Scheduling Transactions in Replicated Distributed Transactional Memory

Junwhan Kim and Binoy Ravindran

Virginia Tech

USA

{junwhan,binoy}@vt.edu

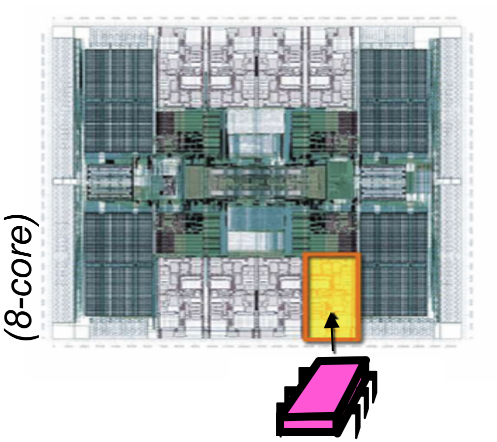
CCGrid 2013

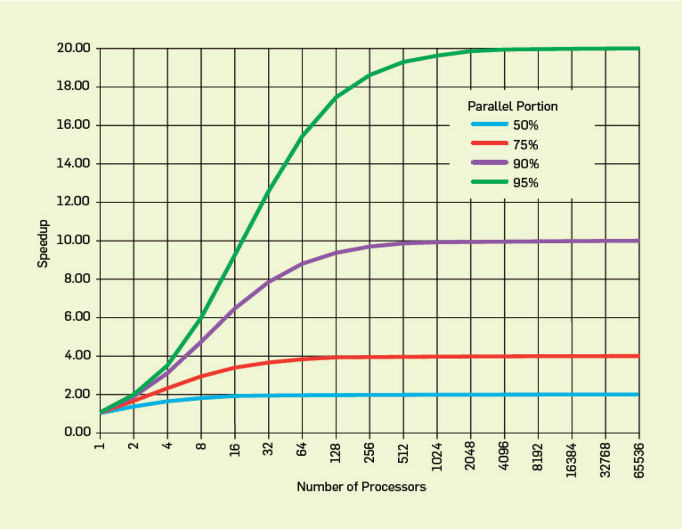
Virginia Tech

Concurrency control on chip multiprocessors significantly affects performance (and programmability)

- Improve performance by exposing greater concurrency
  - *Amdahl's law: relationship between sequential execution time and speedup reduction is not linear*

Sun T2000 Niagara (8-core)





Lock-based concurrency control has serious drawbacks

- Coarse grained locking
  - Simple
  - But no concurrency

```
public boolean add(int item) {
  Node pred, curr;
  lock.lock();
  try {
   pred = head;
   curr = pred.next;
   while (curr.val < item) {
    pred = curr;
    curr = curr.next;
   }
   if (item == curr.val) {
    return false;
   } else {
    Node node = new Node(item);
    node.next = curr;
    pred.next = node;
    return true;
   }
  } finally {
   lock.unlock();
  }
}
```

Fine-grained locking is better, but…

- Excellent performance
- Poor programmability

- Lock problems don’t go away!
  - Deadlocks, livelocks, lock-convoying, priority inversion,….

- Most significant difficulty – composition

```
public boolean add(int item) {
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.val < item) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return false;
      }
      Node newNode = new Node(item);
      newNode.next = curr;
      pred.next = newNode;
      return true;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Lock-free synchronization overcomes some of these difficulties, but…

“lock-free retry loop”

```
public boolean add(int item) {
 while (true) {
   Node pred = null, curr = null, succ = null;
   boolean[] marked = {false};  boolean snip;
   retry: while (true) {
     pred = head; curr = pred.next.getReference();
     while (true) {
       succ = curr.next.get(marked);
       while (marked[0]) {
         snip = pred.next.compareAndSet(curr, succ, false, false);
         if (!snip) continue retry;
          curr = succ; succ = curr.next.get(marked);
       }
       if (curr.val < item)
           pred = curr; curr = succ;
     }
   }
   if (curr.val == item) {  return false;
   } else {
     Node node = new Node(item);
     node.next = new AtomicMarkableReference(curr, false);
     if (pred.next.compareAndSet(curr, node, false, false)) {return true;}
   }
 }
}
```

Transactional memory

- Like database transactions
- ACI properties (no D)
- Easier to program
- Composable

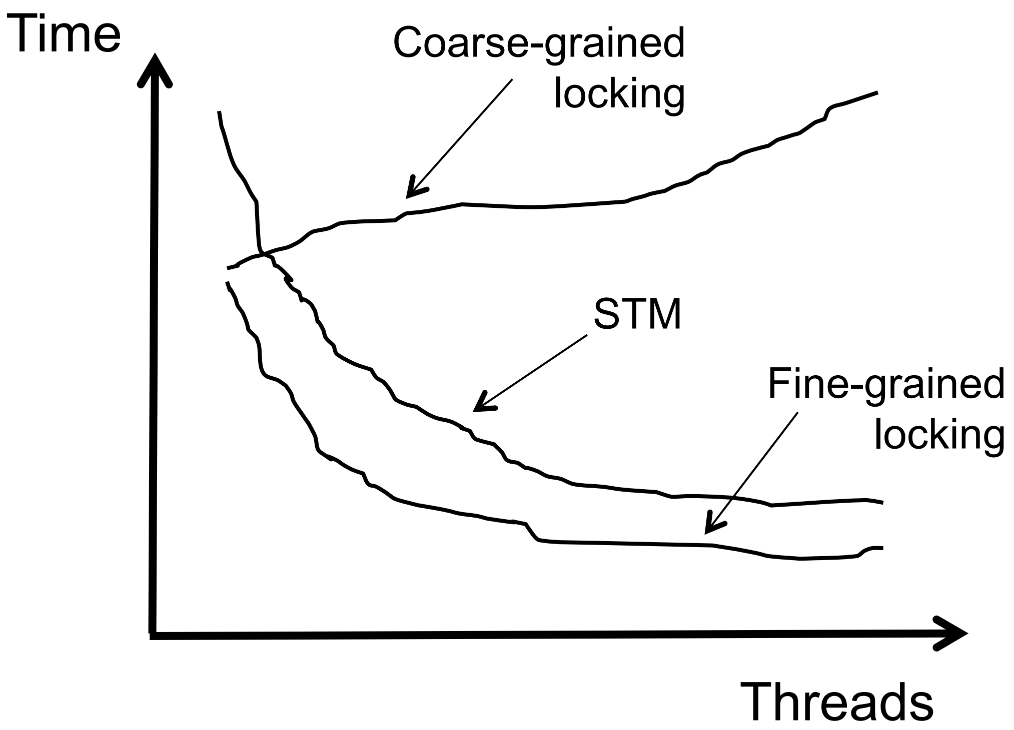
- First HTM, then STM, later HyTM

```
public boolean add(int item) {
  Node pred, curr;
  atomic {
    pred = head;
    curr = pred.next;
    while (curr.val < item) {
      pred = curr;
      curr = curr.next;
    }
    if (item == curr.val) {
      return false;
    } else {
      Node node = new Node(item);
      node.next = curr;
      pred.next = node;
      return true;
    }
  }
}
```


M. Herlihy and J. B. Moss (1993). Transactional memory: Architectural support for lock-free data structures. *ISCA*. pp. 289–300.

N. Shavit and D. Touitou (1995). Software Transactional Memory. *PODC*. pp. 204—213.

Optimistic execution yields performance gains at the simplicity of coarse-grain, but no silver bullet



- High data dependencies
- Irrevocable operations
- Interaction between transactions and non-transactions
- Conditional waiting
- ……

E.g., C/C++ Intel Run-Time System STM (B. Saha et. al. (2006). McRT-STM: A High Performance Software Transactional Memory. *ACM PPoPP*)

Three key mechanisms needed to create atomicity illusion

Versioning

```
atomic{
    x = x + y;
}
```

Where to store new `x` until commit?

- *Eager*: store new `x` in memory; old in *undo log*
- *Lazy*: store new `x` in *write buffer*

Conflict detection

T0

```
atomic{
    x = x + y;
}
```

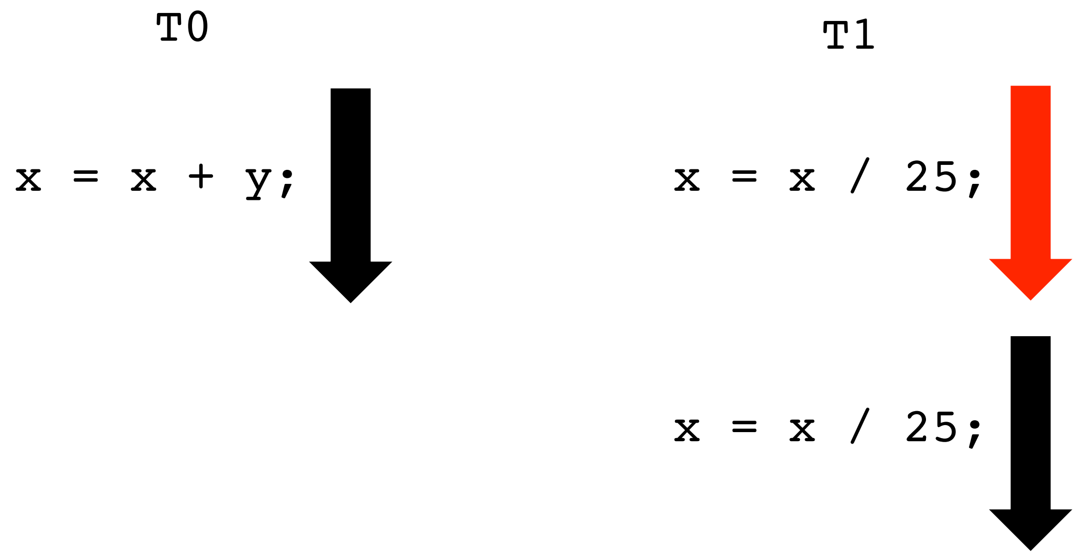
T1

```
atomic{
    x = x / 25;
}
```

How to detect conflicts between `T0` and `T1`?

- Record memory locations read in *read set*
- Record memory locations wrote in *write set*
- Conflict if one's read or write set intersects the other's write set

Third mechanism is contention management



Which transaction to abort?

- Greedy: favor those with an earlier start time
- Karma: ….

Transactional scheduler is not necessary, but can boost performance

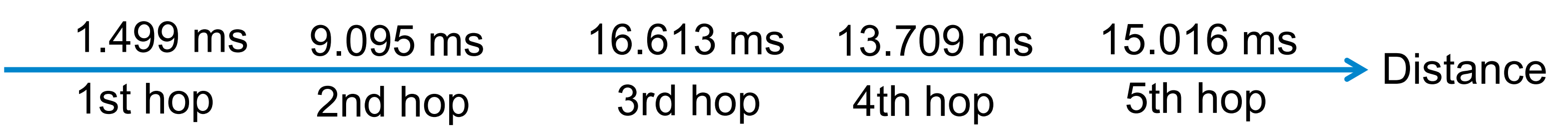
- Contention manager
  - Can cause too many aborts, e.g., when a long running transaction conflicts with shorter transactions
  - An aborted transaction may wait too long

- Transactional scheduler's goal: minimize conflicts (e.g., avoid repeated aborts)

Walther M. et al. (2010). Scheduling support for transactional memory contention management, *PPoPP*, pp 79 - 90

Distributed TM (or DTM)

- Extends TM to distributed systems
  - Nodes interconnected using message passing links
- Execution and network models
  - Execution models
    - Data flow DTM (DISC 05)
      - Transactions are immobile
      - Objects migrate to invoking transactions
    - Control flow DTM (USENIX 12)
      - Objects are immobile
      - Transactions move from node to node
  - Herlihy's metric-space network model (DISC 05)
    - Communication delay between every pair of nodes
    - Delay depends upon node-to-node distance



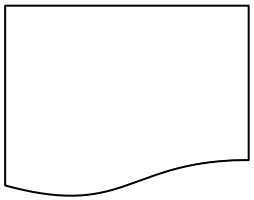
Past research have developed several transactional schedulers

- Multi-core systems
  - BiModal transactional scheduler (OPODIS 09)
  - Proactive transactional scheduler (MICRO 09)
  - Adaptive transactional scheduler (SPAA 08)
  - Steal-On-Abort (HiPEAC 09)
  - CAR-STM (PODC 08)

- Distributed systems
  - Bi-interval transactional scheduler (SSS 10)
    - Single-copy
  - Reactive transactional scheduler (IPDPS 12)
    - Single-copy (and closed-nested transactions)

Replication models in (dataflow) DTM

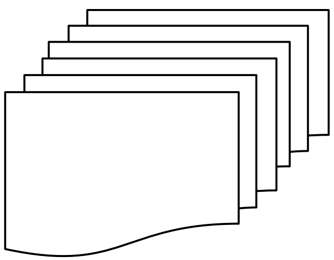
- No replication: non-fault-tolerant



Only one copy for each object

Single points of failure

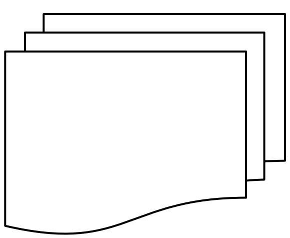
- Full replication: fault-tolerant, but non-scalable



All objects replicated on all nodes

Atomic broadcasting of updates is **non-scalable**

- Partial replication: fault-tolerant and scalable



Each object replicated only at a subset of nodes

Updates atomically broadcast to only node subset

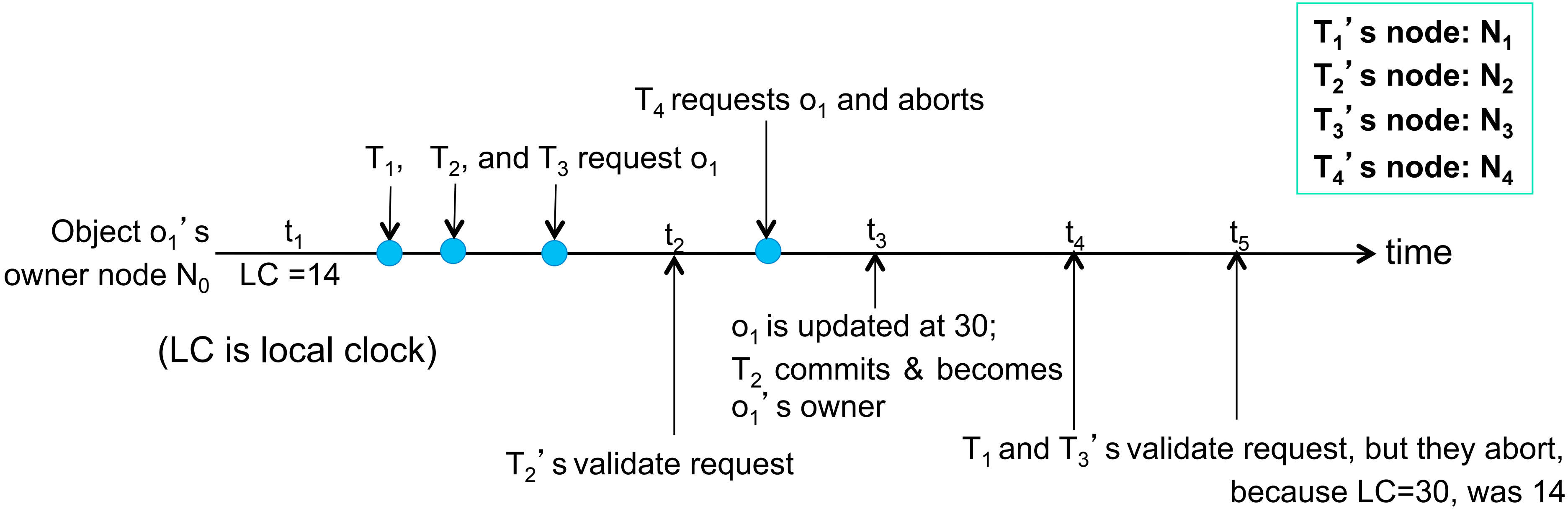
N. Schiper, P. Sutra, and F. Pedone (2010). P-store: Genuine partial replication in wide area networks. *SRDS*. pp. 214–224

Paper's contribution

- Scheduling in partially replicated DTM
  - Extend TFA for partial replication
  - Cluster-based transactional scheduler (CTS)
- Competitive ratio analysis
- Implementation and experimental studies
  - Comparisons with state-of-the art DTMs

M. Saad and B. Ravindran (2011). Hyflow: A high performance distributed software transactional memory framework, *HPDC*, pp. 265-266

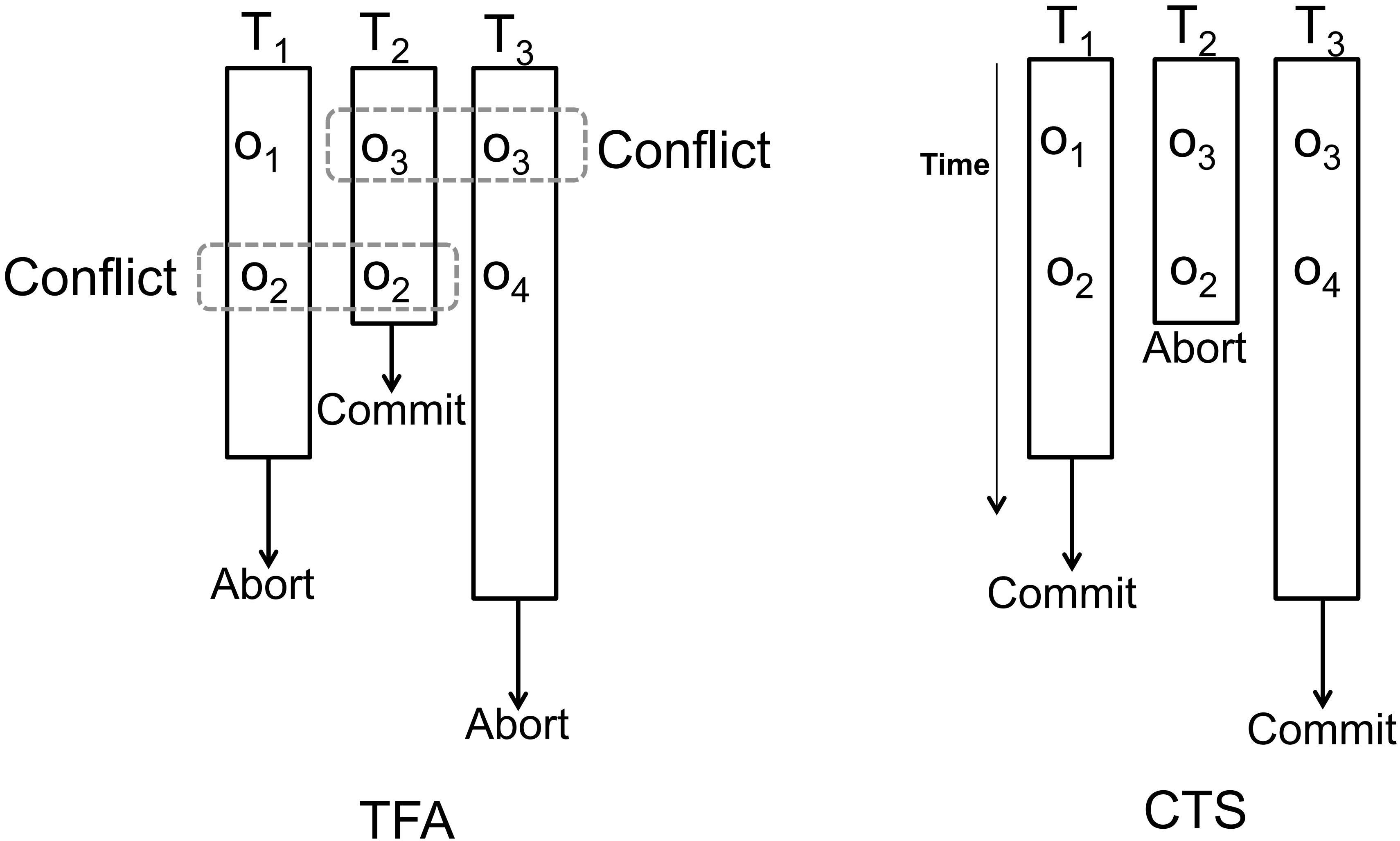
Atomicity, consistency, and isolation in data-flow DTM



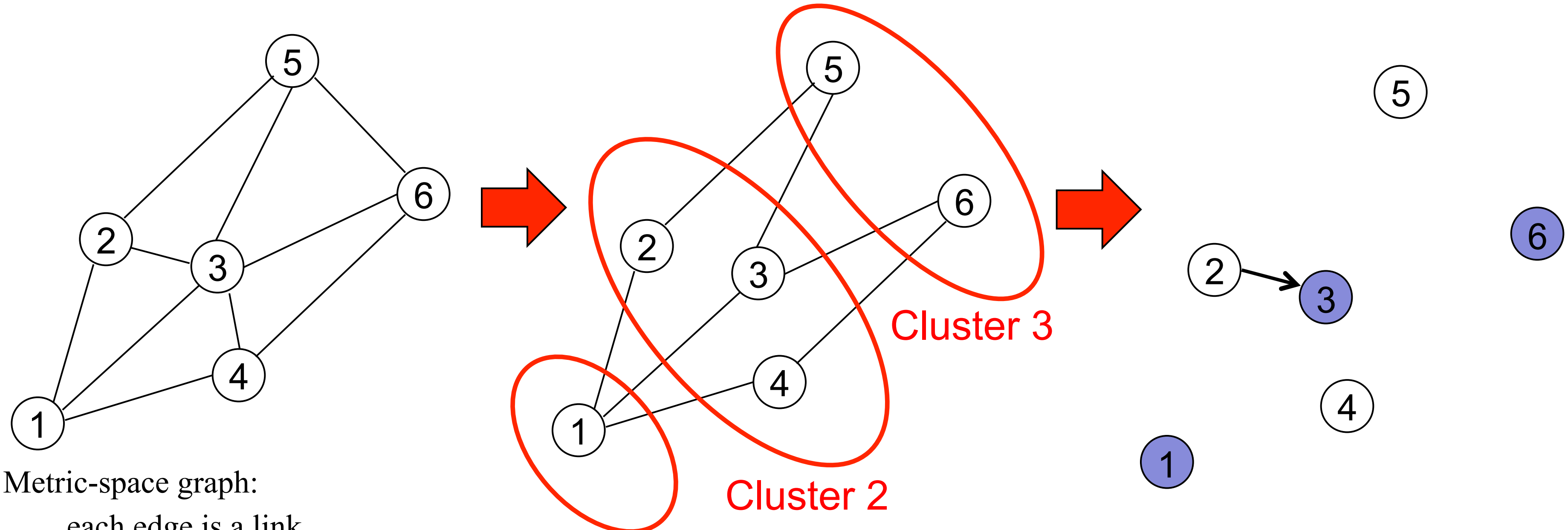
- Transactional Forwarding Algorithm (TFA)
  - Early validation of remote objects
  - Atomicity for object operations in the presence of asynchronous clocks

M. Saad and B. Ravindran (2011). Hyflow: A high performance distributed software transactional memory framework, *HPDC*, pp. 265-266

Motivating example for CTS



Logical partitioning for partial replication



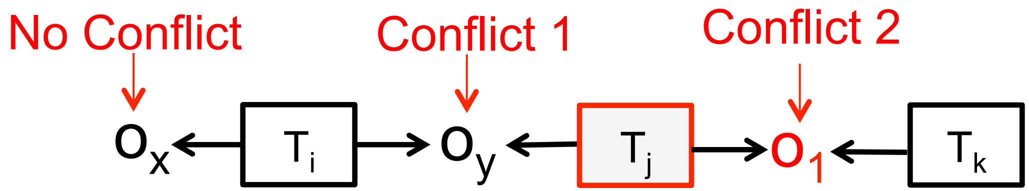
Metric-space graph:
each edge is a link
edge length is distance

- Purpose of partitioning is to enhance locality
  - Partitioning using METIS (SIAM 98)
- Each cluster has an object replica
  - Each cluster has one *object owner* who “owns” all replicas
- A transaction on node 2 requests objects from object owner in node 2’s cluster

G. Karypis and V. Kumar (1998). A fast and high quality multilevel scheme for partitioning irregular graph, *SIAM-JSC*, pp. 359-392

Scheduler design (1/2)

T: requesting transaction
O: requested object



TxTable

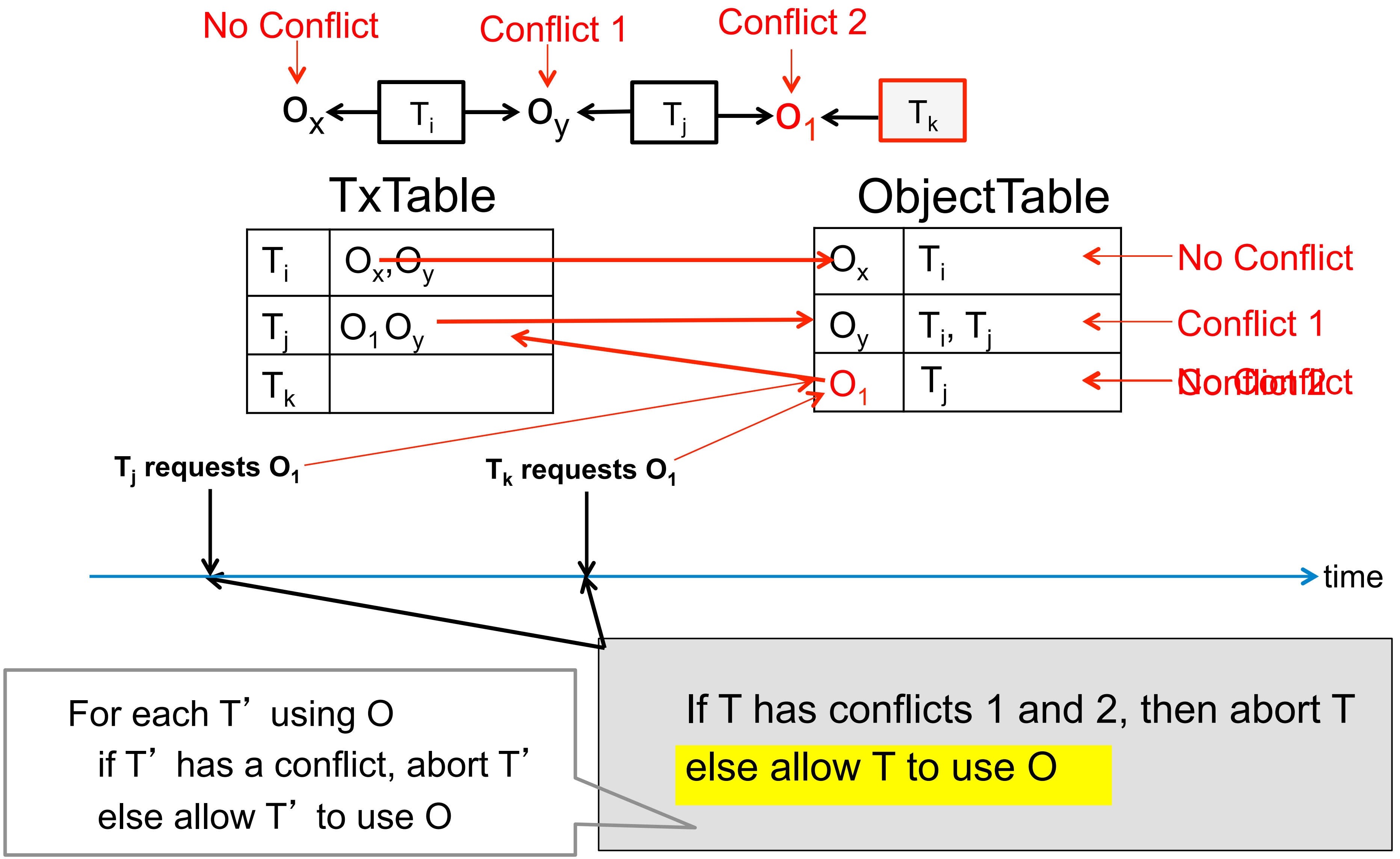
T_i	O_x, O_y
T_j	O_y
T_k	O_1

ObjectTable

O_x	T_i	No Conflict
O_y	$T_i T_j$	Conflict 1
O_1	T_k	Conflict 2

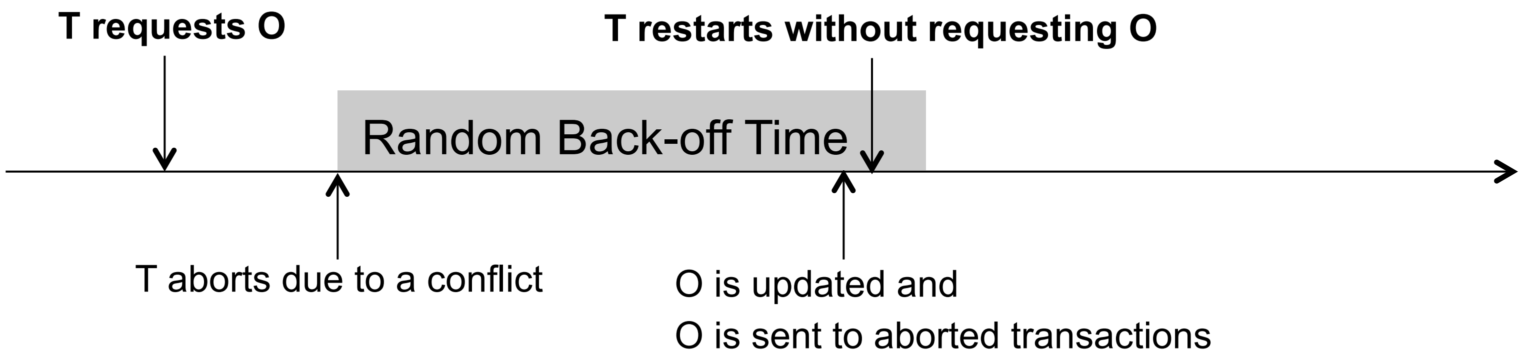
If T has conflicts 1 and 2, then abort T
else allow T to use O

Scheduler design (2/2)



Competitive ratio analysis

- *makespan$_A$* : time that *A* needs to complete *N* transactions
- Definition: Replication Model
  - FR: Full Replication, PR: Partial Replication, NR: No Replication
- *Theorem 1*: *makespan*(FR) < *makespan*(PR) < *makespan*(NR)
- *Theorem 2*: *makespan$_{CTS}$*(PR) < *makespan*(FR), where N > 3



PR incurs requesting and object retrieving times for transactions, but aborted transactions are resent updated objects.

CTS's backoff *generally* allows the update to be received before transaction re-start, resulting in less overall time than FR's broadcasting time.

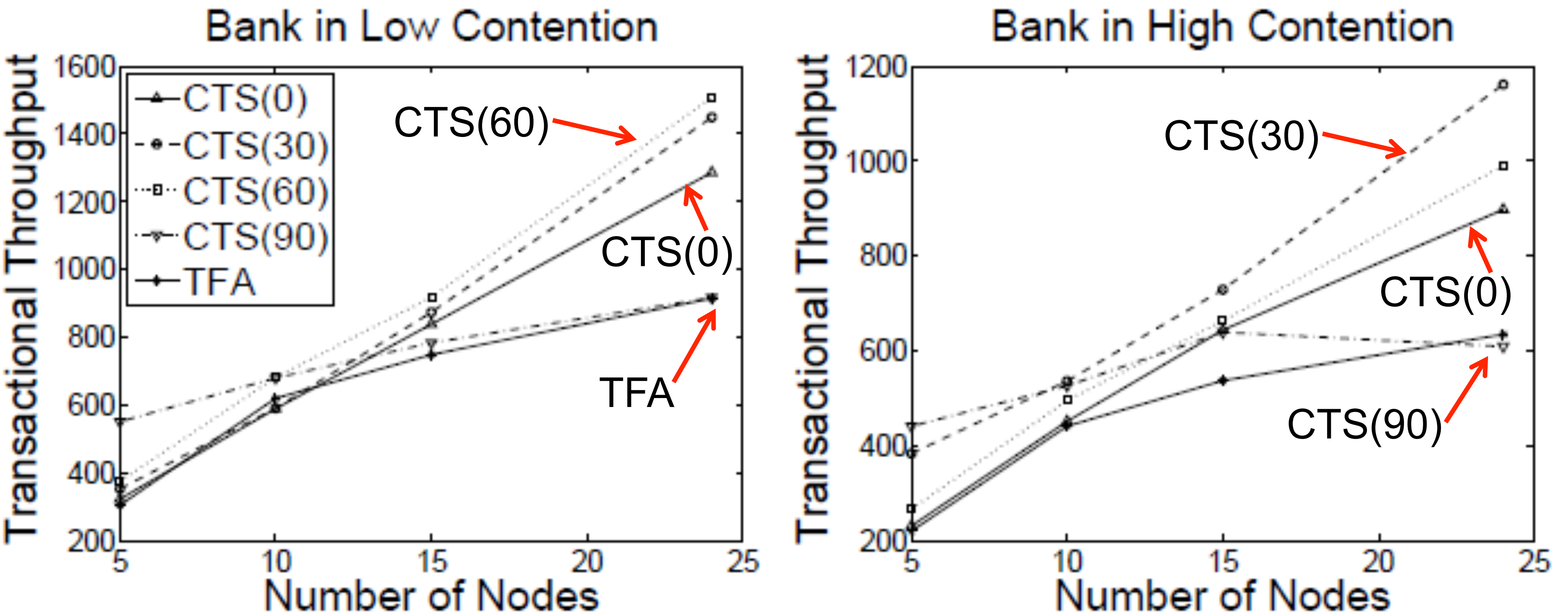
Implementation and experimental setup

- Implemented CTS in HyFlow DTM framework
  - Second generation DTM framework for the JVM (Java, Scala)
  - Open-source**: hyflow.org**
- 24 nodes, each is 2GHz AMD Opteron
- Benchmarks
  - Distributed version of STAMP Vacation
  - Two monetary applications
  - Distributed data structures
    - Counter, Red/Black Tree, DHT
- CTS(30) and CTS(60)
  - CTS over 30% and 60% of the nodes are object owners

M. Saad and B. Ravindran (2011) . Hyflow: A high performance distributed software transactional memory framework, *HPDC*, pp. 265-266

C. Minh, et al. (2008). STAMP: Stanford Transactional Applications for Multi-Processing, *IISWC* , pp. 200-208

Evaluation: Throughput with no node failures



Low Contention: 90% Read Transactions

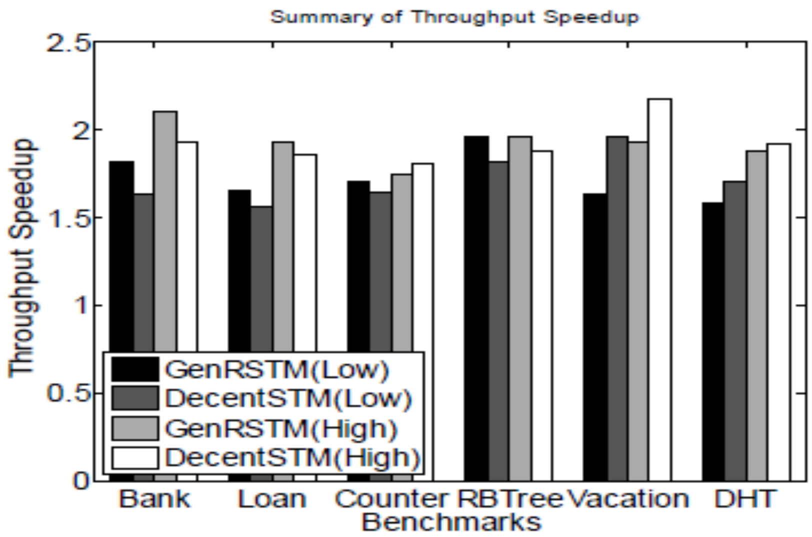
High Contention: 10% Read Transactions

CTS(0): TFA + CTS, but no replication and no fault tolerance

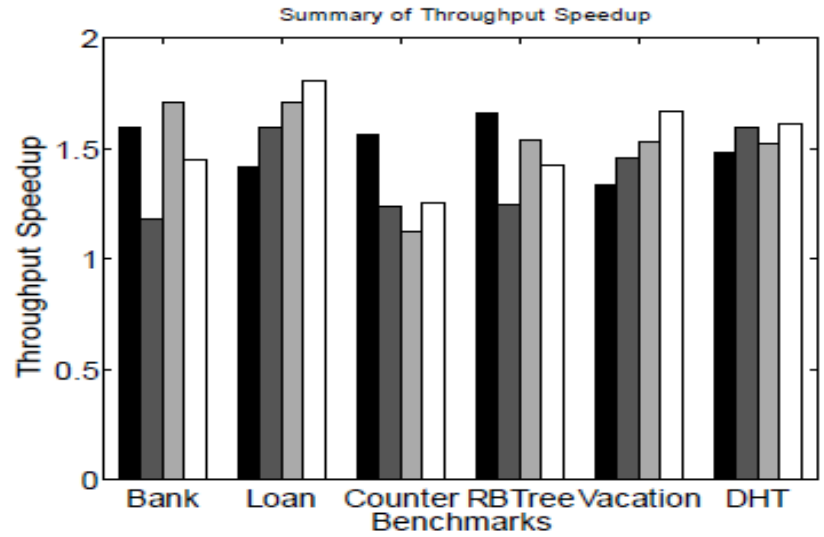
CTS(90): high communication overhead

TFA: no CTS

Evaluation:
Throughput under node failures



Throughput speedup under 20% node failure



Throughput speedup under 50% node failure

- Nodes randomly failed during each experiment
- GenRSTM and DecentSTM use full replication model
- Throughput speedup of CTS(60) over GenRSTM and DecentSTM
  - Speedup range from 1.51x to 2.3x in low contention
  - Speedup range from 1.3x to 1.7x in high contention
- CTS has reasonable performance at 50% failure (GenRSTM and DecentSTM have high communication delay overheads)

Conclusions

- DTM transactional scheduler in partial replication model
  - Uses multiple clusters to support partial replication for fault-tolerance
  - Clusters with small inter-node communication
  - Identifies transactions for aborting to enhancing concurrency
  - Enhances transactional throughput
    - 1.5x over baseline TFA; 1.55x and 1.73x over others

- Tradeoff between locality, communication cost, and fault-tolerance
- Can be effectively exploited in DTM

- Adaptive partial replication?
- Adaptive backoff scheme?
- …