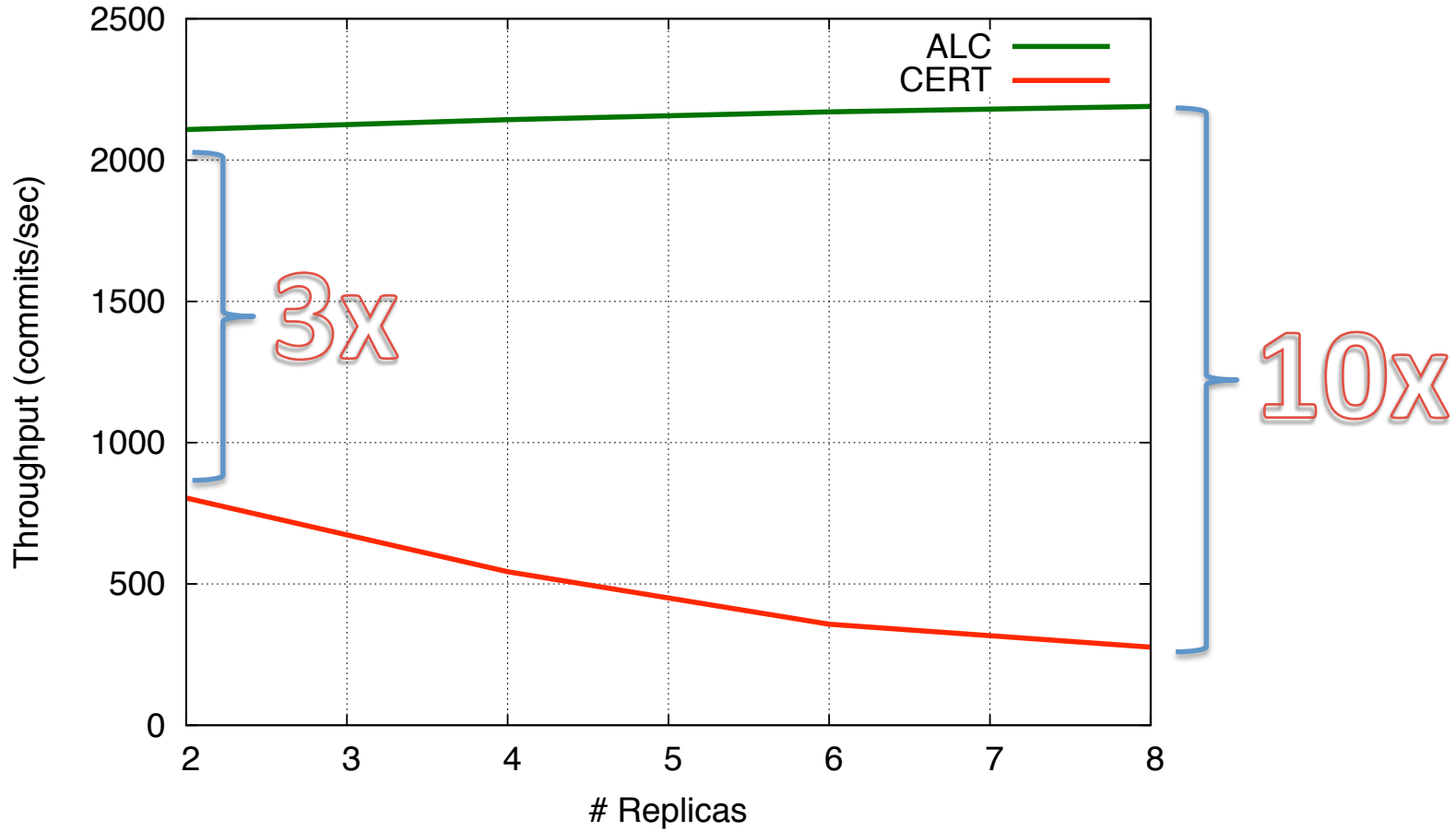
Performance evaluation

- Based on fully fledged prototype
- Relies on JVSTM
- Permits transparent execution of legacy (distribution agnostic) STM applications



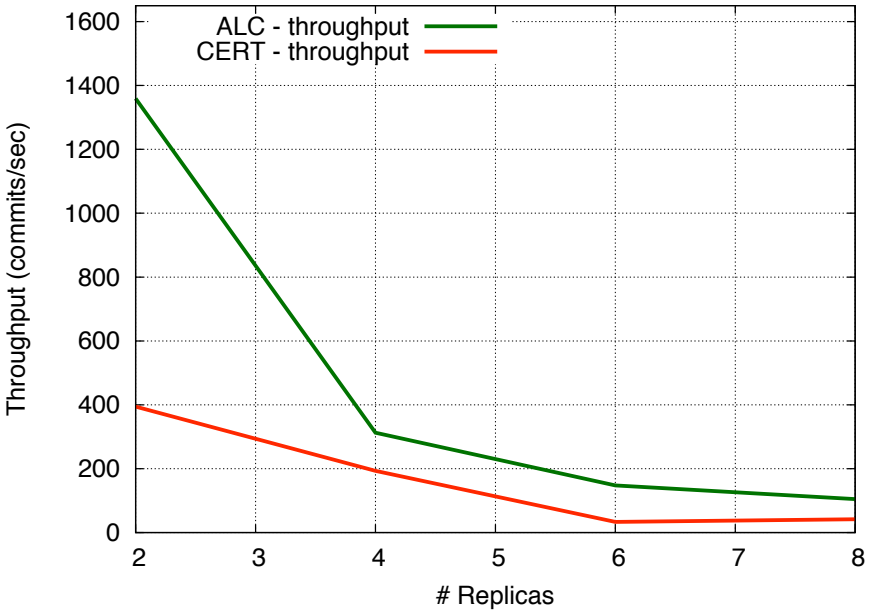
Synthetic “Best case” scenario

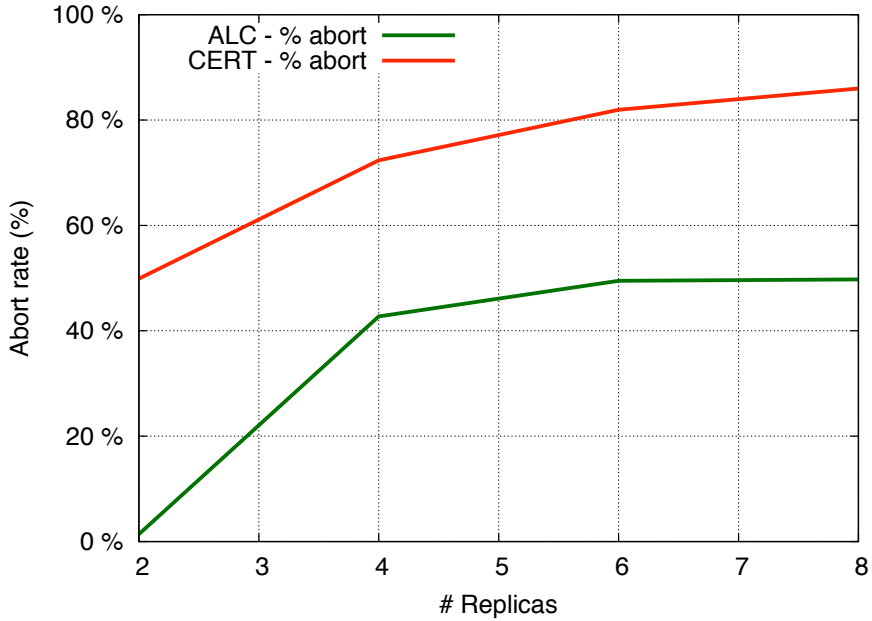
- Replicas accessing distinct memory regions



Synthetic “Worst case” scenario

- All replicas accessing the same memory region

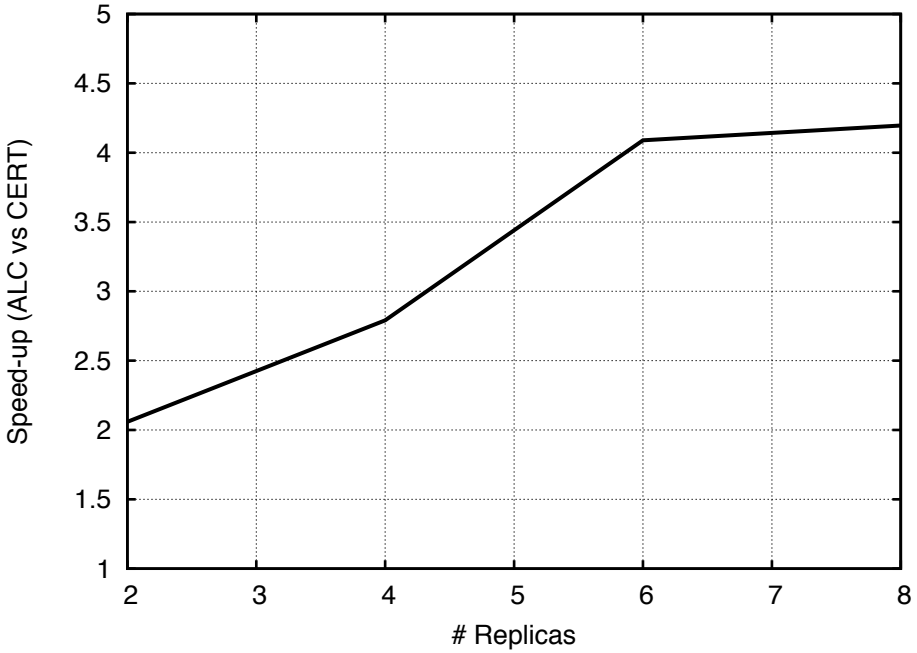


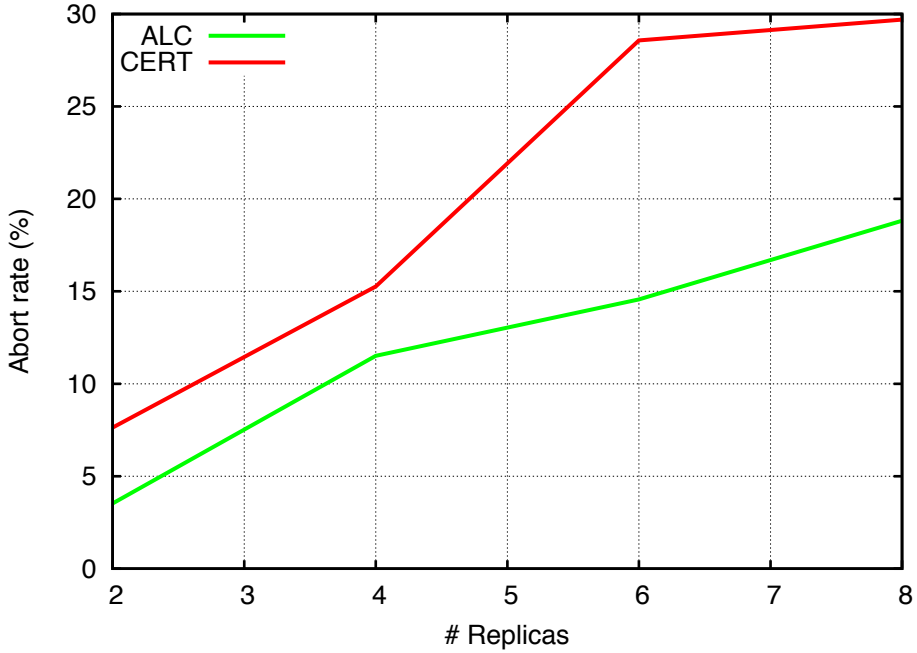


on av. ≈3x speedup due to reduced abort rate

Lee Benchmark

- Complex application with diverse workload:
  - both long and short running transactions





- long running transactions subject to livelock:
  - aborted up to 10 times

Speculative Transactional Replication [RPQCR10]

Beyond certification mechanisms

- Certification schemes achieve no overlapping between transaction processing and replica coordination:
  - AB is started only after transaction ends!

- Can't we do any better to minimize the coordination costs?

YES WE CAN!

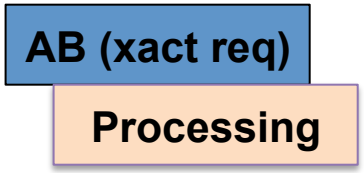
- Using optimistic deliveries + state machine:
  - messages are received from the network long before their final order is established by the AB

1. AB incoming transactions and execute on all nodes:
   - RPC-like execution fashion of the xacts
2. start processing as soon as a xact is opt-delivered

+ overlapping between processing & communication

Certification Scheme

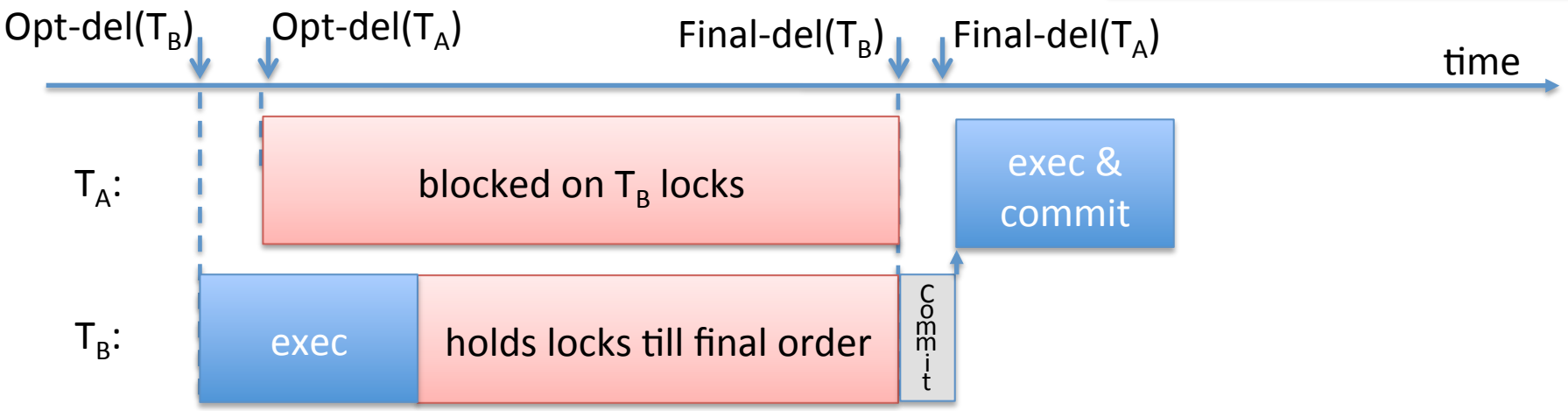
Speculative Scheme



Easier to say than to do….

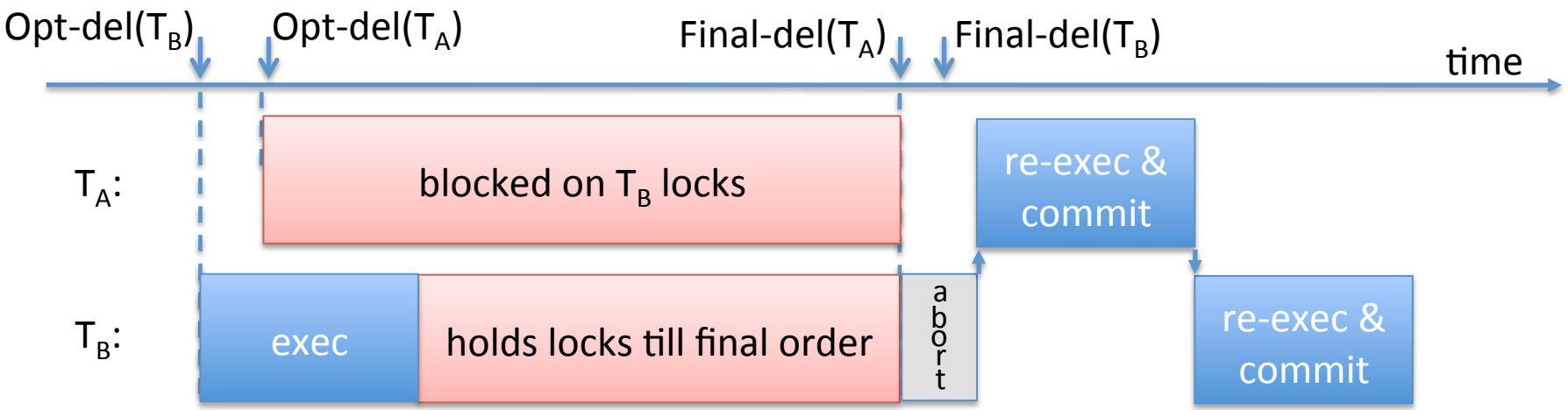
1. This only works if transactions execute deterministically at all replicas

- classic concurrency controls (e.g. 2PL) are not deterministic
- existing solutions have several key limitations:
  - a-priori knowledge of readsets/writesets:
    - may force to large conflict over-estimation
  - acquire **ALL** locks as xact begins
    - way more pessimistic than classic 2PL

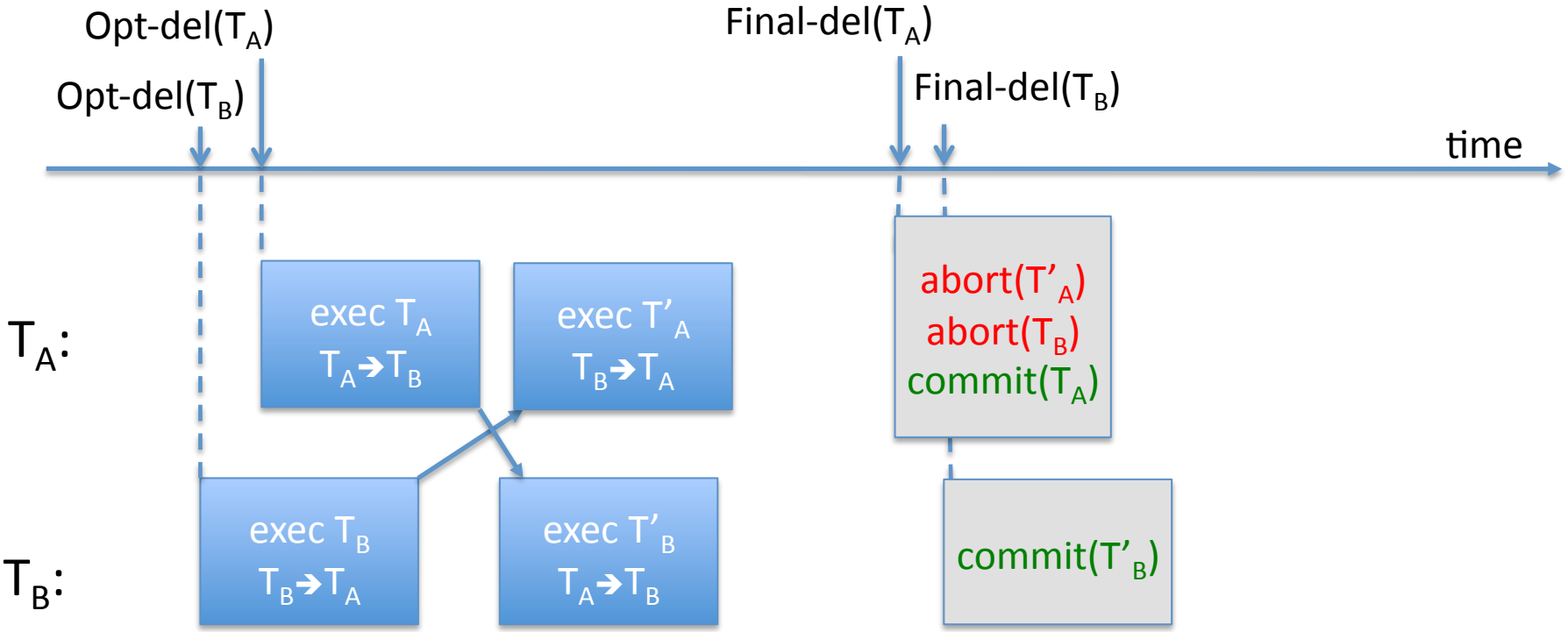


Easier to say than to do….

2. Vulnerable to mismatches between final and optimistic delivery orders!



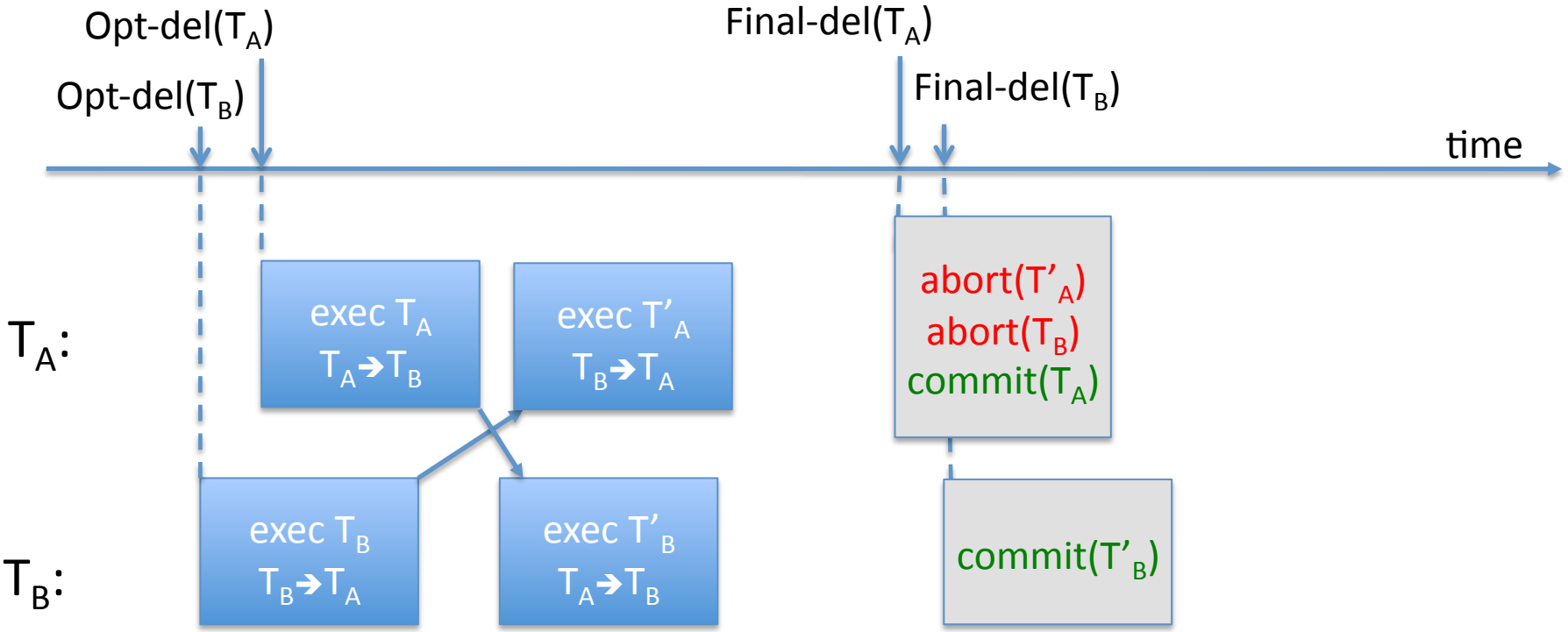
Don’t be pessimistic…be speculative!



Speculatively explore multiple Serialization Orders (SO)

+ **take maximum benefit from modern multi-core architectures**
+ **shelter from network reordering**
+ **avoid lock convoying**

Don’t be pessimistic…be speculative!



Speculatively explore multiple Serialization Orders (SO)

− **#SOs can grow factorially with #msgs not yet finally delivered**

- true in worst case: every xact conflicts with every other, hardly the case in practice

\+ **#SOs in which a xact observes distinct snapshots depends on actual conflict graph**

Problem formalization: Optimal STR protocol

$\Sigma=\{T_1,\ldots,T_n\}$: set of Opt-delivered, but not yet TO-delivered, transactions

$\Sigma'=\{T_1^1,\ldots, T_1^k,\ldots,T_n^1,\ldots,T_n^m\}$: set of fully executed speculative transactions

An optimal STR protocol must guarantee:

Consistency: each speculative xact is view-serializable

Non-redundancy: no two speculative xacts observe the same snapshot

Completeness: if system is quiescent (stops Opt- and TO-delivering messages) then, for every permutation $\pi(\Sigma)$ of Σ and for every T_i in Σ, eventually there is a T_i^j in $\pi(\Sigma)$ that has observed the same snaphot generated by sequentially executing all the transactions preceding T_i.

Filters out trivial solutions that blindly enumerate all permutations of Σ

Shelters from any mismatch between optimistic and final delivery order

An Optimal STR Protocol
Core Technical Challenge

- Design a provably optimal speculative concurrency control:
  - online algorithm driving the dynamic generation of speculative transactions based on actual tx's conflict patterns

An Optimal STR Protocol Core Technical Challenge

- Key Idea:
  - each speculative xact maintains a **Speculative Polygraph (SP)**

- **Speculative polygraphs:**
  - keeps track of conflicts developed with other xacts
  - embeds a family of direct graphs:
    - each digraph is associated with an equivalent serialization order for the transaction
  - unlike classic polygraphs accommodate for the coexistence of non-conciliable speculative transactions

What about complexity?

- Ensuring *completeness* in STR can be very expensive:
  - *NP complete problem [Papadimitrou79]*

- To what extent can speculation enhance performance if completeness is not ensured?

….up to 1 order of magnitude when spontaneous ordering holds (AGGRO)…

AGGRO
[PQR10]

AGGRessively Optimistic Transactional Replication - Overview

- STR protocol which sacrifices *completeness:*
  - speculative processing only according to optimistic delivery order

- Innovative concurrency control mechanism:
  - no a-priori knowledge of tx's read&write sets
  - attempts to serialize transactions according to opt-delivery order:
    - failure can cause abort, but opacity is guaranteed
  - multi-version, lock-based, visible reads

AGGRO – Algorithm

upon **opt-Deliver**(Tx Ti)

append Ti to OAB order

start transaction Ti in a speculative fashion

AGGRO – Algorithm

```
upon write(Tx Ti, Data X, Value v)
    if (X not already in Ti.WS)
     add X to Ti.WS
     mark X as WIP // C&S
     for each Tj that follows Ti in OAB order:
          if (Tj read X from a xact Tk preceding  Ti)  abort Tj
    else
     update X in Ti.WS
```

AGGRO – Algorithm

upon **read**(Tx Ti, Data X)

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

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

wait until (X is markedAsWip from a Tx
that precedes Ti in OAB order)

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

Ti.readFrom.add(Tj)

AGGRO – Algorithm

upon **completed_exec**(Tx Ti)

```
    atomically {
     for each X in Ti.WS: unmark X as WIP by Ti
    }
```

upon **commit**(Tx Ti)

```
    atomically {
     for each X in Ti.WS: mark X as committed
    }
```

AGGRO – Algorithm

upon **abort**(Tx Ti)

abort any transaction that read from Ti

restart Ti

upon **TO-Deliver**(Tx 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)

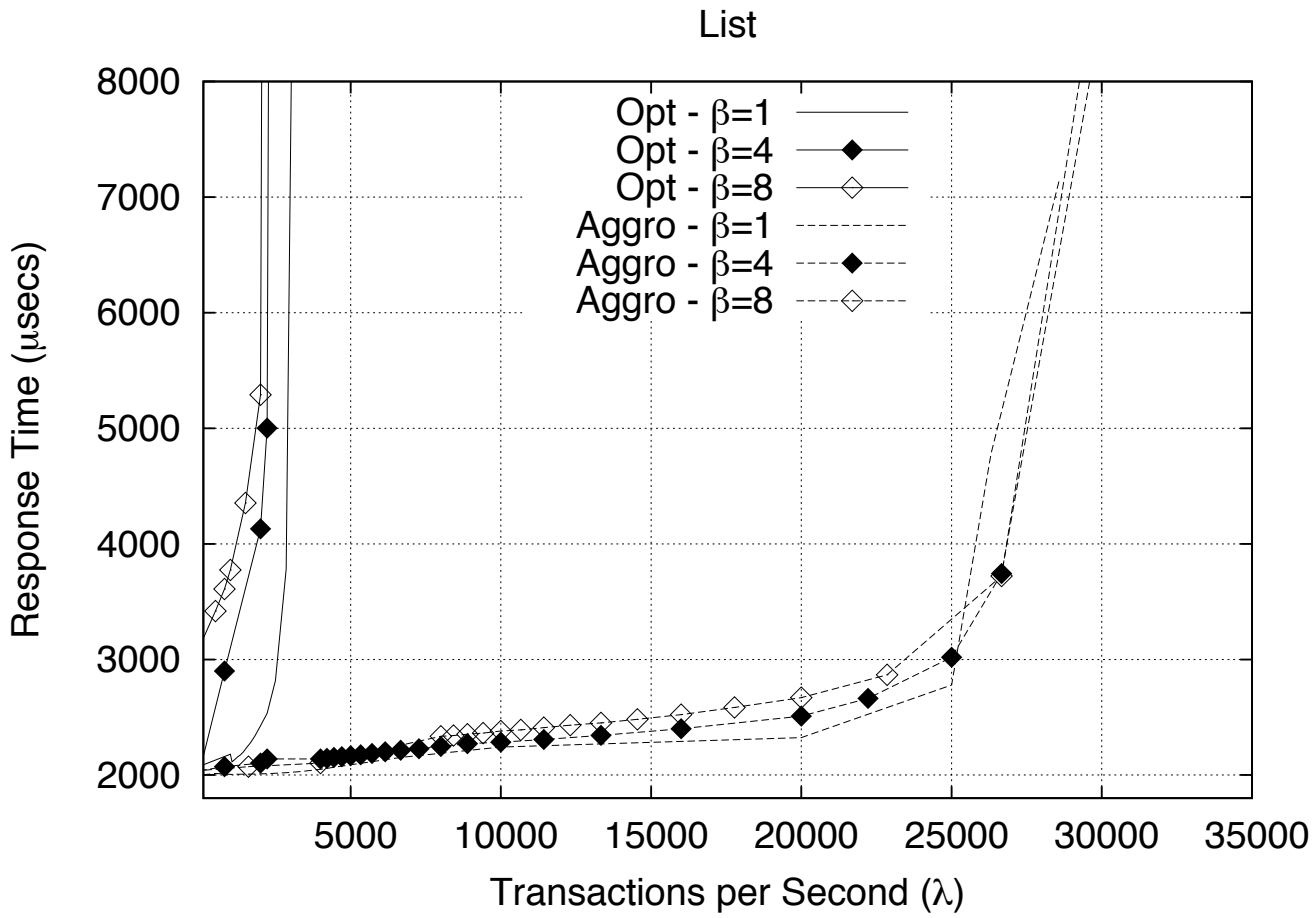
else commit(Ti)

AGGRO – evaluation

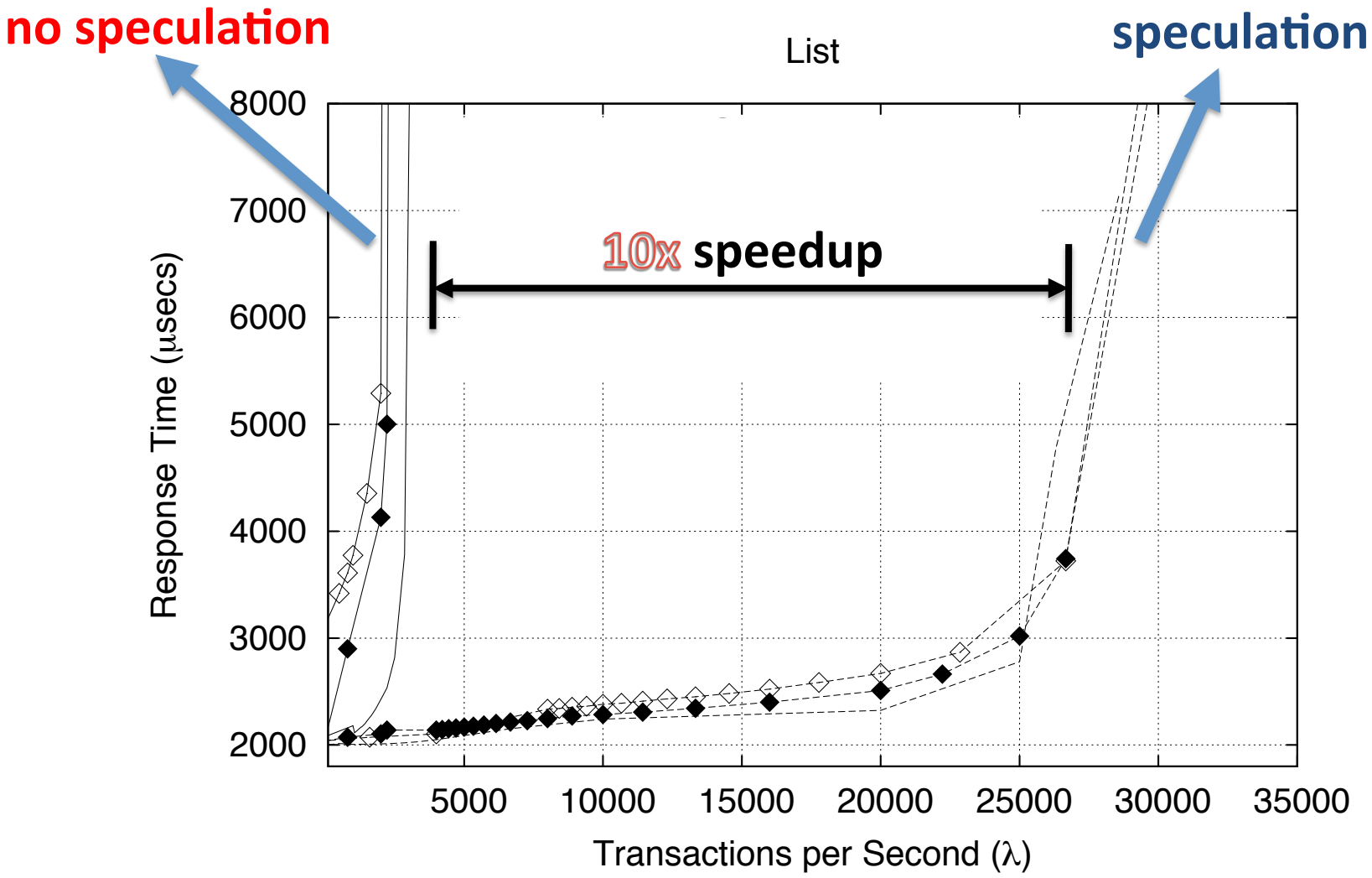
Trace driven simulation:

- data access pattern trace of STM micro-benchmarks running on JVSTM:
  - List & RBTree
- Atomic Broadcast:
  - optimistic delivery: 500 µsec
  - final (total ordered) delivery: 2 msec
  - various levels of message batching:
    - common optimization to enhance AB throughput

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



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



Future Research Directions

DTMs: a programming paradigm for the Cloud ?

- TMs emerged as a powerful abstraction to simplify programming of multi-core systems

- Will DTMs make their way into Cloud platforms? [RRCC10]
  - more powerful alternative to paradigms such as MapReduce
  - several in memory transactional data grids are starting to appear:
    - Oracle Coherence, JBoss Infinispan

DTMs: a programming paradigm for the Cloud ?

- Challenge: how to deal with Cloud's elasticity?
  - system scale fluctuates depending on workload
    - what-you-use-is-what-you-pay pricing model

DTMs: a programming paradigm for the Cloud ?

- Challenge: how to deal with Cloud's elasticity?
  - system scale fluctuates depending on workload
    - what-you-use-is-what-you-pay pricing model

Question 1:

How to reduce data relocation overhead?

- highly dynamic system
- much to borrow from P2P literature:
  - » consistent hashing
  - » DHTs….

DTMs: a programming paradigm for the Cloud ?

- Challenge: how to deal with Cloud's elasticity?
  - system scale fluctuates depending on workload
    - what-you-use-is-what-you-pay pricing model

Question 2:

How to ensure optimal performance when:

- scales varies from few to hundreds of nodes
- workload characteristics change

No One-Size-Fits-All solutions

DTMs: a programming paradigm for the Cloud ?

- Challenge: how to deal with Cloud's elasticity?
  - system scale fluctuates depending on workload
    - what-you-use-is-what-you-pay pricing model

Question 3:

How to integrate DTM & Cloud storage solutions?

DTMs: 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

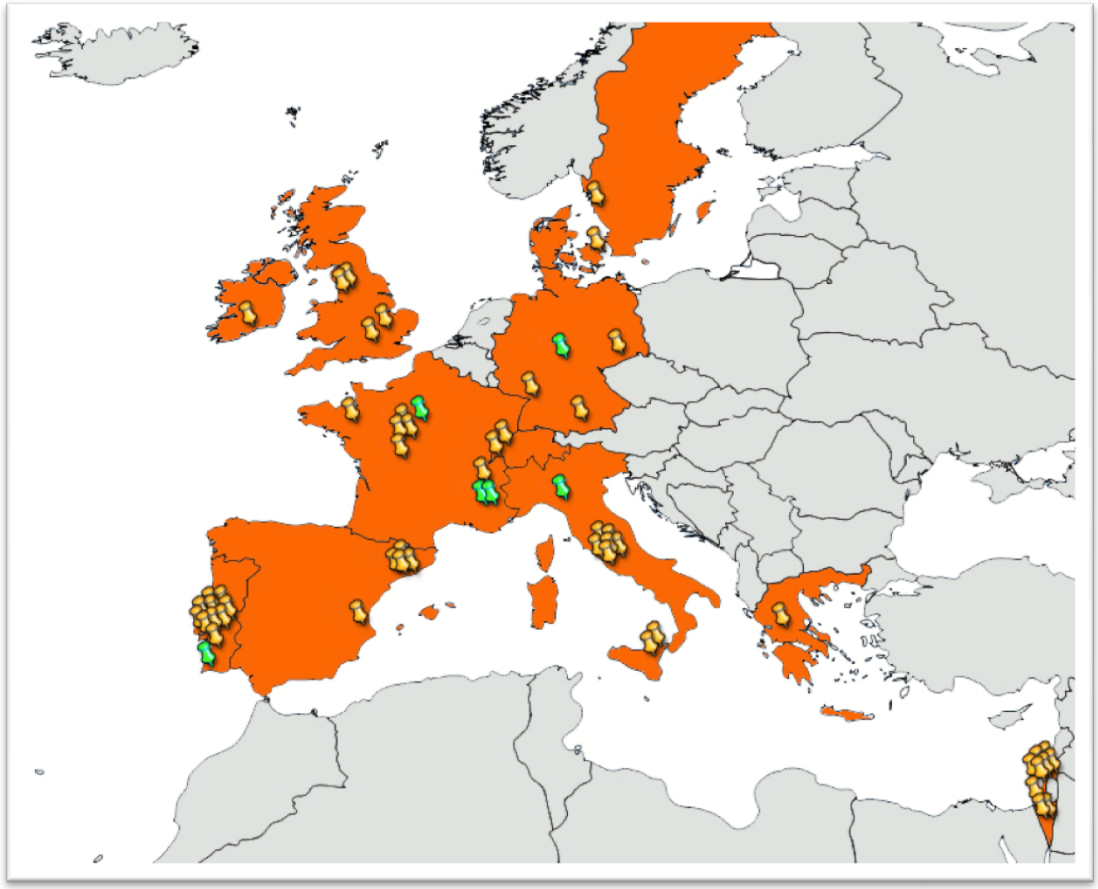
- Participation can be extended also to non-European countries!

- Kick-off: 1 April 2011

- Contact us if you are interested in joining it:

romano@inesc-id.pt

ler@inesc-id.pt




That's all Folks!

References

[AGHK06] Hagit Attiya, Leah Epstein, Hadas Shachnai, and Tami Tamir, `Transactional contention management as a non-clairvoyant scheduling problem" PODC 2006.

[AKWKLJ08] Mohammad Ansari, Christos Kotselidis, Kim Jarvis, Mikel Luján, Chris Kirkham, and Ian Watson, Lee-TM: A Non-trivial Benchmark for Transactional Memory, In Proceedings of the 8th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP 2008), Aiya Napa, Cyprus, June 2008.

[AMSVK09] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2009. Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. 27, 3, Article 5 (November 2009)

[BAC08] Robert L. Bocchino, Vikram S. Adve, and Bradford L. Chamberlain. 2008. Software transactional memory for large scale clusters. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). ACM, New York, NY, USA, 247-258

[CRR10] N. Carvalho, Paolo Romano and L. Rodrigues, Asynchronous Lease-based Replication of Software Transactional Memory, Proceedings of the ACM/IFIP/USENIX 11th Middleware Conference (Middleware), Bangalore, India, ACM Press, November 2010

[CRRC09] M. Couceiro, Paolo Romano, L. Rodrigues and N. Carvalho, D2STM: Dependable Distributed Software Transactional Memory, Proc. IEEE 15th Pacific Rim International Symposium on Dependable Computing (PRDC'09)

[CS06] João Cachopo and Antonio Rito-Silva. 2006. Versioned boxes as the basis for memory transactions. Sci. Comput. Program. 63, 2 (December 2006), 172-185.

[DSS06] D. Dice, O.Shalev, N. Shavit, Transactional Locking II, In In Proc. of the 20th Intl. Symp. on Distributed Computing (2006)

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

[GK08] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. In Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming (PPoPP '08). ACM, New York, NY, USA, 175-184.

References

[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

[HLMS03] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N. Scherer, III. 2003. Software transactional memory for dynamic-sized data structures. In Proceedings of the twenty-second annual symposium on Principles of distributed computing (PODC '03). ACM, New York, NY, USA, 92-101.

[JKV07] Rachid Guerraoui, Michal Kapalka, and Jan Vitek. 2007. STMBench7: a benchmark for software transactional memory. SIGOPS Oper. Syst. Rev. 41, 3 (March 2007), 315-324.

[MCKO08] Chi Cao Minh, JaeWoong Chung, Christos Kozyrakis, Kunle Olukotun, STAMP: Stanford Transactional Applications for Multi-Processing, In IISWC '08: Proceedings of The IEEE International Symposium on Workload Characterization, Sept. 2008.

[MMA06] Kaloian Manassiev, Madalin Mihailescu, and Cristiana Amza. 2006. Exploiting distributed version concurrency in a transactional memory cluster. In Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming (PPoPP '06). ACM, New York, NY, USA, 198-208.

[KAJLKW08] 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

[PQR10] R. Palmieri, Paolo Romano and F. Quaglia, AGGRO: Boosting STM Replication via Aggressively Optimistic Transaction Processing, Proc. 9th IEEE International Symposium on Network Computing and Applications (NCA), Cambridge, Massachussets, USA, IEEE Computer Society Press, July 2010

[RPQCR10] Paolo Romano, R. Palmieri, F. Quaglia, N. Carvalho and L. Rodrigues, An Optimal Speculative Transactional Replication Protocol, Proc. 8th IEEE International Symposium on Parallel and Distributed Processing with Applications (ISPA), Taiwan, Taipei, IEEE Computer Society Press, September 2010

[RRCC10] Paolo Romano, L. Rodrigues, N. Carvalho and J. Cachopo, Cloud-TM: Harnessing the Cloud with Distributed Transactional Memories , ACM SIGOPS Operating Systems Review, Volume 44 , Issue 2, April 2010

Additional interesting readings on (non-distributed) TMs

- Hagit Attiya, "*The inherent complexity of transactional memory and what to do about it"*, Invited talk at PODC 2010.
- Maurice Herlihy and Nir Shavit, "*The art of multiprocessor programming",* Morgan Kaufmann, 2008
- Rachid Guerraoui, "*Tutorial on transactional memory*", CAV 2009.
- Nir Shavit, "*Software Transactional Memory*", Keynote Lecture at DISC 2009.