Scheduling Open-Nested Transactions in Distributed Transactional Memory

Junwhan Kim, Roberto Palmieri, and Binoy Ravindran

Virginia Tech
USA
{junwhan,robertop,binoy}@vt.edu

COORDINATION 2013

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.

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;
}
```

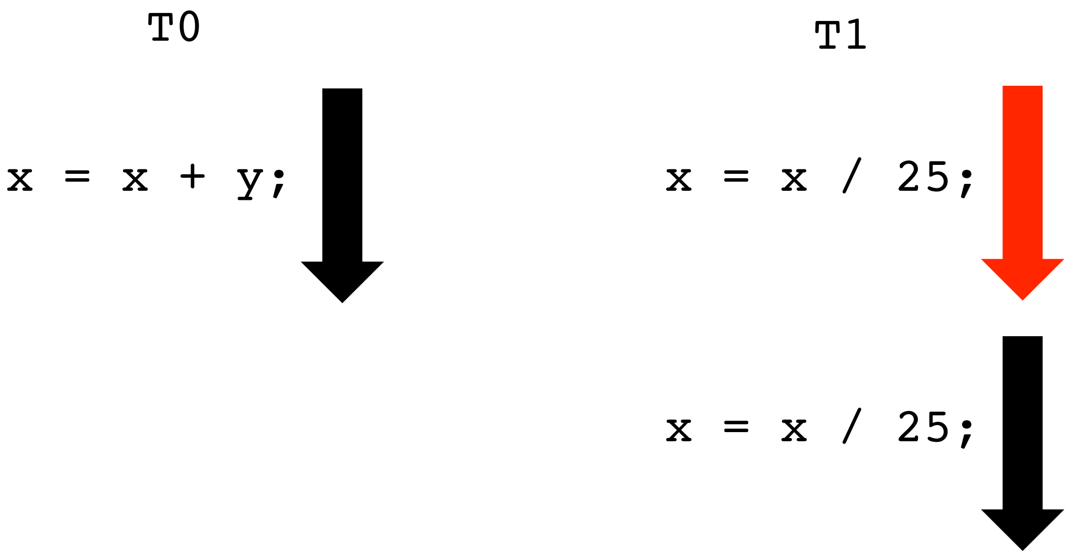
T1

```
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

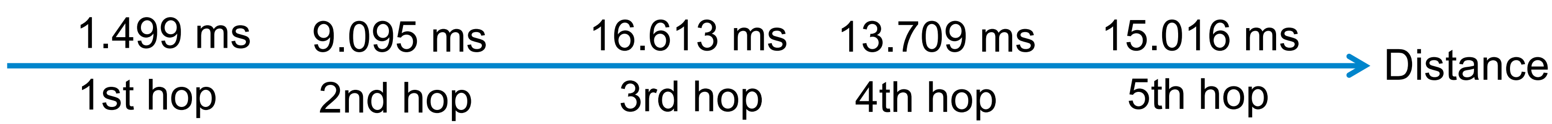
- Contention manager
  - Can cause too many aborts, e.g., when a long running transaction conflicts with shorter transactions
  - An aborted transaction may wait too long

- 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

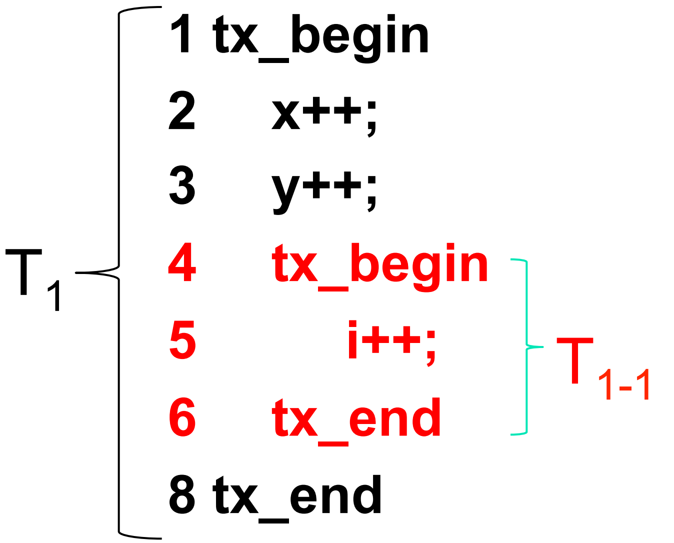


Nested Transactions

- A transaction is nested
  - When it is enclosed within another transaction

- Motivations
  - Make code composability easy
  - Potential for improved performance
  - Fault management

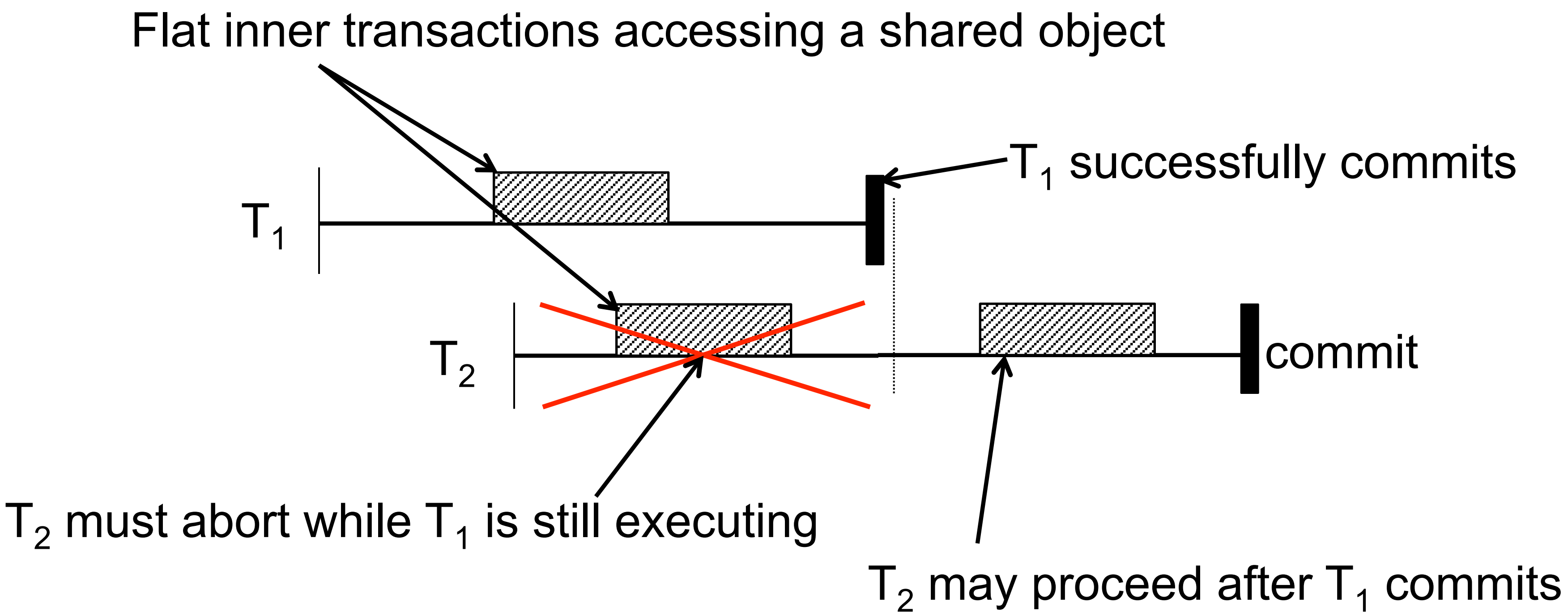
- Three types of nesting models
  - *Flat*, *Closed*, *Open*



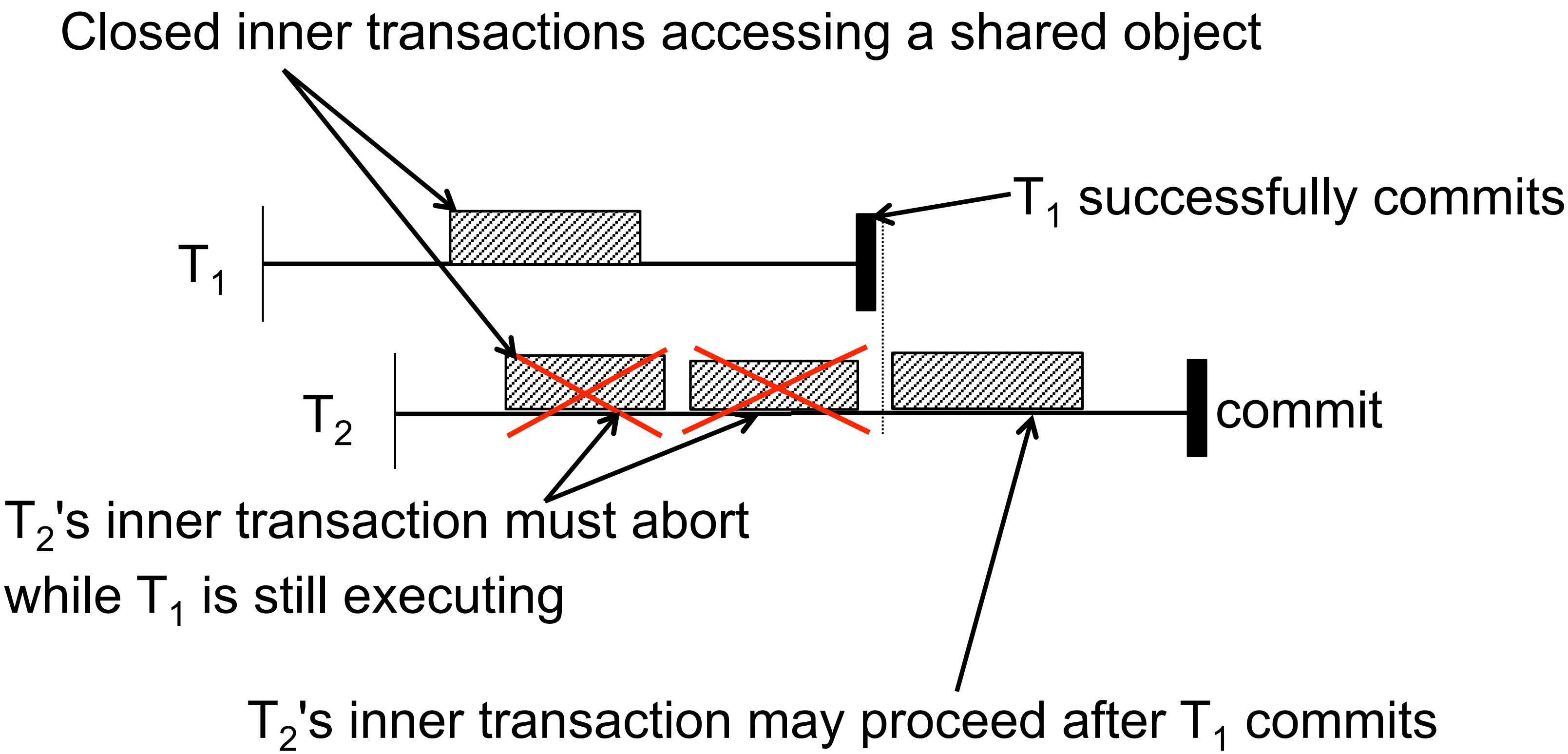
Example of a nested transaction

J. E. Moss (1981). Nested transactions: an approach to reliable distributed computing.

Flat Nested Transactions

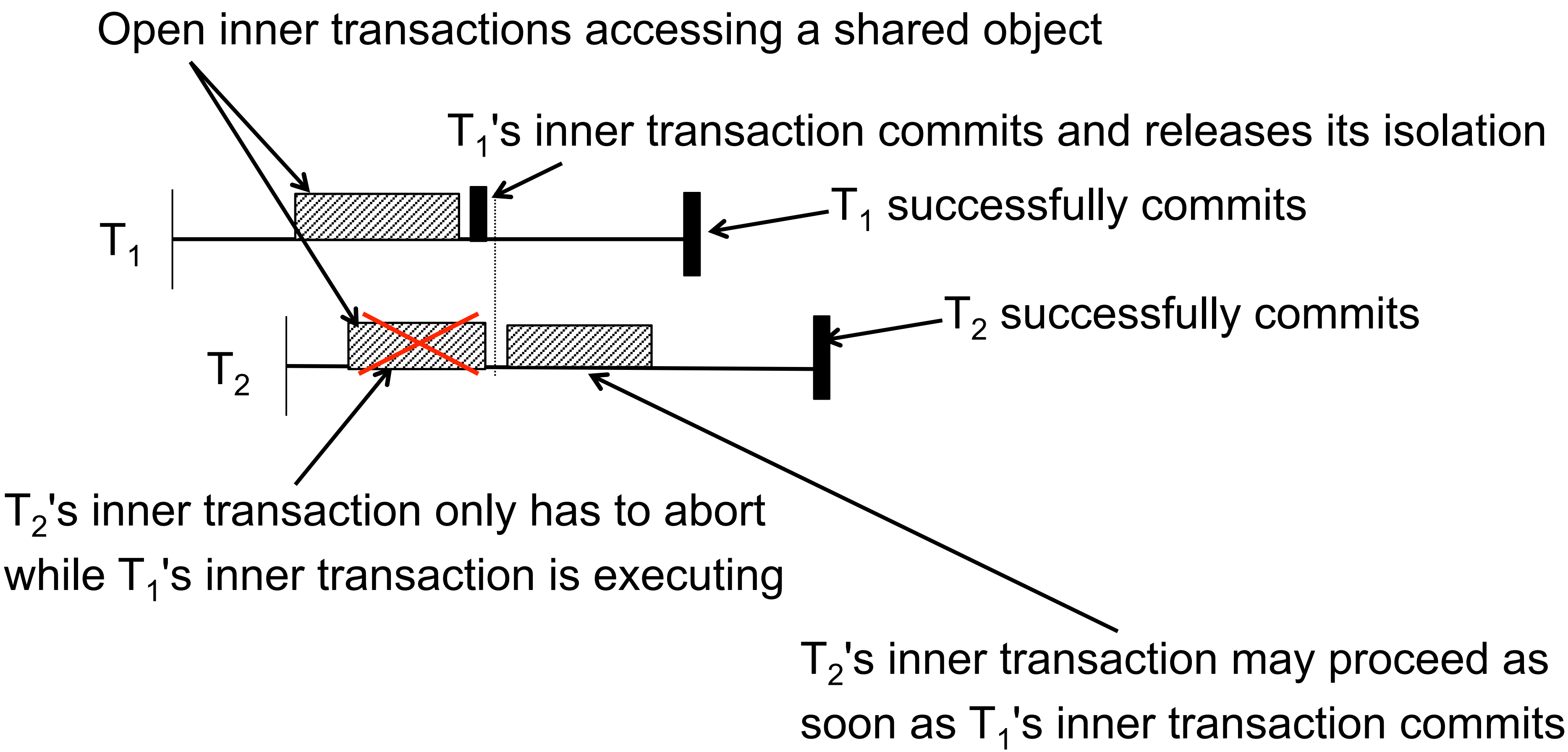


Closed Nested Transactions



A. Turcu and B. Ravindran (2012). On Closed Nesting in Distributed Transactional Memory, *TRANSACT*, pp 1- 10

Open Nested Transactions



A. Turcu and B. Ravindran (2012). On Open Nesting in Distributed Transactional Memory, *SYSTOR*, pp 1- 12

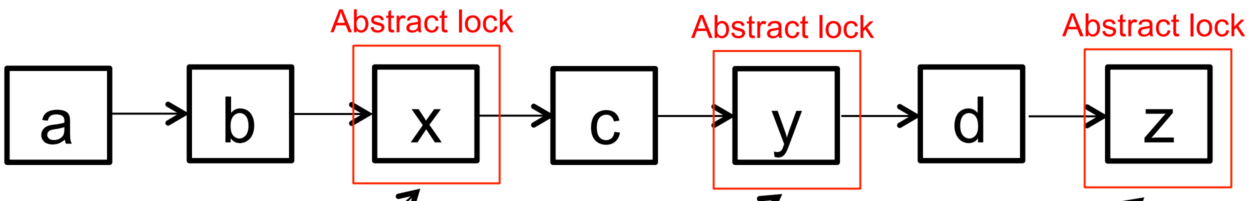
Abstract serializability, abstract locks, and correctness of open nesting

- Multi-level serializability
  - Abstract-level
    - T1 and T2 can execute and commit concurrently iff $x \neq y \neq z$
  - Physical-level
    - T1 and T2 conflicts because both access same physical structure where *x*, *y*, and *z* are stored
    - If $x \neq y \neq z$ *and* physical conflict => **false conflict**

```
Shared set s;
Transaction 1:        Transaction 2:
Atomic {              Atomic {
  s.insert(x);          s.insert(z);
  s.insert(y);        }
}
```

- Abstract locks
  - Abstract locks are acquired on objects in the write-set when an open-nested transaction commits
  - Read-set is immediately released
  - Abstract serialization is broken if readers do not check the abstract lock before accessing an object

Open nesting with abstract locks reduces false conflicts



Transaction 1:
Atomic {
  BeginNest_1
    s.insert(*x*);
  CommitNest_1
  BeginNest_2
    s.insert(*y*);
  CommitNest_2
}

Transaction 2:
Atomic {
  s.insert(*z*);
}

time

- $x \neq y \neq z$ => no conflict at abstract level
- T1 and T2 traverse the same structure => conflict at physical level
- Upon *CommitNest_1* (and *CommitNest_2*), read-set is released and abstract locks are acquired
- No conflicts on *a* ,*b*, *c*, *d*, but only on *x*, *y*

Past research have developed several transactional schedulers

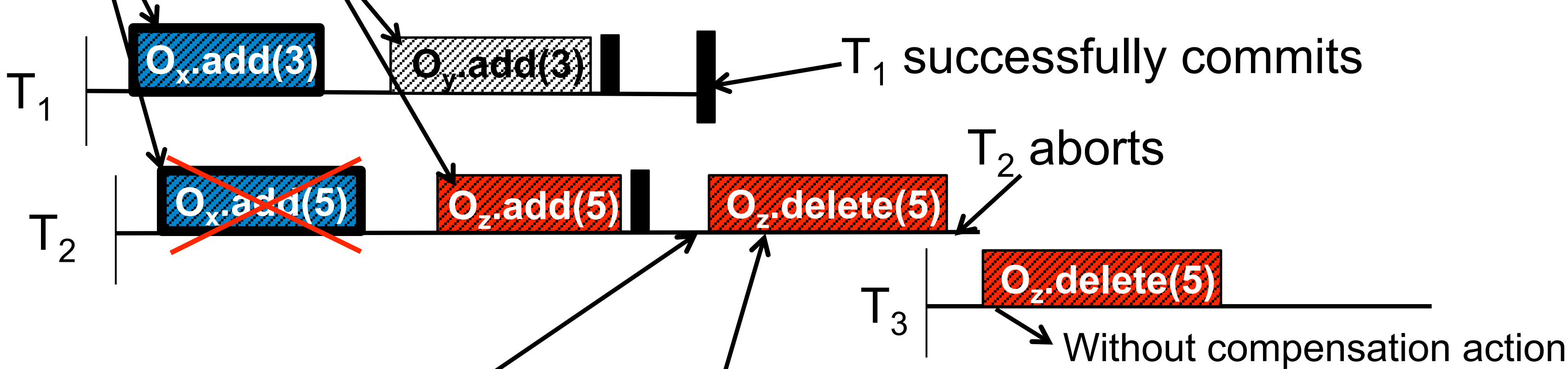
- Multi-core systems
  - BiModal transactional scheduler (OPODIS 09)
  - Proactive transactional scheduler (MICRO 09)
  - Adaptive transactional scheduler (SPAA 08)
  - Steal-On-Abort (HiPEAC 09)
  - CAR-STM (PODC 08)

- Distributed systems
  - Bi-interval transactional scheduler (SSS 10)
    - Flat nested transactions in a single copy model
  - Reactive transactional scheduler (IPDPS 12)
    - Closed nested transactions in a single copy model
  - Cluster-based transactional scheduler (CCGrid 13)
    - Flat nested transactions in a replication model

Motivation

Outer transactions accessing a shared object

Open inner transactions accessing different shared objects commit



Outer transaction aborts and the compensation action of T_2's inner transaction has to be executed, since the modification of T_2's inner transaction has become visible to other transactions

Our goal is to minimize aborts of outer transactions with committed inner transactions (to minimize compensations) through scheduling

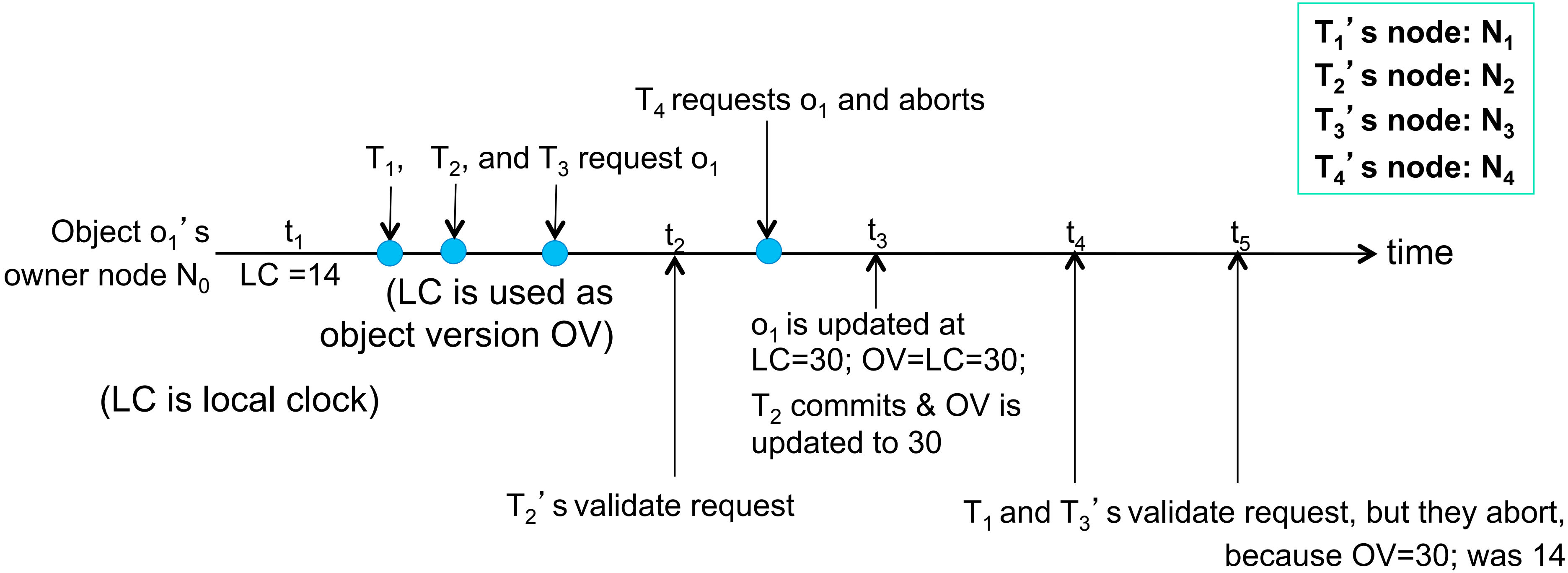
Paper's contribution

- Dependency-Aware Transactional Scheduler (DATS)
  - Minimizes aborts of outer transactions
  - Uses TFA for DTM concurrency control
  - Open-nested transactions are assumed to do operations for which *inverses* are well-defined
    - E.g., *add*(*x*) is inverse of *delete*(*x*)
    - Exists for collection classes
    - Two operations *add*(*x*) and *add*(*y*) are commutative if executing them in either order results in the same behavior
    - True when *x* and *y* are distinct; otherwise not

- Implementation and experimental studies
  - HyFlow DTM framework (hyflow.org)

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

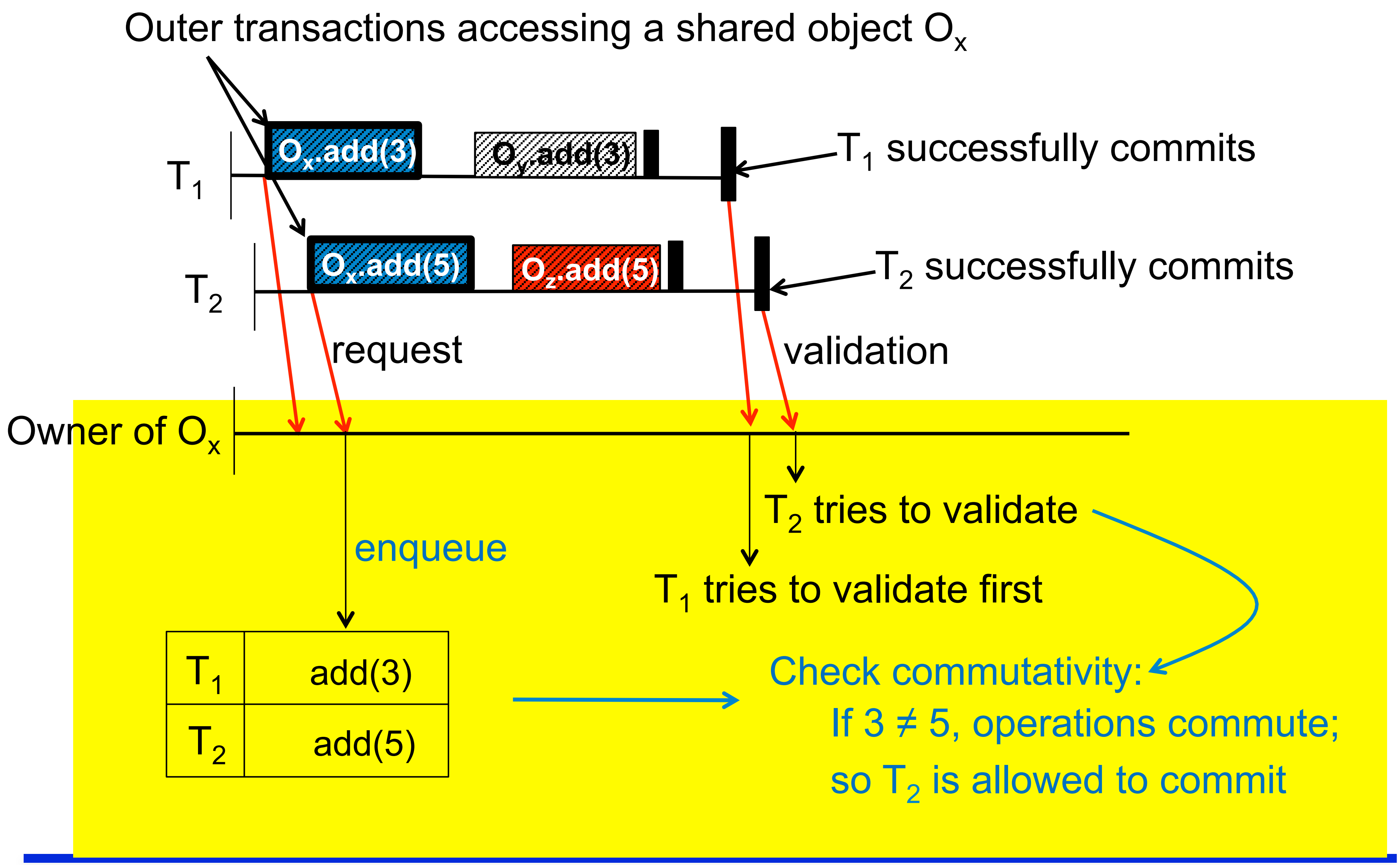
Atomicity, consistency, and isolation in data-flow DTM



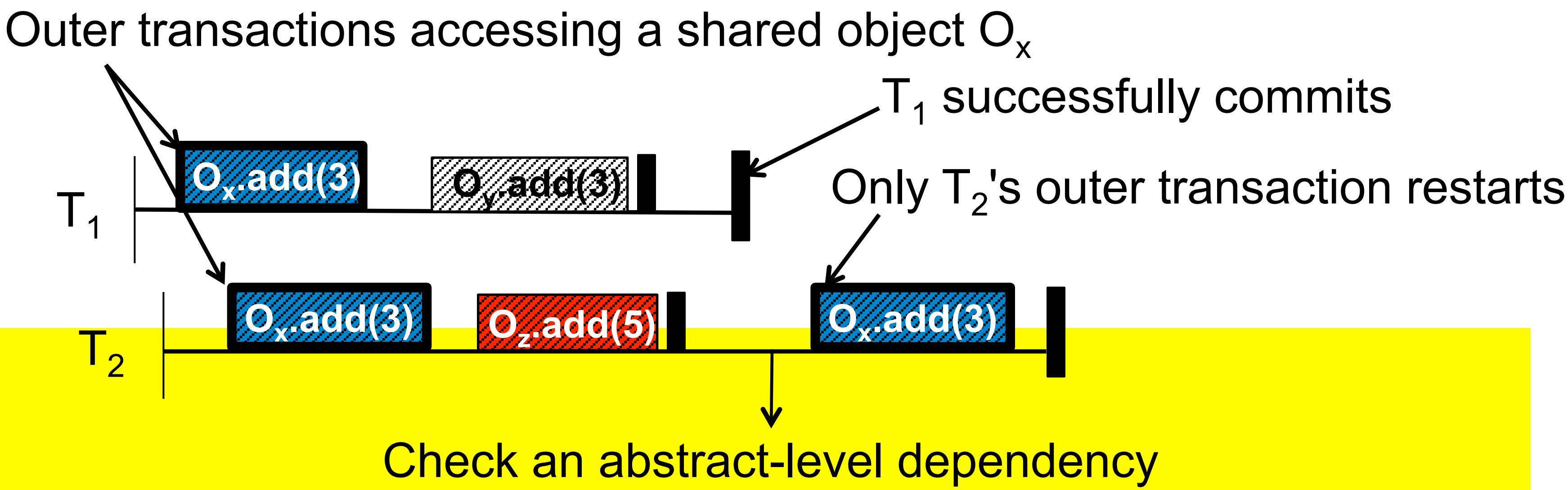
- Transactional Forwarding Algorithm (TFA)
  - Early validation of remote objects (earlier validated commits first)
  - Atomicity for object operations in the presence of asynchronous clocks

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

DATS: checking object level dependency



DATS: checking abstract-level dependency



Independent Case

```
Atomic{
  List ll = request (list2);
  ll. add(3) ;
  ADD(5) ; // inner tx
}
```

Dependent Case

```
Atomic{
  List ll = request (list2);
  deleted = ll. delete(3) ;
  if (deleted) ADD(5) ; // inner tx
}
```

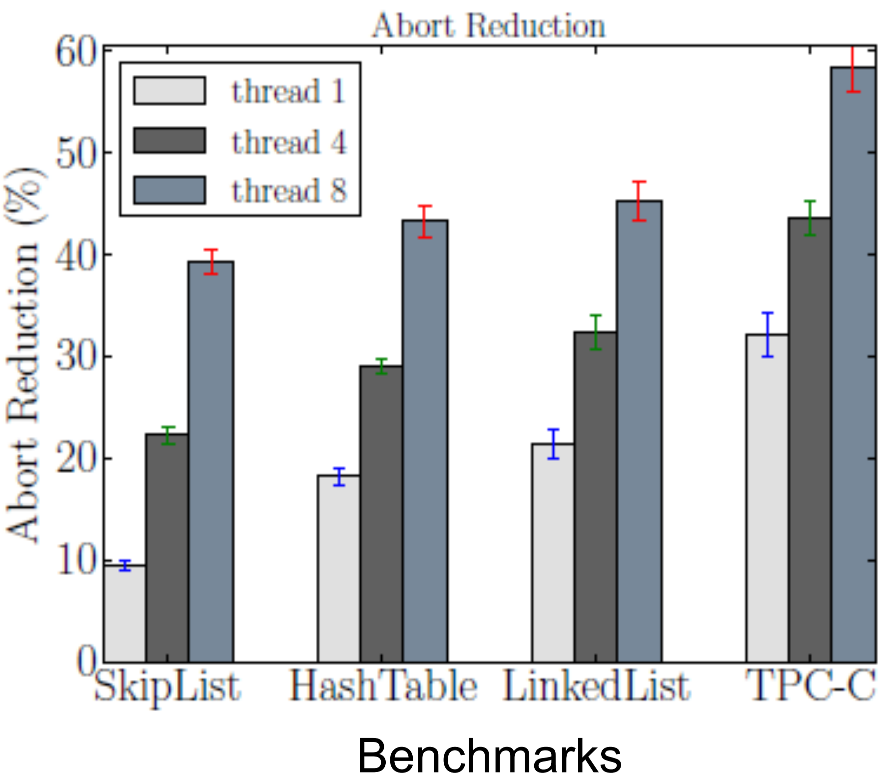
Implementation and experimental setup

- Implemented DATS in HyFlow DTM framework
  - Second generation DTM framework for the JVM (Java, Scala)
  - hyflow.org
- 10 nodes
  - Each is an Intel Xeon 1.9GHz processor with 8 CPU cores
- Benchmarks
  - Skip-list, Linked-list, Hash table, TPC-C

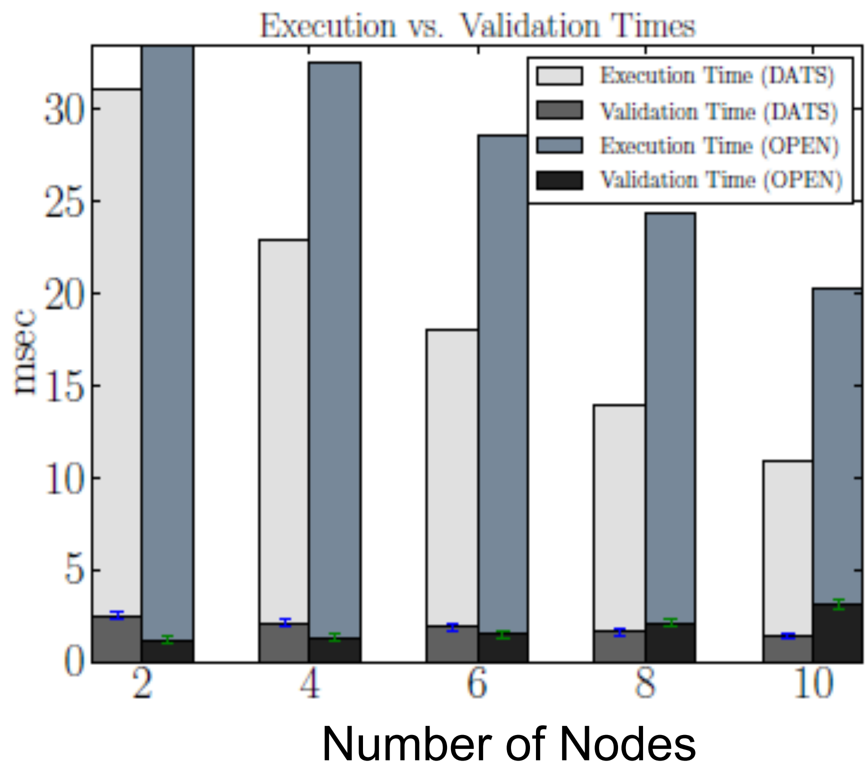
M. Saad and B. Ravindran (2011) . Hyflow: A high performance distributed software transactional memory framework, *HPDC*, pp. 265-266

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

Scheduling overhead and abort reduction

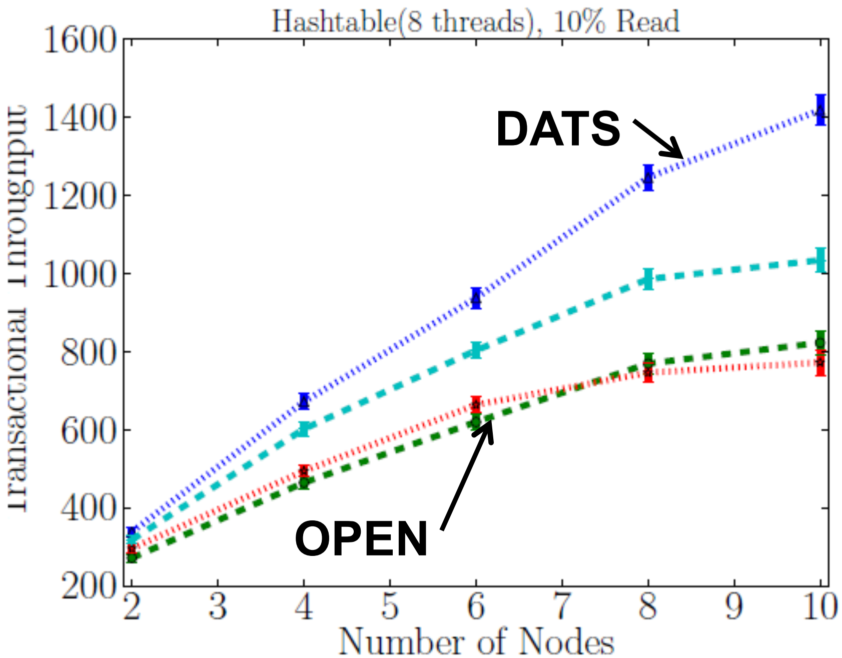


% Abort transactions

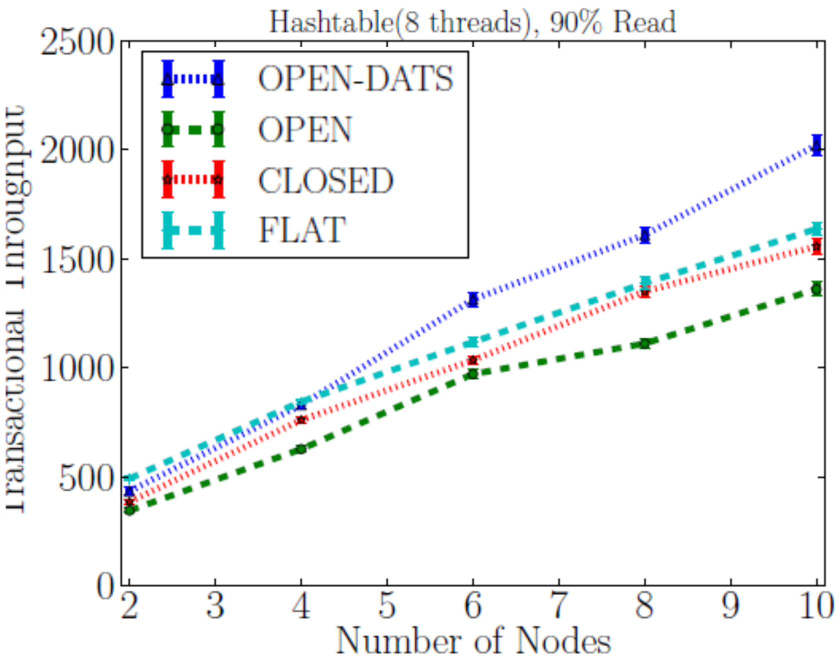


Execution vs. Validation Time

Hash table throughput (8 threads per node)



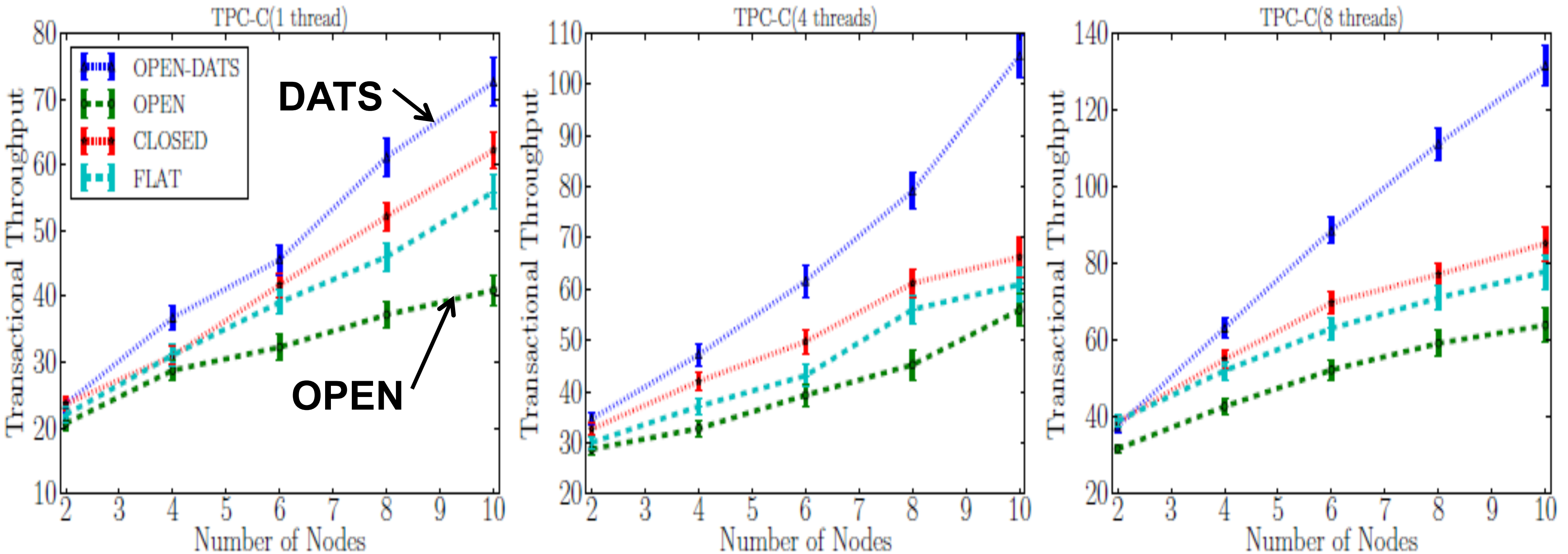
10 % Read



90 % Read

DATS enhances throughput for open-nested transactions over no DATS by as much as 1.7 for micro-benchmarks

TPC-C throughput



1 thread   4 threads   8 threads

DATS enhances throughput for open-nested transactions over no DATS by as much as 2.2 for TPC-C

Conclusions

- DATS avoids unnecessary compensating actions through abstract-level dependency analysis
- DATS enhances transactional throughput for open nested transactions over no DATS
  - By as much as 1.7 and 2.2 with micro-benchmarks and TPC-C

- Compensations needed only if abstract-level transactional dependencies exist
  - Can be detected through dependency analysis
  - Effective for improve concurrency of open-nested transactions

- Future work
  - Automated transactional nesting
  - Open and closed nested transactions in control flow