On Reducing False Conflicts in Distributed Transactional Data Structures*

Aditya Dhoke, Roberto Palmieri, Binoy Ravindran

Systems Software Research Group

Virginia Tech

**Appeared as short paper in Middleware'13*

ICDCN 2015

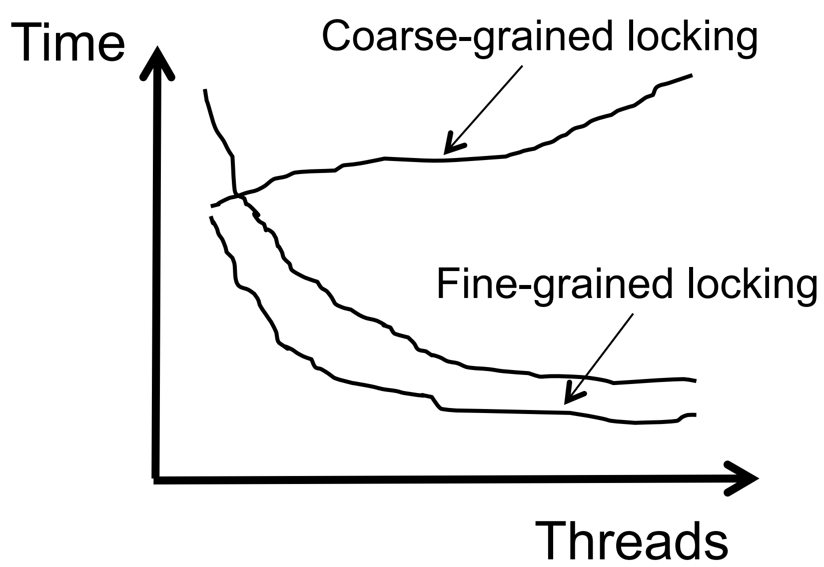
Motivation: concurrent data structures

Wide use in multithreaded programming

```
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();
  }
}
```

Set with APIs:

- add(x)
- remove(x)
- contains(x)



What if you need composability?

Transactional data structures?

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
}
```

Transactional data structures?

```
Shared data: concurrentList

atomicFoo()
{
        concurrentList.add(x);
        concurrentList.add(y);
}
```

- Compose multiple operations to form a transaction (with transactional properties)

Example deux

```
Shared data: concurrentList1
Shared data: concurrentList2

atomicFoo()
{
        concurrentList1.remove(x);
        concurrentList2.add(x);
}
```

A possible solution:
use software transactional memory

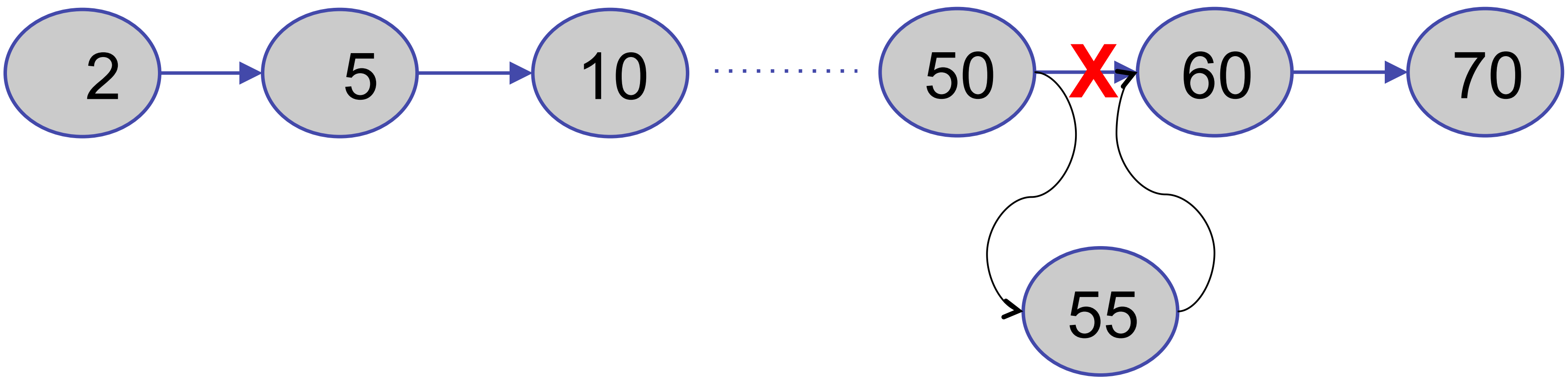
```
Shared data: sequentialList

@Atomic
atomicFoo()
{
        sequentialList.add(x);
        sequentialList.add(y);
}
```

- Works! But poor performance
  - STM is a general framework
  - Data structures will suffer from “false conflicts”

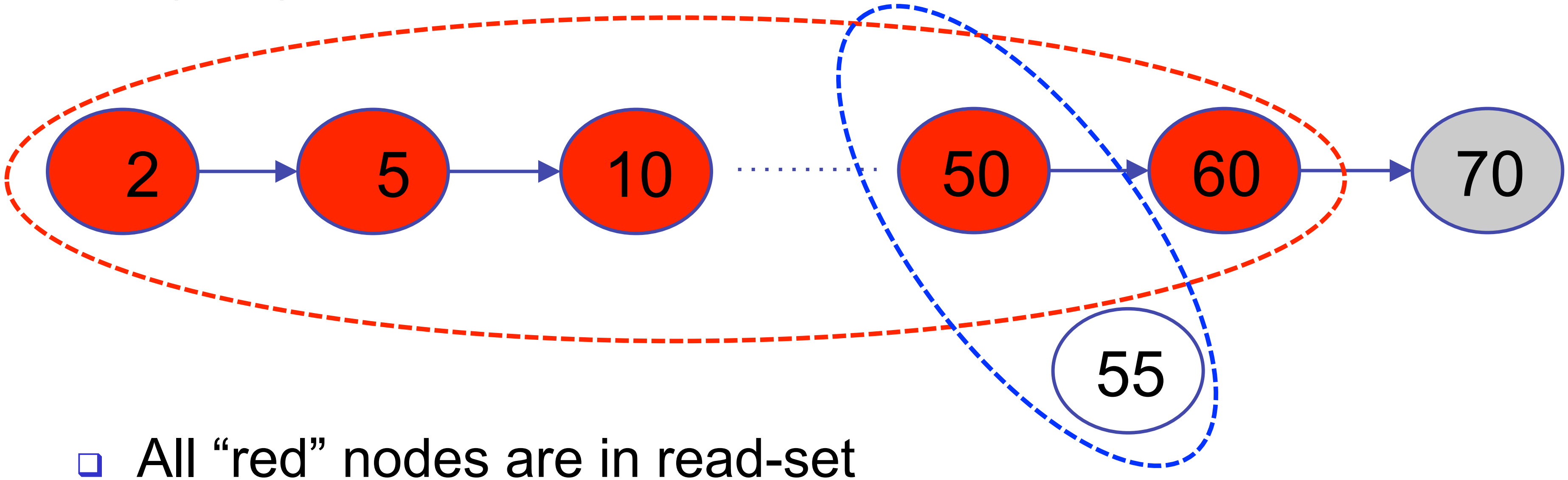
False conflict example: linked-list

add("55")



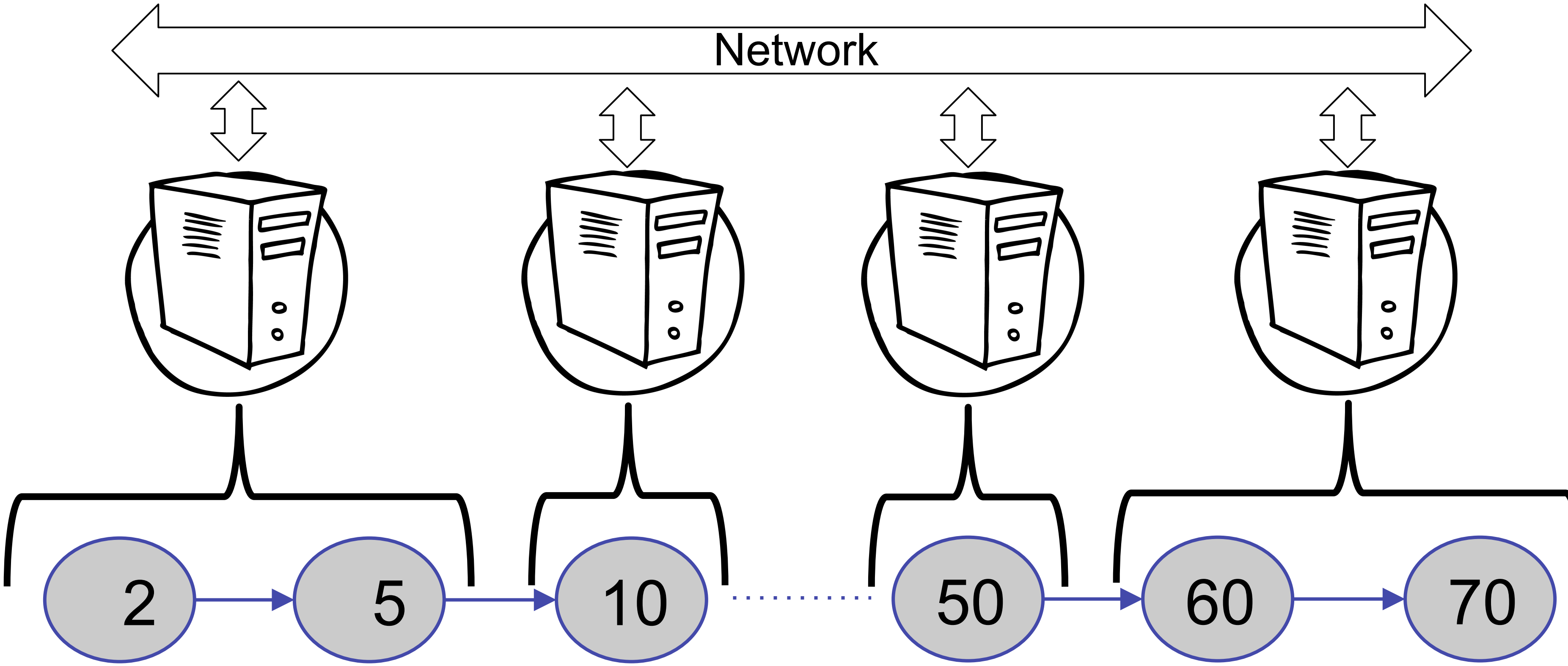
False conflict example: linked-list

add(“55”)



- All “red” nodes are in read-set
- “50” and “55” are in write-set
- If a concurrent transaction deletes “5”, STM will detect a conflict; will abort and retry
  - Even though add(“55’) and remove (“5”) commute
  - False conflict

If transactions involve remote communications, false conflicts (significantly) degrade performance



- Data structure may be distributed (e.g., partitioned, replicated)
  - To exploit locality
  - Cope with memory constraints
  - For fault-tolerance

Objective: reduce impact of false conflicts in distributed transactional data structures

- Three techniques

- QR-ON
  - Exploit Open Nesting [Moss, ‘06] in a distributed setting
  - Inner transactions commit globally and release objects; not validated during final commit

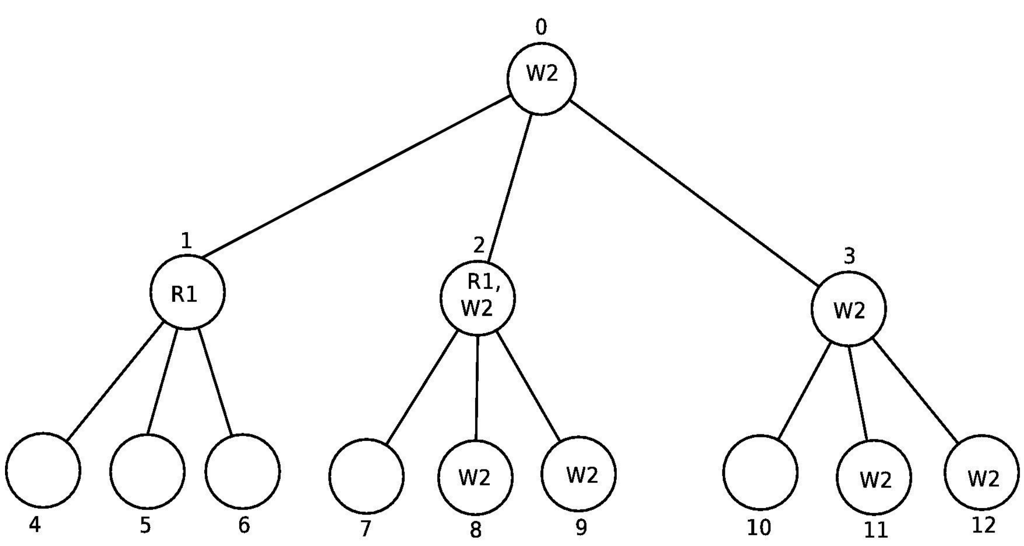
- QR-OON
  - Optimistic Open Nesting: reduce commit cost through non-blocking commit; next transaction executes speculatively

- QR-ER
  - Early release of objects not affecting transaction semantics

Quorum-based Replication (QR) [Zhang, '11] is base protocol

Motivation: cost of synchronization is higher with replicated data (QR exemplifies this)

- Nodes logically organized as a tree
- Nodes belong to a *read quorum* and/or a *write quorum*



- Commit operation:

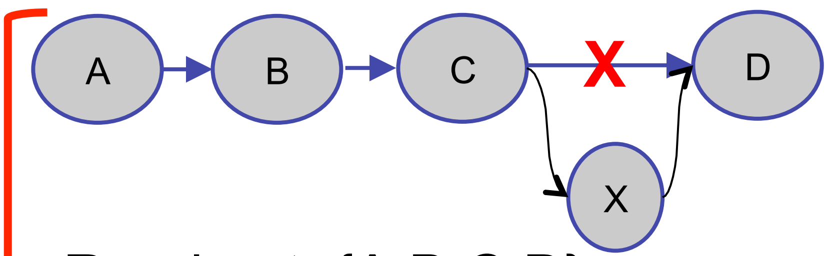
  Contact a write quorum to update new value
- Read/write operation:

  Contact a read quorum to fetch latest object version

QR-ON:
QR + Open Nesting

- Divide transaction into multiple sub-transactions
  - Sub-transaction's commit is globally visible
- Acquire abstract locks to serialize non-commutative operations
- Reduced false conflicts (but not eliminated)
- (On abort, fire compensations for committed sub-transactions)

```
atomicFoo()
{
    List.add(x);
    var = List.contains(x);
    If (var)
        List.add(z);
    else
        List.add(y);
}
```



Read-set: {A,B,C,D}

Write-set: {C,X}

Commit sub-transaction

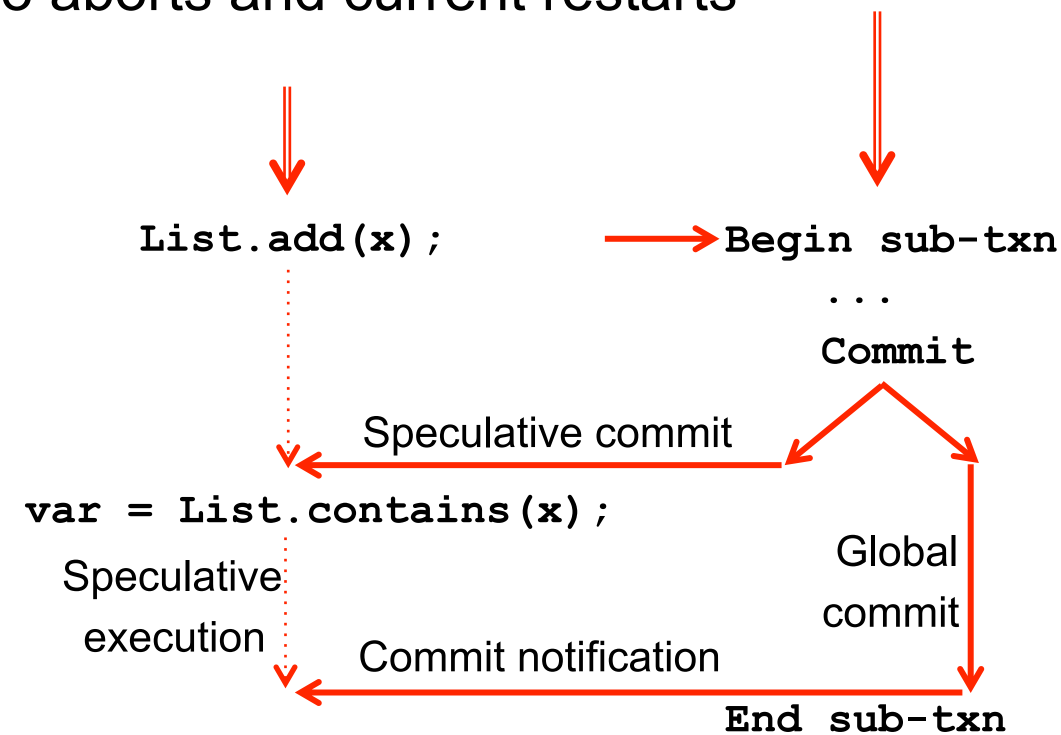
Read-set: {}   Abs Lock: {X}

Write-set: {}

QR-OON:
QR + Optimistic Open Nesting

- QR-ON reduces false conflicts, but at higher commit costs
- Reduce by asynchronous commit of current inner transaction
- Next inner transaction reads speculatively
- If current commits, next continues its execution
- If current aborts, next also aborts and current restarts

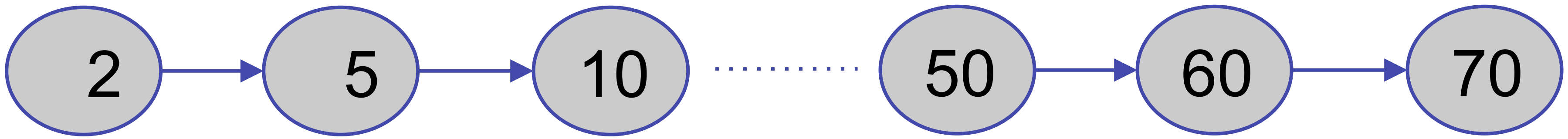
```
atomicFoo()
{
    List.add(x);
    var = List.contains(x);
    If (var)
        List.add(z);
    else
        List.add(y);
}
```



QR-ER:
QR + Early Release

- Does not use nested transactions
- Requires programmer to:
  - define data structure's semantics
  - identify read objects to release from transaction's read-set
- (Data structure-specific library can be rolled out)

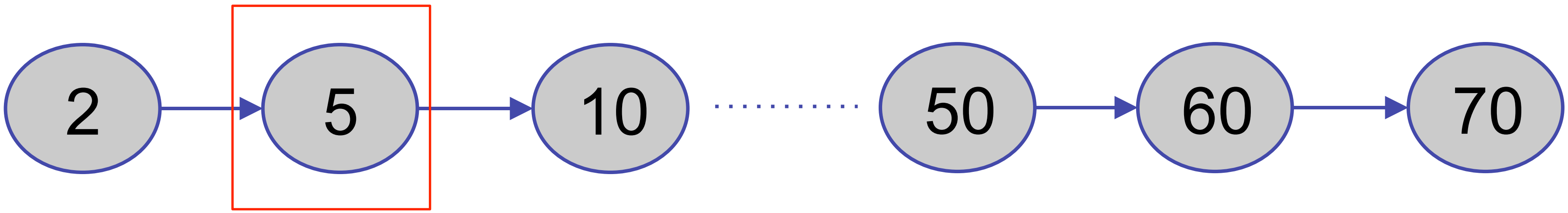
Example: List.add(55)



Read-set: {}

Early Release example

Read-set inclusion conditions for List.add(55)



Would 5 be the successor of 55?

NO

-> No inclusion in Read-set

Would 5 be the predecessor of 55?

NO

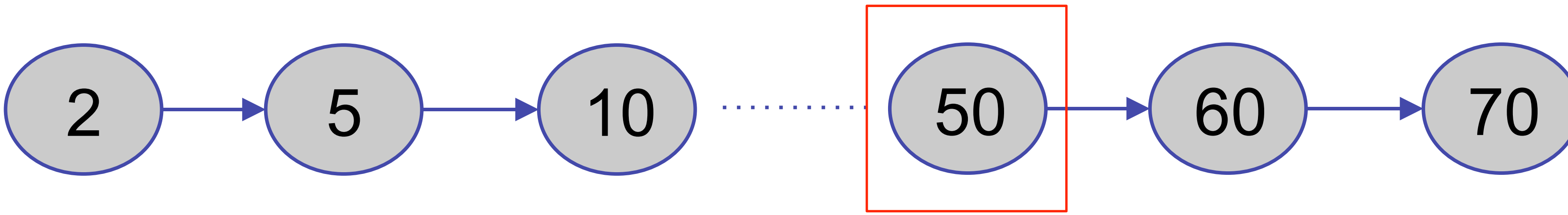
-> No inclusion in Read-set

Read-set: {}

```
add()
{
    while(curr.next < 55){
        if (needToBeIcnluded(curr))
            readSet.get(curr).setValidate(true)
        curr = curr.next;
    }
     . . .
}
```

Early Release example

Read-set inclusion conditions for List.add(55)



Would 50 be the successor of 55?

NO

-> No inclusion in Read-set

Would 50 be the predecessor of 55?

YES

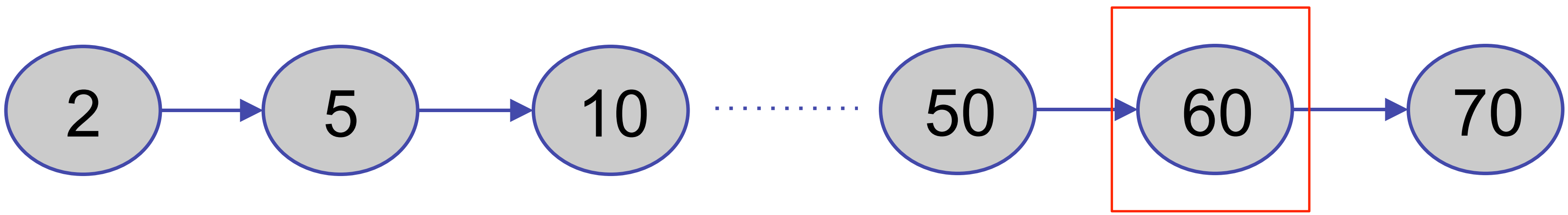
-> Inclusion in Read-set

Read-set: {50}

```
add()
{
    while(curr.next < 55){
        if (needToBeIcnluded(curr))
            readSet.get(curr).setValidate(true)
        curr = curr.next;
    }
     . . .
}
```

Early Release example

Read-set inclusion conditions for List.add(55)



Would 60 be the successor of 55?
YES
-> Inclusion in Read-set

Would 60 be the predecessor of 55?
NO
-> No inclusion in Read-set

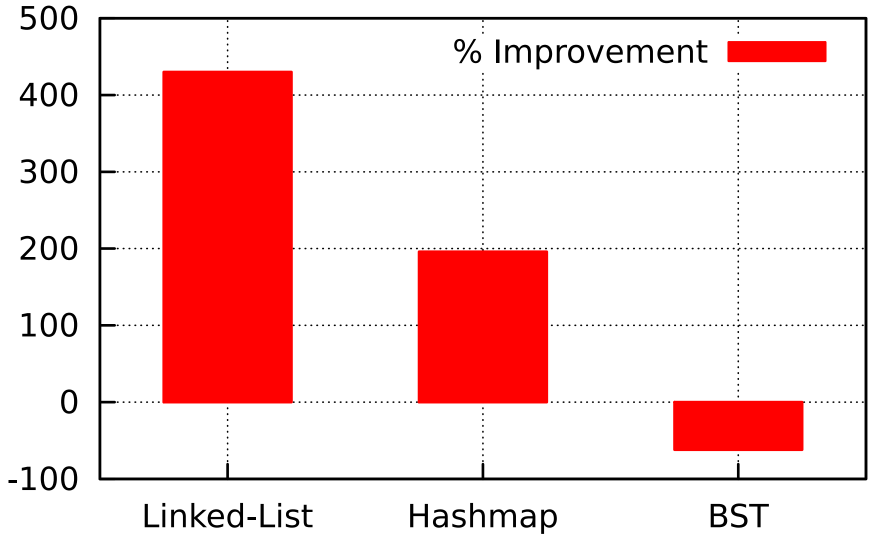
Read-set: {50,60}

```
add()
{
    while(curr.next < 55){
        if (needToBeIcnluded(curr))
            readSet.get(curr).setValidate(true)
        curr = curr.next;
    }
     . . .
}
```

Experimental Study

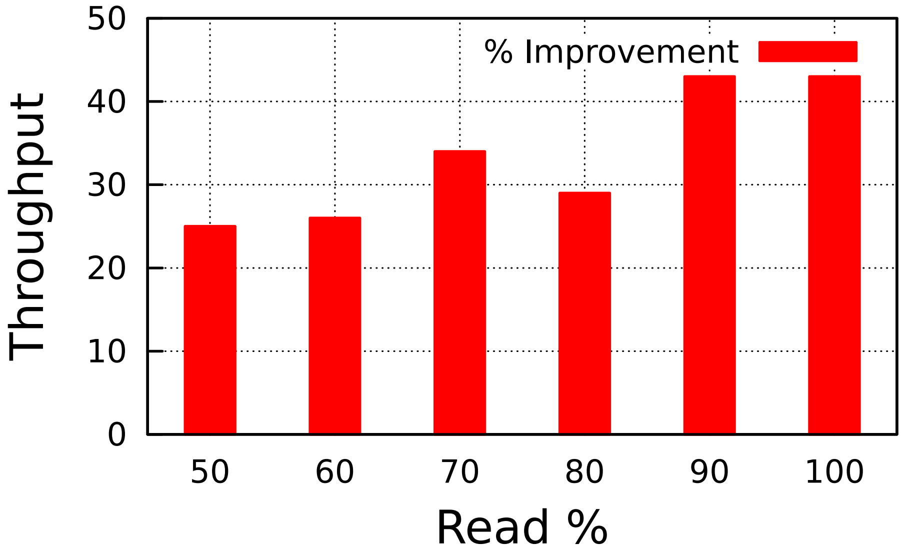
- Private Cluster
- 13 nodes (8 cores each)
- Three data structures:
  - Linked-List
  - Hash-Map
  - BST
- Competitors:
  - QR-DTM
  - QR-ON
  - QR-OON
  - QR-ER

Experimental results: ON and OON are most effective with greater conflicts and read workloads



QR-ON vs QR

Throughput improvement



QR-OON vs QR-ON

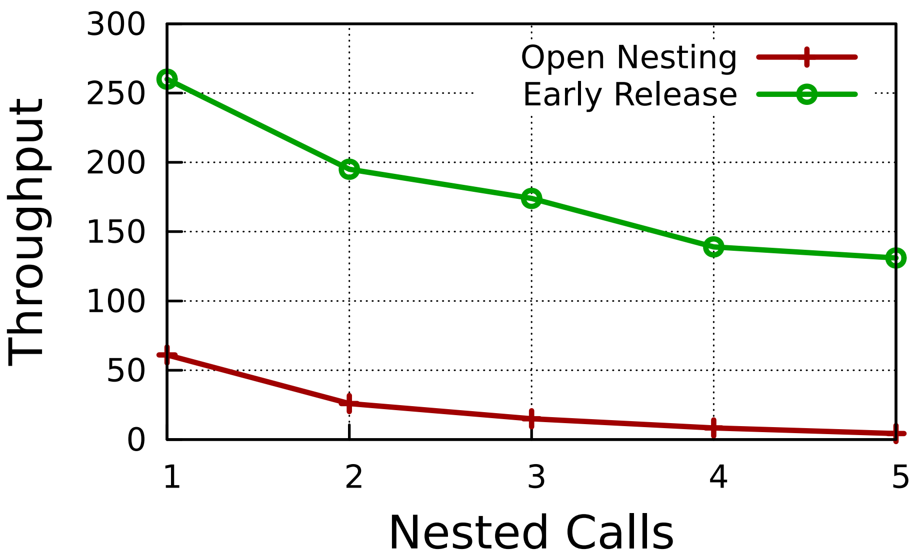
Linked-List

#calls per transaction=5

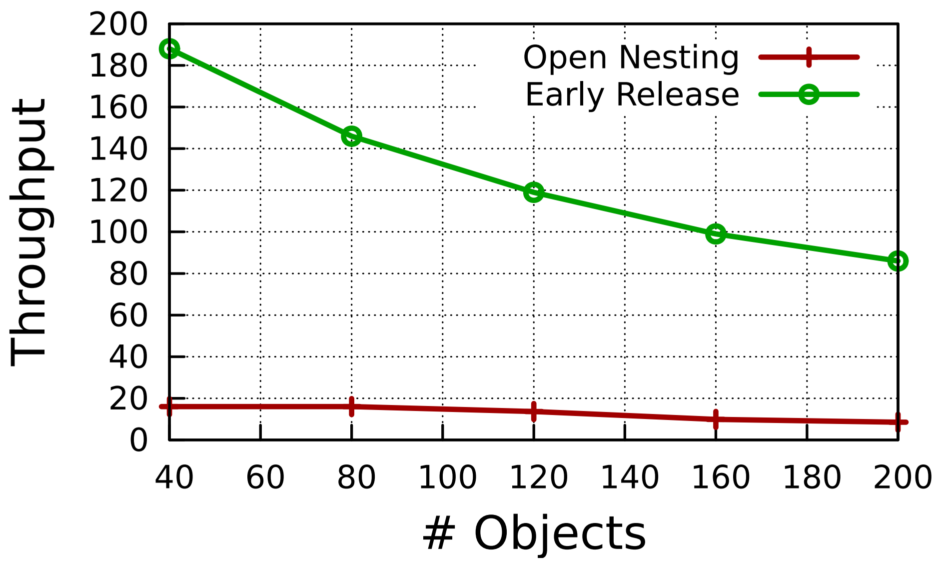
object count=500

Experimental results: ER's gains are significant

- Linked-List benchmark
- One nested operation per nested transaction



Objects = 500



Calls = 3

Conclusions

- Need transactional data structures for composability
- False conflicts degrade performance

- Open nesting reduces false conflicts, does not require heavy programmer’s intervention, but commit cost is high
- Commit cost can be reduced through NB implementation
- Early release involves programmer in identifying precise validation set, but significant performance gain

- Tradeoff between programmability and performance