An Automated Framework for Decomposing Memory Transactions to Exploit Partial Rollback

Aditya Dhoke, Roberto Palmieri, and <u>Binoy Ravindran</u>

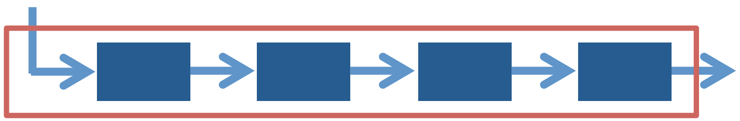
Systems Software Research Group

Virginia Tech

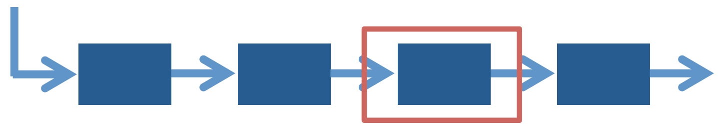
IPDPS 2015

Lock-based concurrency control has serious drawbacks

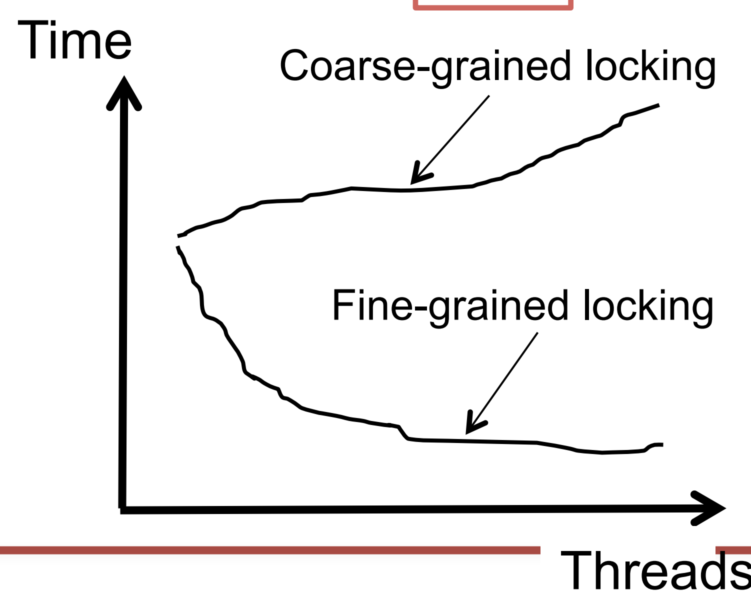
- Coarse-grained locking
  - Simple, but no concurrency



- Fine-grained locking
  - Excellent performance, but poor programmability
  - Hard to compose



```
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();
  }
}
```

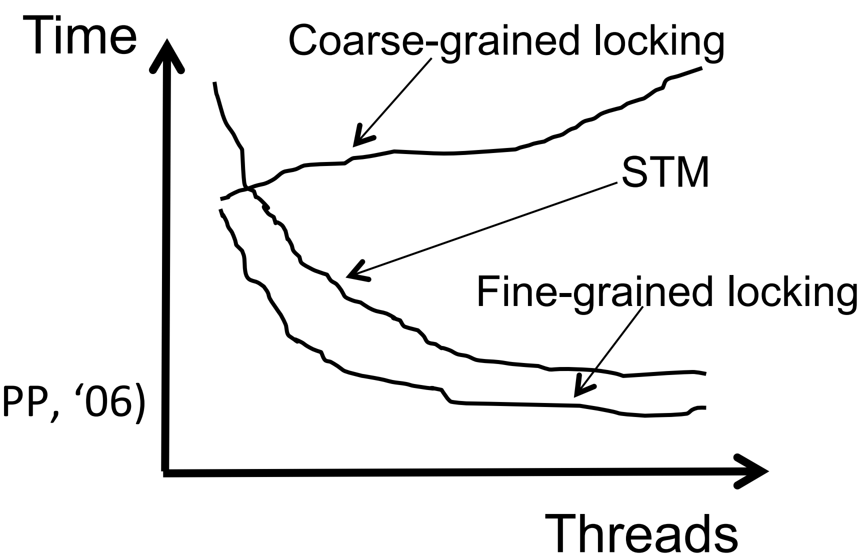



```
public boolean add(int item) {
  head.lock();
  Node pred = head;
  try {
    Node curr = pred.next;
    curr.lock();
    try {
      while (curr.val < item) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        return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

Research Group

Transactional memory promises to alleviate these difficulties

- Similar to ACID transactions
- Easier to program
- Decent performance
- Composable



(PPoPP, '06)

```
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;
    }
  }
}
```

Herlihy and Moss, '93

Transactions can be nested for better composability, performance, …

```
@Atomic{
  @Atomic{
    Account src = getAccount(a_src);
    int b_src = getBalance(src);
    setBalance(b_src - X);
  }
  @Atomic{
    Account dst = getAccount(a_dst);
    int b_dst = getBalance(dst);
    setBalance(b_dst + X);
  }
}
```

Sub-transaction

Sub-transaction

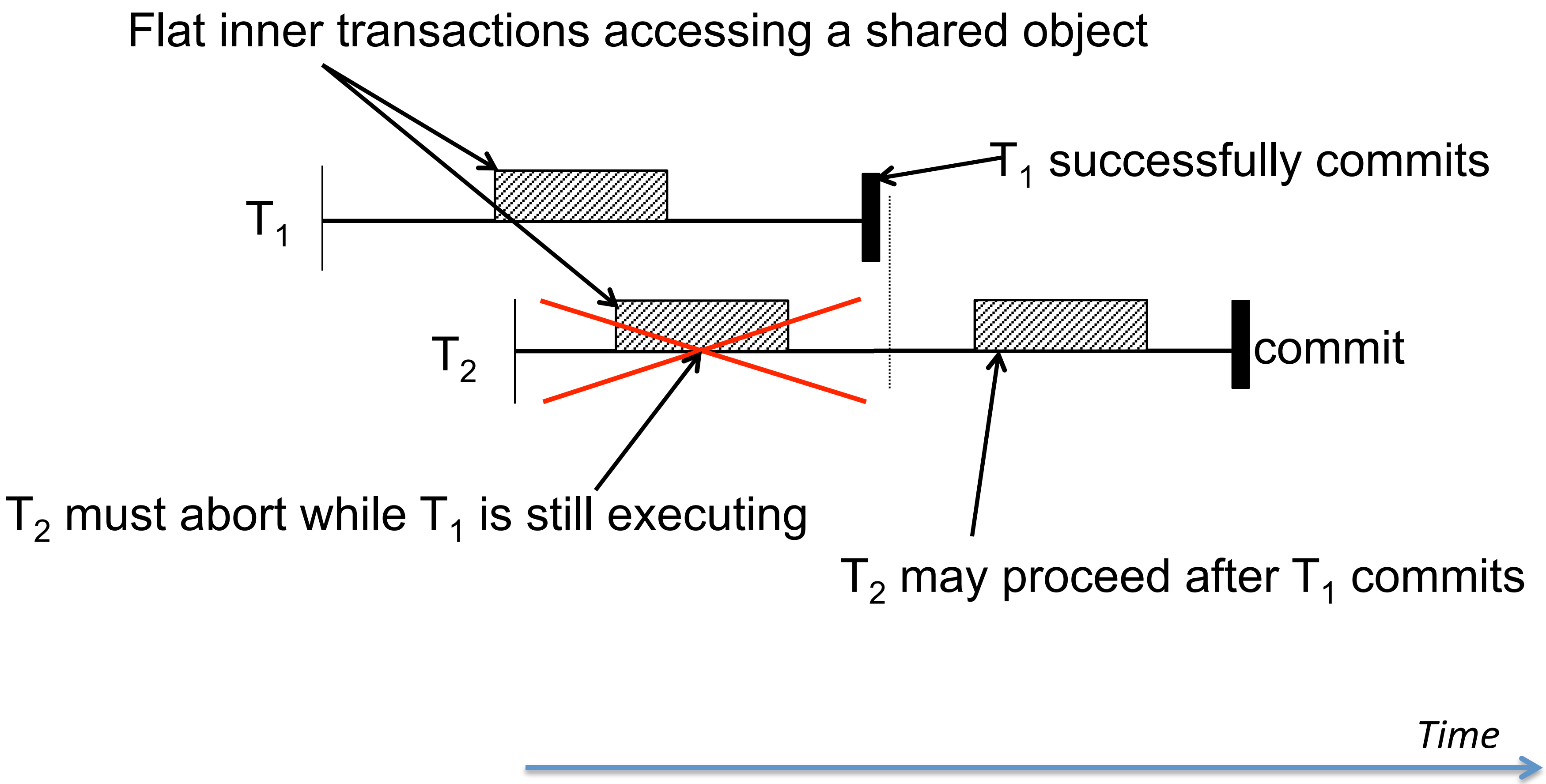
root-transaction

(Moss and Hosking, '06)

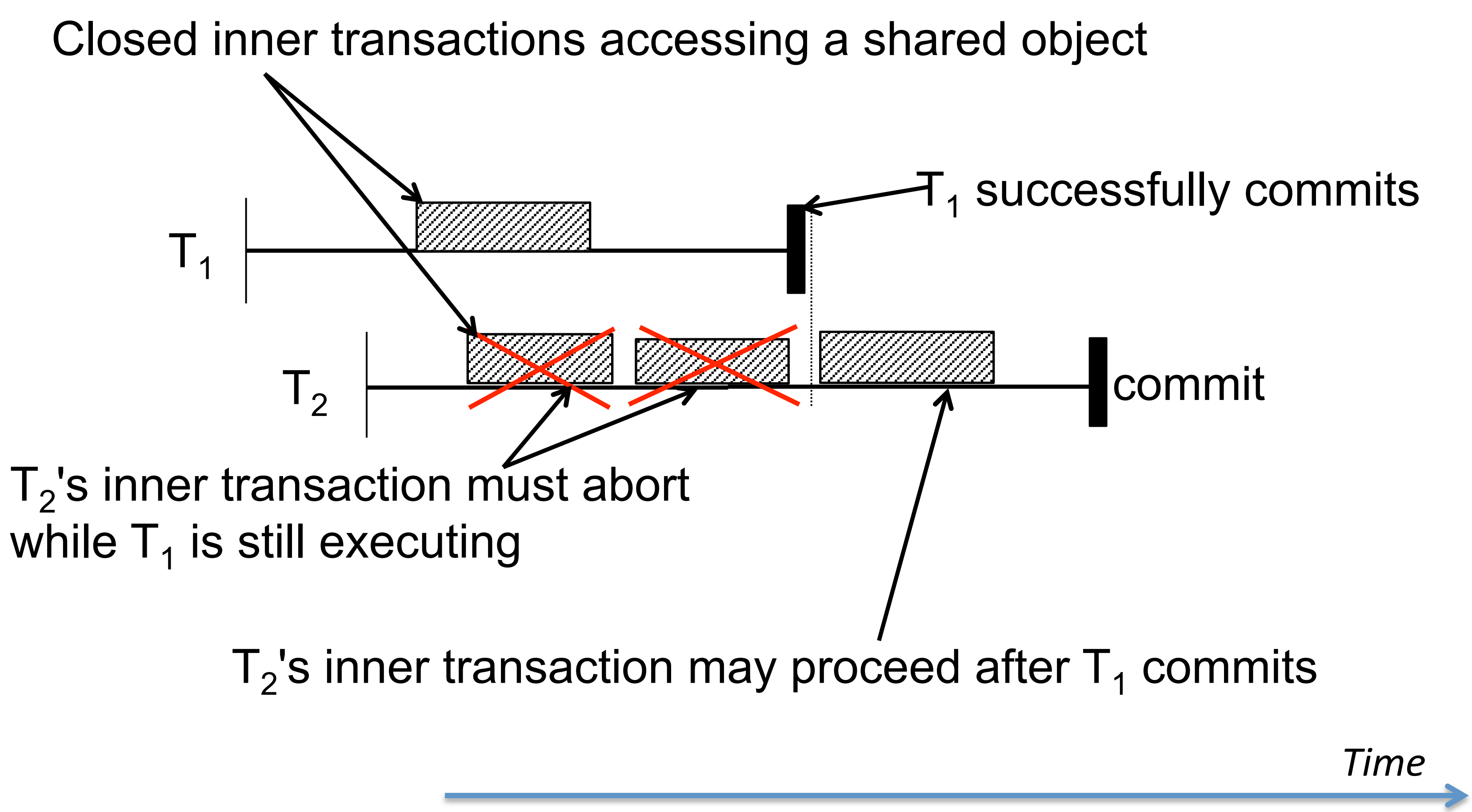
Several nesting models exist

- Flat nesting
- Closed nesting
- Open nesting

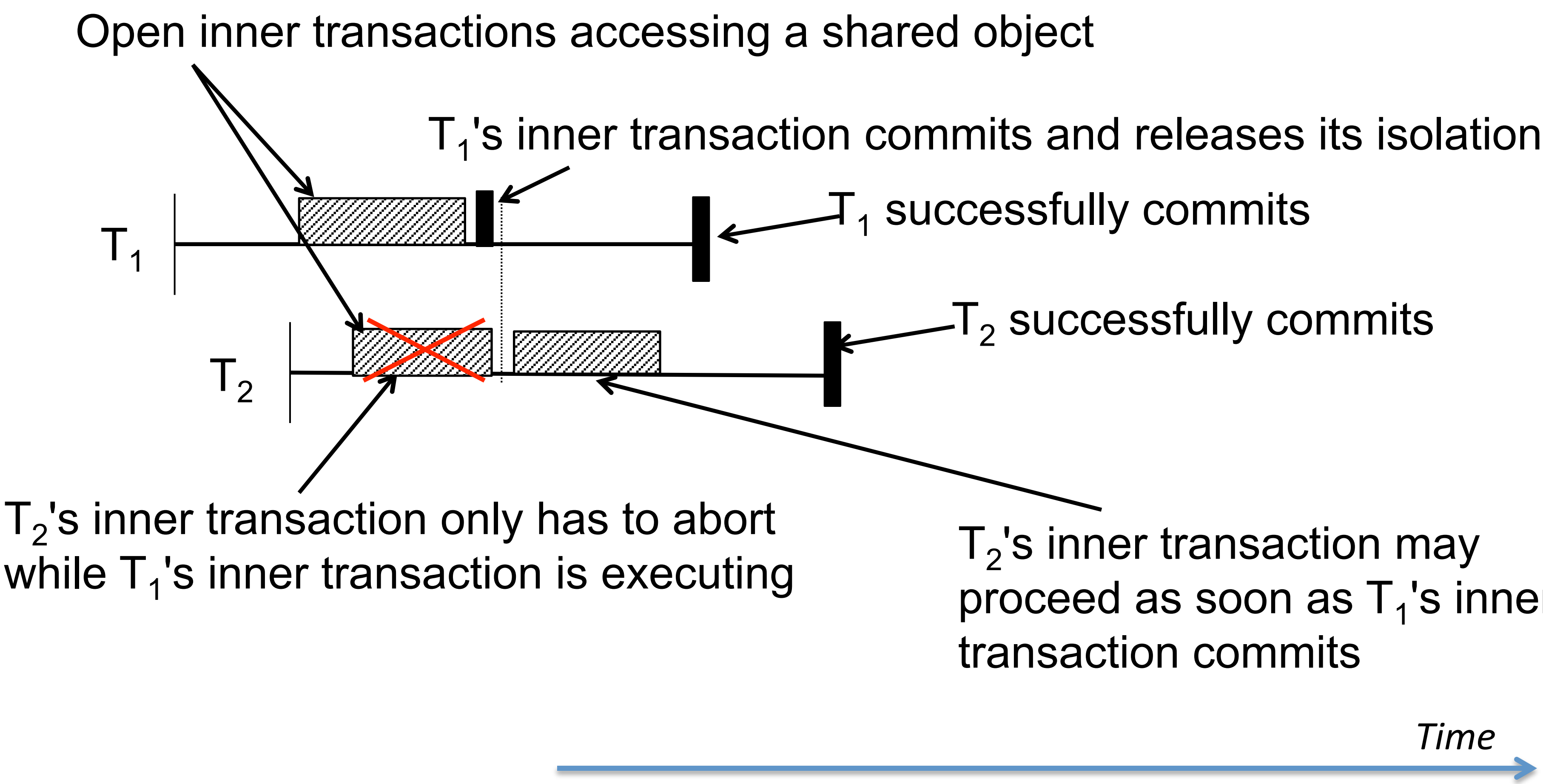
Flat nesting is no nesting



Closed nesting may improve performance



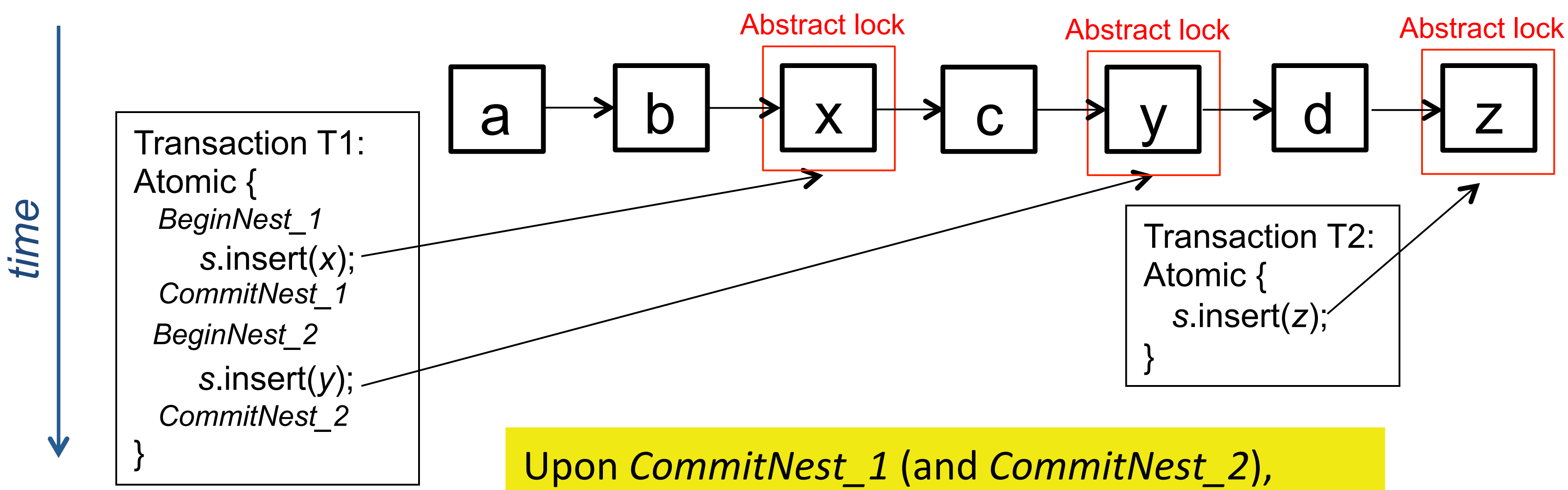
Open nesting may perform even better, at the expense of physical serializability



Open nesting reduces false conflicts and yields *abstract* serializability

- T1 and T2 can execute and commit concurrently iff x ≠ y ≠ z
- But T1 and T2 traverse same physical structure => physical conflict
  - *False conflict*

```
Shared set s;
Transaction T1:        Transaction T2:
Atomic {               Atomic {
  s.insert(x);           s.insert(z);
  s.insert(y);         }
}
```



Upon *CommitNest_1* (and *CommitNest_2*), read-set is released and abstract locks acquired
No conflicts on *a* ,*b*, *c*, *d*, but only on *x*, *y*

Paper's focus is on closed nesting

- If there is a conflict on accessing m_3:
  - flat nesting will restart from **T_flat**
  - closed nesting will restart **T_closed,** saving operations on m_1 and m_2
- Root's commit will likely succeed
- Gains can be significant in distributed systems
  - Object lookup involves network communications

```
// Matrices: m1, m2, m3
@Atomic{ // T_flat
        m1 = getObj(m1_Obj);
        m2 = getObj(m2_Obj);
        m3 = getObj(m3_Obj);
        intm = add(m1,m2);
  @Atomic{ // T_closed
        result = add(intm,m3);
  }
}
```

But sub-transactions have to be programmer-defined

- Step backwards!
  - Reduces TM's high programmability
- Closed nesting enables partial abort in TM, potentially increasing performance

- Is it possible to automate the definitions of closed nested transactions?
  - Increases TM performance, retaining high programmability

Contribution is ACN

- Automatic framework for composing closed nested transactions
  - Completely programmer-transparent
  - Heuristic algorithm
- Dynamic framework
  - Optimize (closed-nested) transaction definition at run-time to adapt to transactional contentions and workload fluctuations
  - (Non-trivial to do so manually)

Multiple factors affect performance of closed-nested transactions

- Nesting granularity
  - # operations performed by a sub-transaction
- Contention
  - Shared objects accessed by a sub-transaction
- Lexical position
  - Each sub-transaction's position in root

```
@Atomic{
    branch1 = getObject(branchId1);
    branch2 = getObject(branchId2);
    branch1.withdraw(amt1);
    branch2.deposit(amt2);
    account1 = getObject(accountId1);
    account2 = getObject(accountId2);
    account1.withdraw(amt1);
    account2.deposit(amt2);
}
```

Bank benchmark's transaction (flat nesting)

Granularity impacts performance

Finest granularity: wrap each operation as a sub-transaction

Coarse granularity: wrap all operations as one sub-transaction

```
@Atomic{
    branch1 = getObject(branchId1);
    branch2 = getObject(branchId2);
    branch1.withdraw(amt1);
    branch2.deposit(amt2);
    account1 = getObject(accountId1);
    account2 = getObject(accountId2);
    account1.withdraw(amt1);
    account2.deposit(amt2);
}
```

NO PARTIAL ABORT!

```
@Atomic{
  @Atomic{
    branch1 = getObject(branchId1);
  }
  @Atomic{
    branch2 = getObject(branchId2);
  }
  @Atomic{
    branch1.withdraw(amt1);
  }
  @Atomic{
    branch2.deposit(amt2);
  }
…
}
```

INEFFECTIVE!

Grouping objects with similar access probability affects performance

System hot spots: *branch1, branch2*
Objects less contended: *account1, account2*

```
@Atomic{
  @Atomic{
    branch1 = getObject(branchId1);
    account1 = getObject(accountId1);
    …
  }
  @Atomic{
     branch2 = getObject(branchId2);
     account2 = getObject(accountId2);
     …
   }
}
```

INEFFECTIVE!

System hot spots: *branch1, branch2*
Objects less contended: *account1, account2*

```
@Atomic{
  @Atomic{
    branch1 = getObject(branchId1);
    branch2 = getObject(branchId2);
    …
  }
  @Atomic{
     account1 = getObject(accountId1);
     account2 = getObject(accountId2);
     …
   }
}
```

EFFECTIVE!

Lexical scoping of sub-transactions also affects performance

System hot spots: *branch1, branch2*
Objects less contended: *account1, account2*

```
@Atomic{
  @Atomic{
     branch1 = getObject(branchId1);
     branch2 = getObject(branchId2);
…
  }
  @Atomic{
     account1 = getObject(accountId1);
     account2 = getObject(accountId2);
     …
   }
}
```

INEFFECTIVE!

System hot spots: *branch1, branch2*
Objects less contended: *account1, account2*

```
@Atomic{
  @Atomic{
    account1 = getObject(accountId1);
    account2 = getObject(accountId2);
    …
  }
  @Atomic{
     branch1 = getObject(branchId1);
     branch2 = getObject(branchId2);
     …
   }
}
```

EFFECTIVE!

Algorithm composes sub-transactions from code blocks

- Transactional code is composed of *UnitBlocks*
  - Smallest logical unit of code involving only one object
  - Includes all local computations on object

```
@Atomic{
    branch1 = getObject(branchId1 );
    branch2 = getObject(branchId2 );
    branch1.withdraw(amt1);
    branch2.deposit(amt2);
    account1 = getObject(accountId1);
    account2 = getObject(accountId2);
    account1.withdraw(amt1);
    account2.deposit(amt2);
}
```

⇒

```
@Atomic{
    branch1 = getObject(branchId1 );
    branch1.withdraw(amt1);

    branch2 = getObject(branchId2 );
    branch2.deposit(amt2);

    account1 = getObject(accountId1);
    account1.withdraw(amt1);

    account2 = getObject(accountId2);
    account2.deposit(amt2);
}
```

Multiple *UnitBlocks* may be combined to form a *Block*

- *UnitBlocks* are tagged with object contention levels
  - Measured at run-time
- *UnitBlocks* with comparable contention are merged
  - *Block*: smallest executable unit of code

UnitBlock

```
@Atomic{
    branch1 = getObject(branchId1 );
    branch1.withdraw(amt1);
    branch2 = getObject(branchId2 );
    branch2.deposit(amt2);
    account1 = getObject(accountId1);
    account1.withdraw(amt1);
    account2 = getObject(accountId2);
    account2.deposit(amt2);
}
```

⇨

Block

```
@Atomic{
    branch1 = getObject(branchId1 );
    branch1.withdraw(amt1);
    branch2 = getObject(branchId2 );
    branch2.deposit(amt2);
    account1 = getObject(accountId1);
    account1.withdraw(amt1);
    account2 = getObject(accountId2);
    account2.deposit(amt2);
}
```

Blocks are reordered

- Ordered in increasing contention level, from root
  - Ensuring data dependencies
- Safe, as transactions are all-or-nothing

```
@Atomic{
    branch1 = getObject(branchId1 );
    branch1.withdraw(amt1);
    branch2 = getObject(branchId2 );
    branch2.deposit(amt2);
    account1 = getObject(accountId1);
    account1.withdraw(amt1);
    account2 = getObject(accountId2);
    account2.deposit(amt2);
}
```

```
@Atomic{
    account1 = getObject(accountId1);
    account1.withdraw(amt1);
    account2 = getObject(accountId2);
    account2.deposit(amt2);
    branch1 = getObject(branchId1 );
    branch1.withdraw(amt1);
    branch2 = getObject(branchId2 );
    branch2.deposit(amt2);
}
```

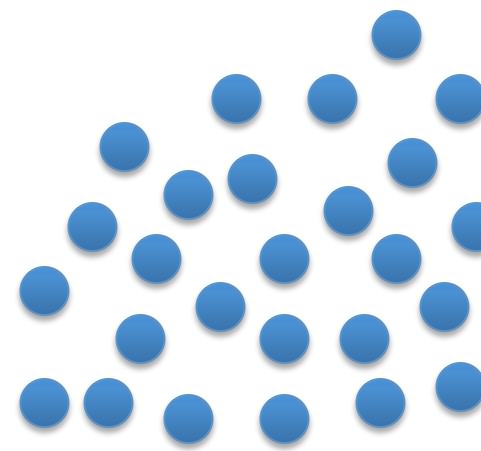
Effectiveness is evaluated at run-time, and recomposed if needed

- Current *Block* sequence is discarded
  - Merged *Block*s are split
- Adjacent dependent *UnitBlock*s with similar contention levels are merged
- *Block*s are sorted in increasing order of (new) contention level

(Difficult to statically optimize, manually)

Case study: distributed TM setting

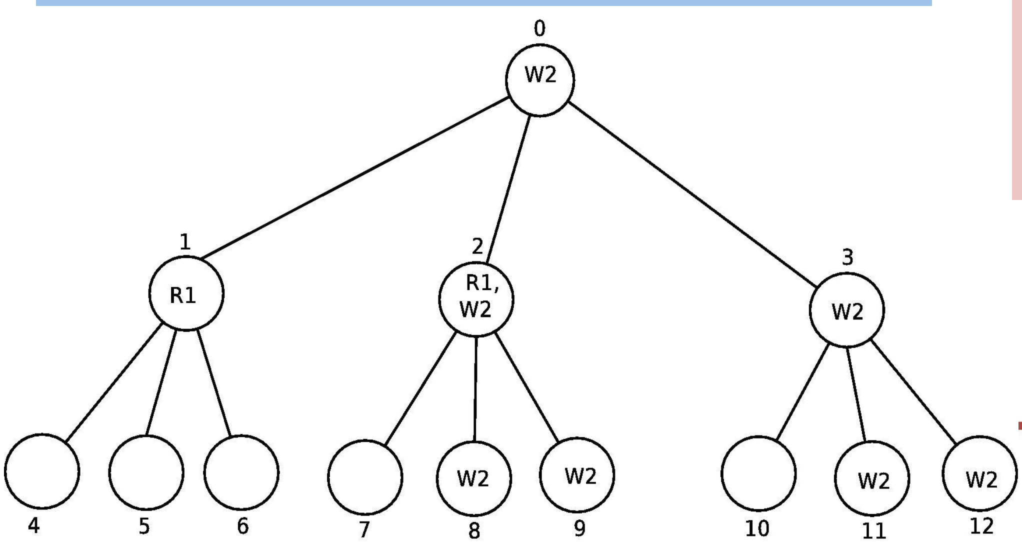
- Distribution has several motivations
  - Exploit locality, fault-tolerance, cope with memory constraints, etc
- If transactions involve remote communications, full aborts are expensive!
- Excellent problem space for evaluating partial abort techniques
  - Closed nesting more effective than checkpointing (Dhoke, '13)



Quorum-based Replication (QR) is base DTM protocol

- Cost of synchronization is higher with replication
  - Exemplified in QR

Nodes logically organized as a tree
Nodes belong to a *read quorum* and/or a *write quorum*
Quorums intersect: any write-q and read-q always intersect



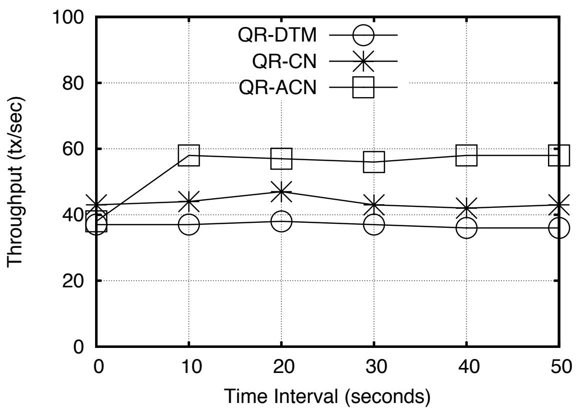
Commit operation:
Contact a write quorum to update new value
Read/write operation:
Contact a read quorum to fetch latest object version

Zhang, '11

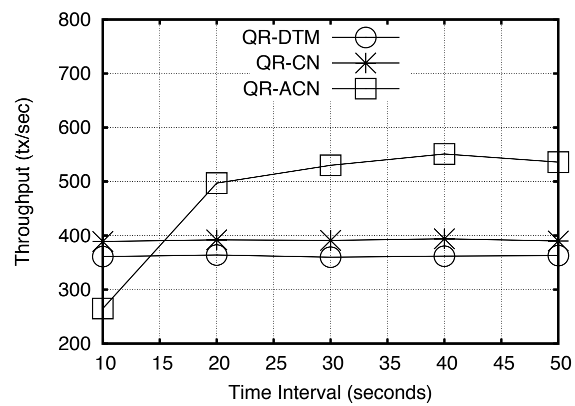
Evaluations used TPC-C and manual closed-nesting as competitor

- Three benchmarks:
  - TPC-C
  - Vacation (from STAMP suite)
  - Bank
- Competitors:
  - QR-DTM (flat nesting)
  - QR-CN (manual closed nesting)
  - QR-ACN (automatic; reconfig every 10secs)
- 30-node private cluster (8-core nodes; 1GBPS link)

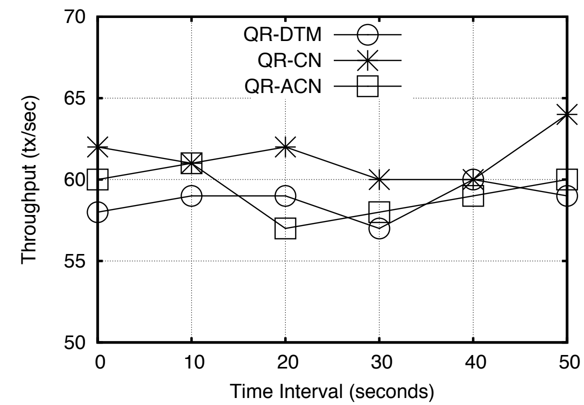
ACN is effective on TPC-C write transactions



100% New Order Transactions



100% Payment Transactions



100% Delivery Transactions

Block containing updates on *District* object is moved to transaction end

District and *Warehouse* objects are most contended; moved closer to transaction end

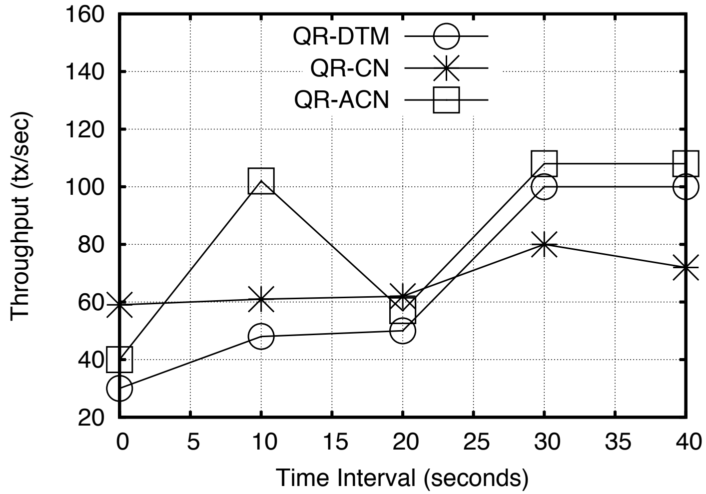
Delivery transaction objects have similar contention; ACN's throughput changes every 10s

Systems Software Research Group

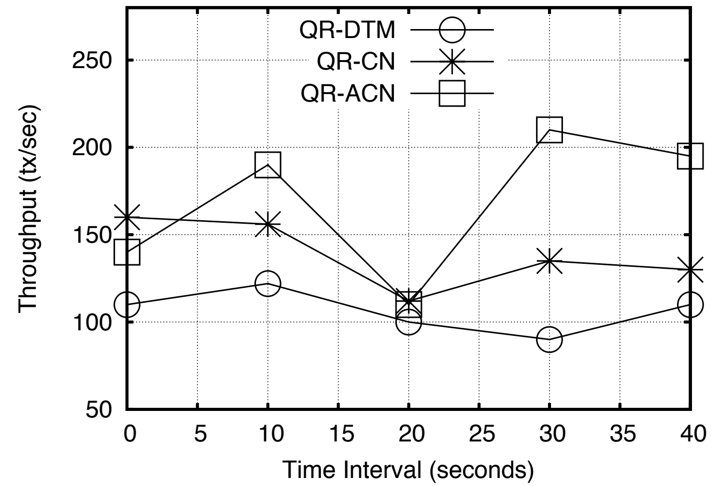
VirginiaTech Invent the Future

ACN also adapts to workload fluctuations

Object contention varied every 20s



STAMP-Vacation



Bank

Manual closed nesting cannot adapt; worse than flat

ACN is always best. Even if most contended *Branches* are changed every 20secs, their contention is still higher than *Accounts*'

Closed nested transactions can be auto-composed, with effective performance

- Lightweight technique for partial aborts
- Manual composition reduces programmability
- Automation
  - Is possible (and works!)
  - Can run-time optimize to adapt to workload changes
  - Is particularly effective in distributed settings
- Code available at hyflow.org

- Auto-compose open-nesting?