Lock-free Transactions without Rollbacks for Linked Data Structures

Deli Zhang and Damian Dechev

Department of Computer Science
University of Central Florida

28th ACM Symposium on Parallelism in Algorithms and Architectures

Composite Operations

Example

```
void Move(set a, set b, int key){
  a.Delete(key);
  b.Insert(key);
}
```

Composite Operations

Example

```
void Move(set a, set b, int key){
  a.Delete(key);
  b.Insert(key);
}
```

Example

```
if(!map.containsKey(key)) {
    value = ... // some computation
    map.put(key, value);
}
```

Composite Operations

Example

```
void Move(set a, set b, int key){
  a.Delete(key);
  b.Insert(key);
}
```

Example

```
if(!map.containsKey(key)) {
    value = ... // some computation
    map.put(key, value);
}
```

- Composing linearizable operations is error-prone [Shacham et al., 2011]

Transactional Data Structures

Atomicity

If one operation fails, the entire transaction should abort.

Isolation

Concurrent execution of transactions appears to take effect in some sequential orders that respect real-time ordering.

Software Transactional Memory (STM)

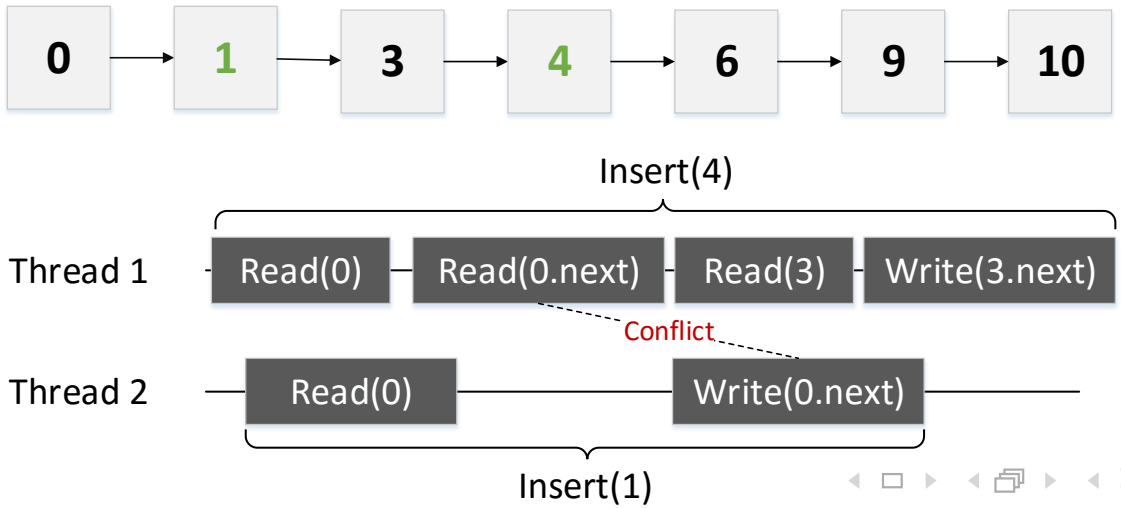
An STM instruments threads' memory accesses, which records the locations a thread read in a *read set*, and the locations it writes in a *write set*. Conflicts are detected among the *read/write sets* of different threads. In the presence of conflicts, only one transaction is allowed to commit while the others are aborted and restarted.

- Memory instrumentation exhibits large overhead

Software Transactional Memory (STM)

An STM instruments threads' memory accesses, which records the locations a thread read in a *read set*, and the locations it writes in a *write set*. Conflicts are detected among the *read/write sets* of different threads. In the presence of conflicts, only one transaction is allowed to commit while the others are aborted and restarted.

- Memory instrumentation exhibits large overhead
- Low-level memory conflicts do not translate to high-level semantic conflicts, which cause excessive aborts



Transactional Boosting [Herlihy and Koskinen, 2008]

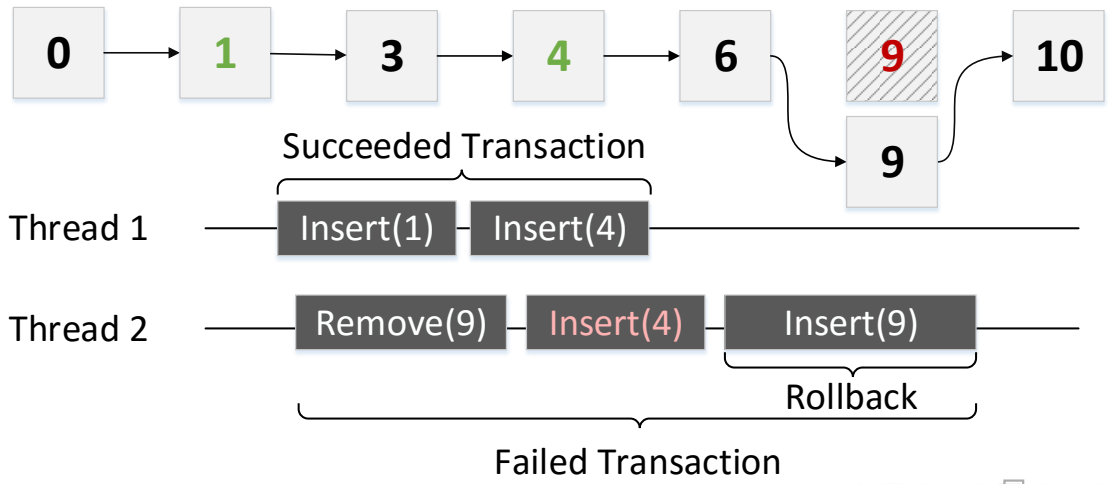
Transactional boosting uses *abstract lock* to ensure that non-commutative method calls never occur concurrently. A transaction makes a sequence of invocation to the objects methods after acquiring the abstract lock associated with each one. It aborts when failed to acquire a lock and recovers from failure by invoking the *inverses* of already executed operations.

- Use of locks degrades the progress guarantee of lock-free data structures

Transactional Boosting [Herlihy and Koskinen, 2008]

Transactional boosting uses *abstract lock* to ensure that non-commutative method calls never occur concurrently. A transaction makes a sequence of invocation to the objects methods after acquiring the abstract lock associated with each one. It aborts when failed to acquire a lock and recovers from failure by invoking the *inverses* of already executed operations.

- Use of locks degrades the progress guarantee of lock-free data structures
- Upon transaction failure the rollback process causes overhead



Automatic Semantic Locking [Golan-Gueta et al, 2015]

Automatically enforces atomicity of given code fragments (multiple calls to different data structures) by inserting pessimistic *abstract locks*.

```
1   atomic {
2     set=map.get(id);
3     if(set==null) {
4       set=new Set(); map.put(id,
5     }
6     set.add(x); set.add(y);
7     if(flag) {
8       queue.enqueue(set);
9       map.remove(id);
10    }
11  }
```

```
1   atomic { map.lock({get(id),put(id,*),remove(id)});
2     set=map.get(id);
3     if(set==null) {
4       set=new Set(); map.put(id, set);
5     }
6     set.lock({add(*)}); set.add(x); set.add(y);
7     if(flag) { queue.lock({enqueue(set)});
8       queue.enqueue(set); queue.unlockAll();
9       map.remove(id);
10    }
11    map.unlockAll(); set.unlockAll();
12  }
```

Automatic Semantic Locking [Golan-Gueta et al, 2015]

Automatically enforces atomicity of given code fragments (multiple calls to different data structures) by inserting pessimistic *abstract locks*.

```
1   atomic {
2     set=map.get(id);
3     if(set==null) {
4       set=new Set(); map.put(id,
5     }
6     set.add(x); set.add(y);
7     if(flag) {
8       queue.enqueue(set);
9       map.remove(id);
10    }
11  }
```

```
1   atomic { map.lock({get(id),put(id,*),remove(id)});
2     set=map.get(id);
3     if(set==null) {
4       set=new Set(); map.put(id, set);
5     }
6     set.lock({add(*)}); set.add(x); set.add(y);
7     if(flag) { queue.lock({enqueue(set)});
8       queue.enqueue(set); queue.unlockAll();
9       map.remove(id);
10    }
11    map.unlockAll(); set.unlockAll();
12  }
```

- Need additional container to store key-lock map
- Lock acquiring is reordered to avoid rollbacks

Lock-free Transactional Transformation (LFTT)

- Challenges
  - Buffering write operations
  - Rollback failed transactions

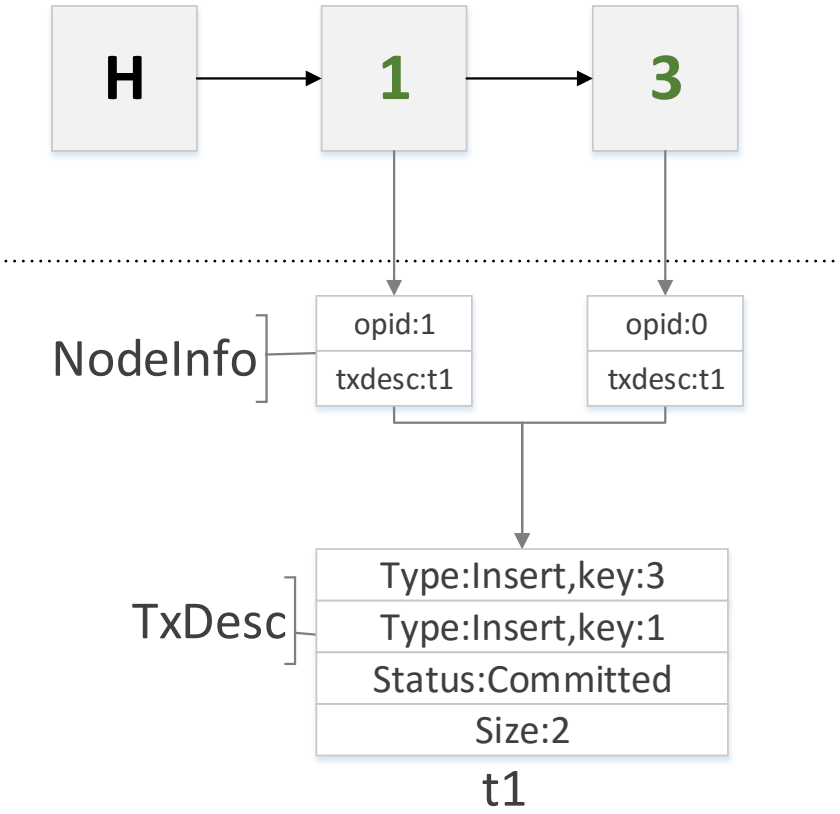
Lock-free Transactional Transformation (LFTT)

- Challenges
  - Buffering write operations
  - Rollback failed transactions
- Key contributions
  - Lock-free semantic conflict detection
  - Logical status interpretation eliminates rollbacks
  - Cooperative transaction execution minimizes aborts

Lock-free Transactional Transformation (LFTT)

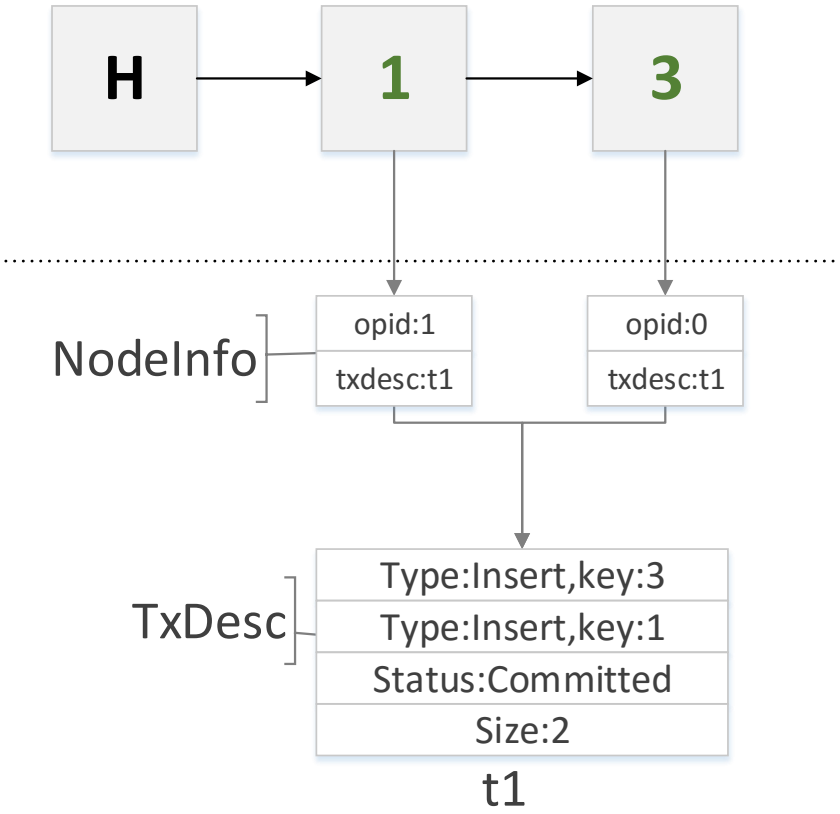
- Challenges
  - Buffering write operations
  - Rollback failed transactions
- Key contributions
  - Lock-free semantic conflict detection
  - Logical status interpretation eliminates rollbacks
  - Cooperative transaction execution minimizes aborts
- Applicable data structures
  - Set: Insert(k), Delete(k), Find(k)
  - Linked data structures: list, skiplist, trees

Node-based Conflict Detection



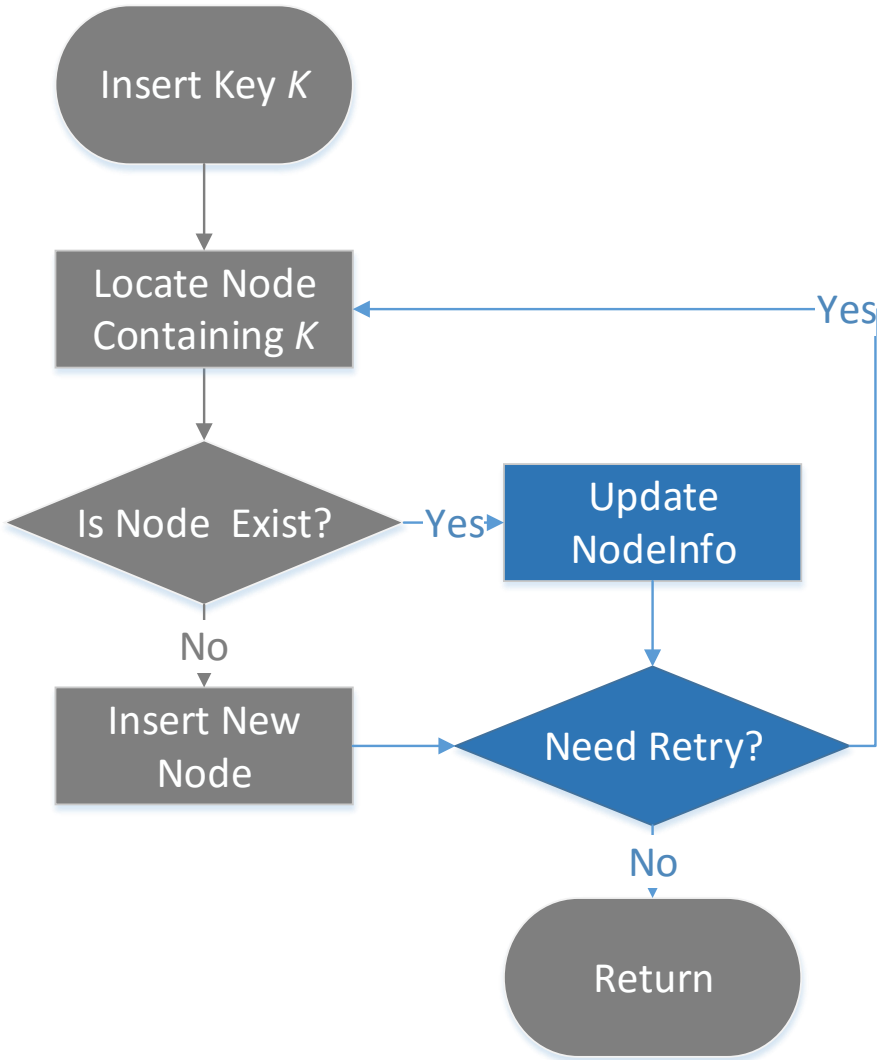
- Embed NodeInfo as a monitors
- TxDesc contains operation context and status

Node-based Conflict Detection



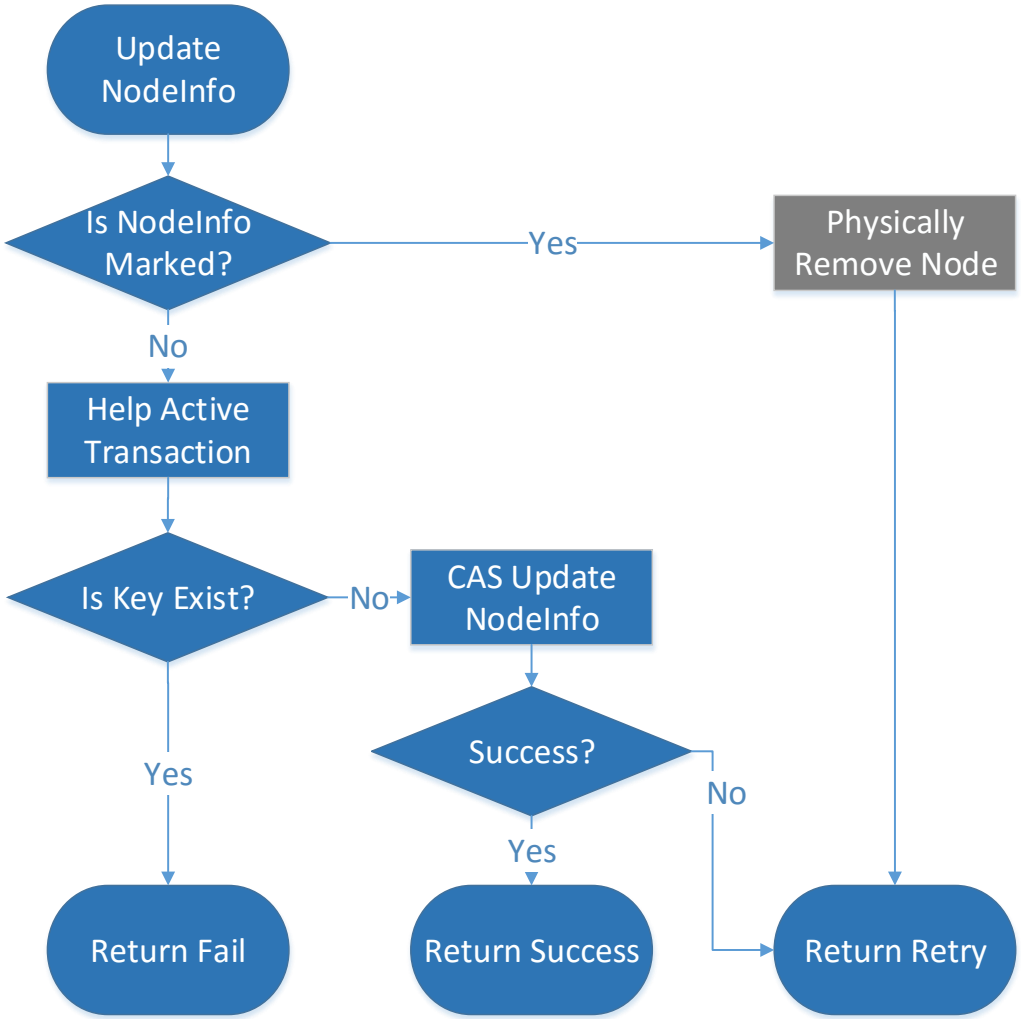
- Embed NodeInfo as a monitors
- TxDesc contains operation context and status
- Eager detection
- Key access = node access

Transformed INSERT Workflow



- Extract DO_LOCATEPRED, and DO_INSERT
- New code path to update NODEINFO

UpdateNodeInfo Workflow for Insert



- Physically remove node with marked NodeInfo
- Enforce serialization through helping
- Interpret logical status using IsKeyExisit
- Returns a tri-state value

Interpretation-based Logical Rollback

Table: IsKeyExist Predicate

Operation \ TxStatus	Committed	Aborted	Active
Insert	True	False	True (same transaction)
Delete	False	True	False (same transaction)
Find	True	True	True

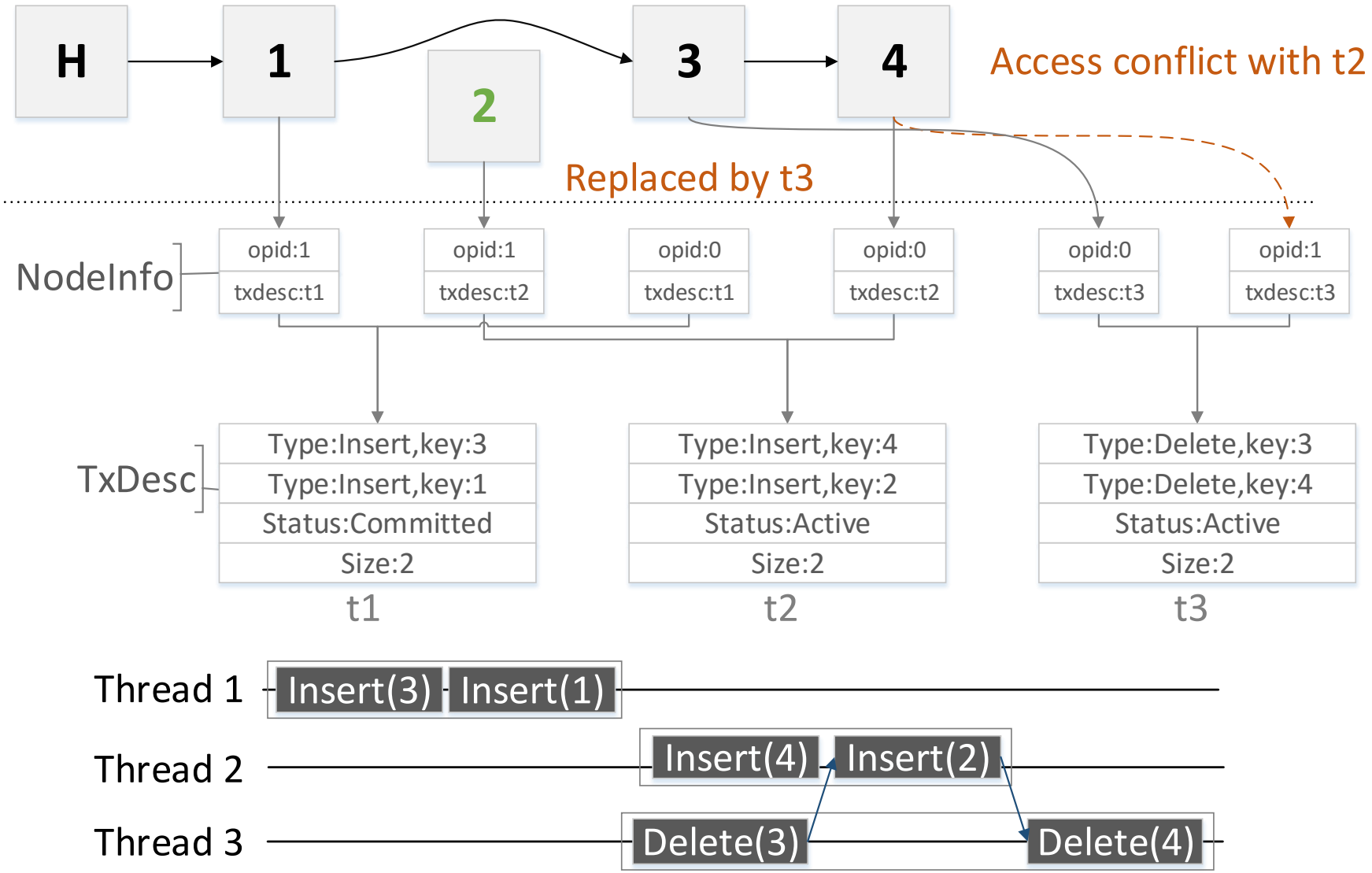
Cooperative Transaction Execution

- Process
  - Invoke the sequence of operation in TxDesc
  - Update the transaction status using CAS
  - Mark NodeInfo on successfully deleted nodes

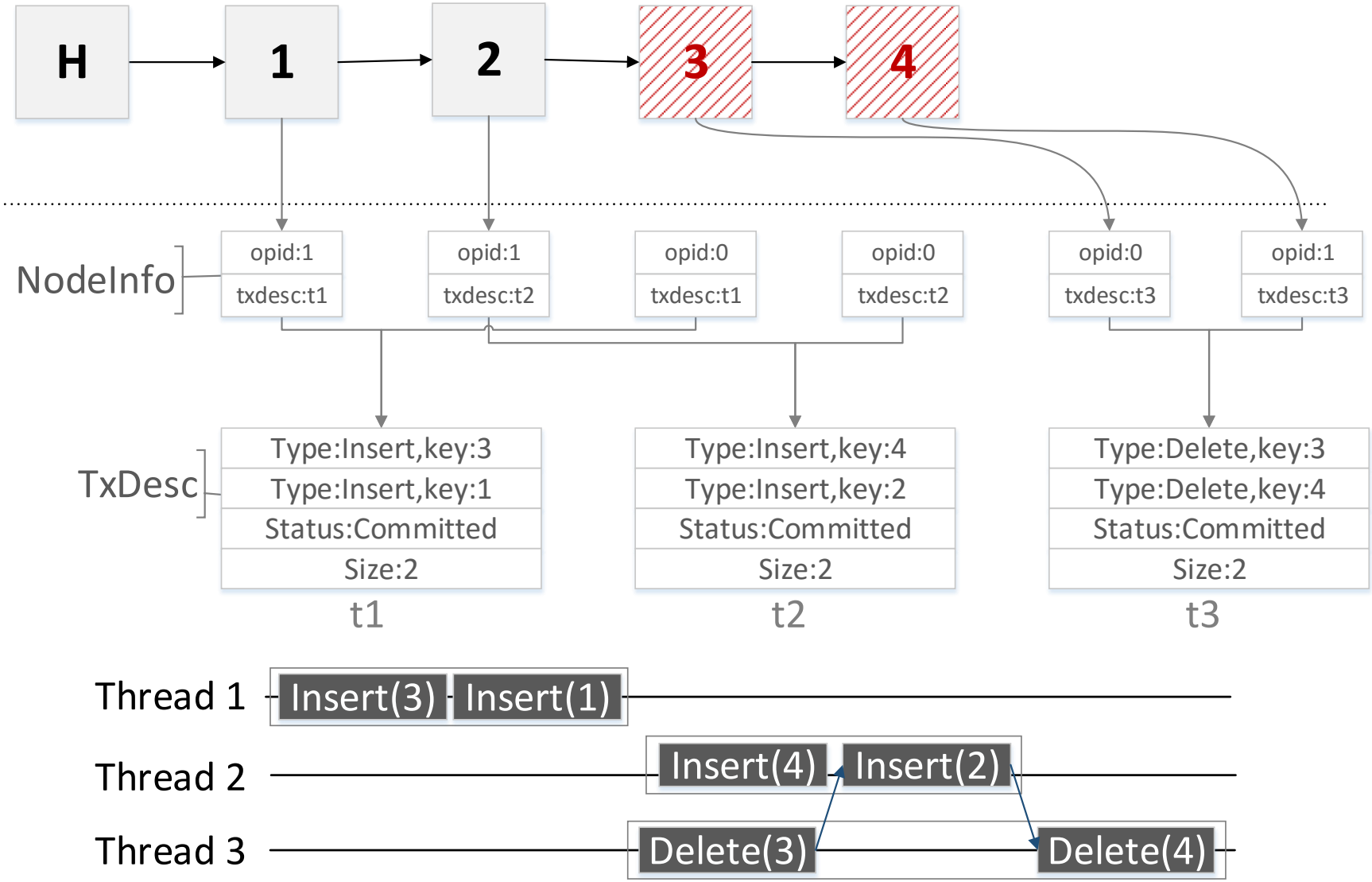
Cooperative Transaction Execution

- Process
  - Invoke the sequence of operation in TxDesc
  - Update the transaction status using CAS
  - Mark NodeInfo on successfully deleted nodes
- Nuances
  - Cyclic dependency check and recovery
    - Duplicate descriptor in HelpStack
  - No help = obstruction-freedom
    - Contention management: aggressive, polite, karma

LFTT in Action



LFTT in Action



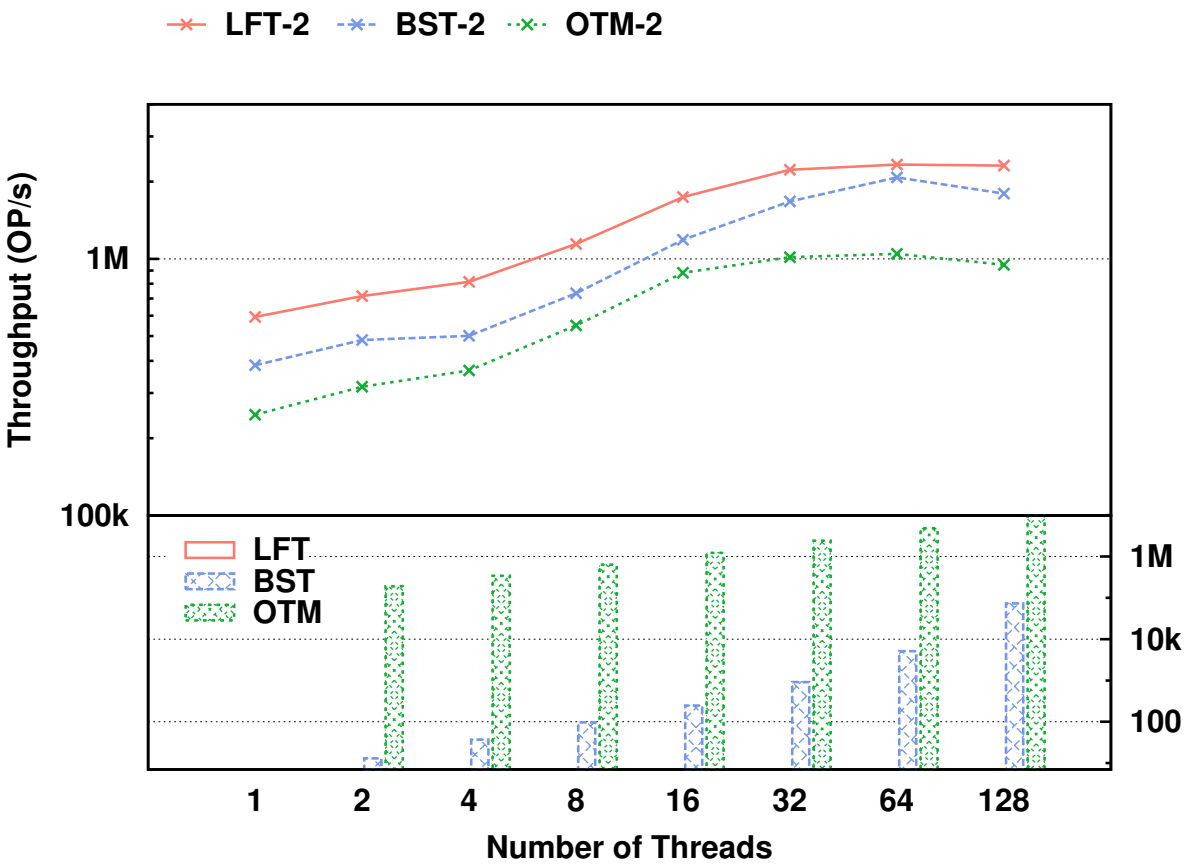
Environment

- Hardware
  - 64-core NUMA (4 AMD Opteron @2.1GHz)
- Micro-benchmark
  - GCC 4.7 w/ O3
  - 1, 2, 4, 8, and 16 operations per transaction
  - Write-dominated, read-dominated, and mixed workloads

Alternatives

- Transactional skiplist [Fraser, 2004]
  - Object-based STM (OTM) [Fraser, 2004]
  - Transactional boosting (BST)
  - Lock-free Transactional Transformation (LFT)
- Transactional linked list [Harris, 2001]
  - Norec word-based STM (NTM) [Dalessandro, 2010]
  - Transactional boosting (BST)
  - Lock-free Transactional Transformation (LFT)

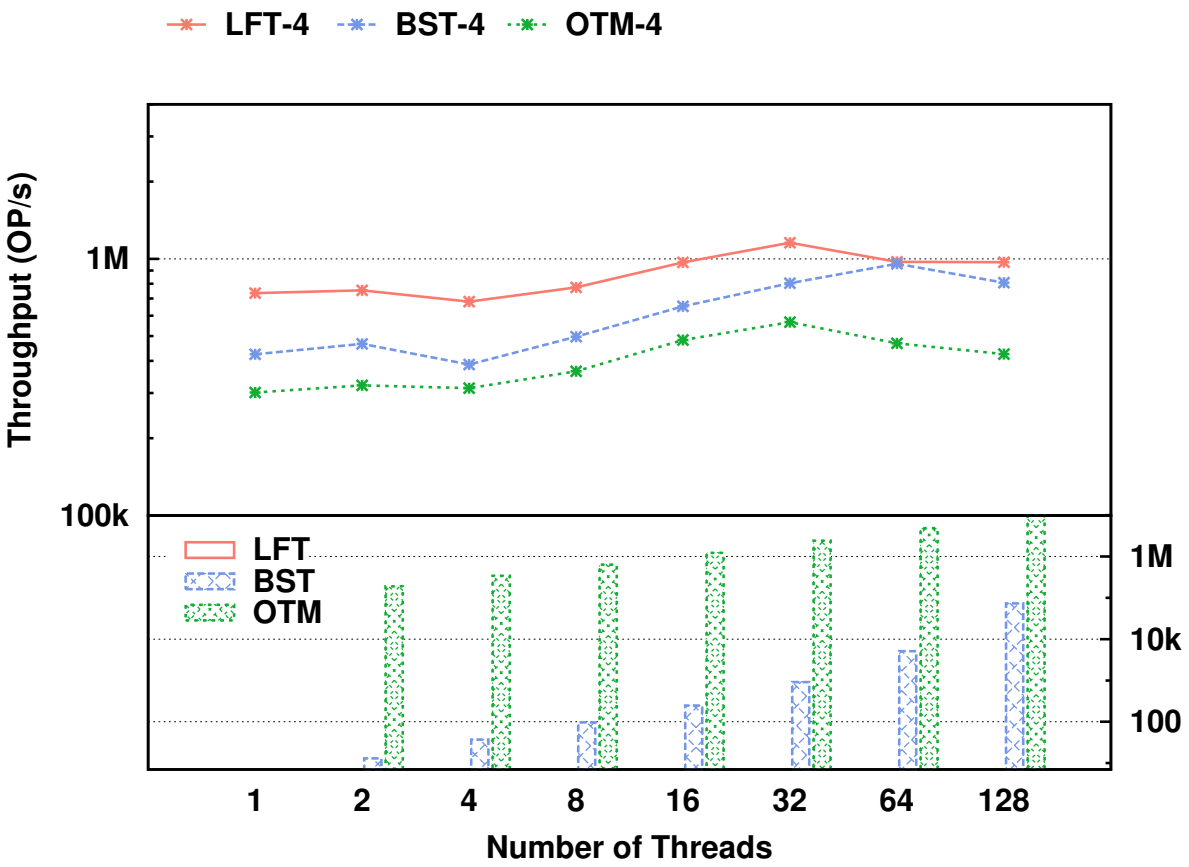
Throughput — Skiplist Mixed Workload



1M Key, 33% Insert, 33% Delete, 34% Find

- Average of 60% speedup over boosting
- 3 times speedup over STM
- No spurious aborts

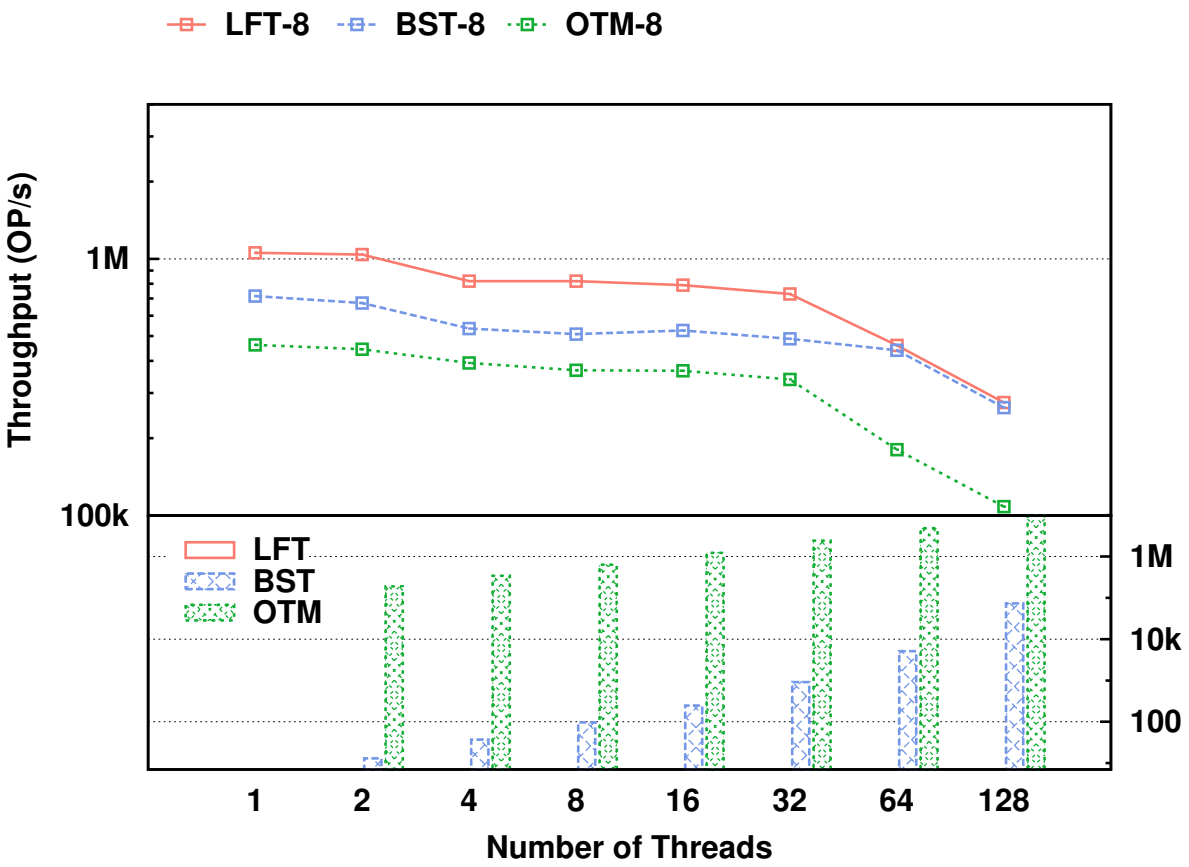
Throughput — Skiplist Mixed Workload



1M Key, 33% Insert, 33% Delete, 34% Find

- Average of 60% speedup over boosting
- 3 times speedup over STM
- No spurious aborts

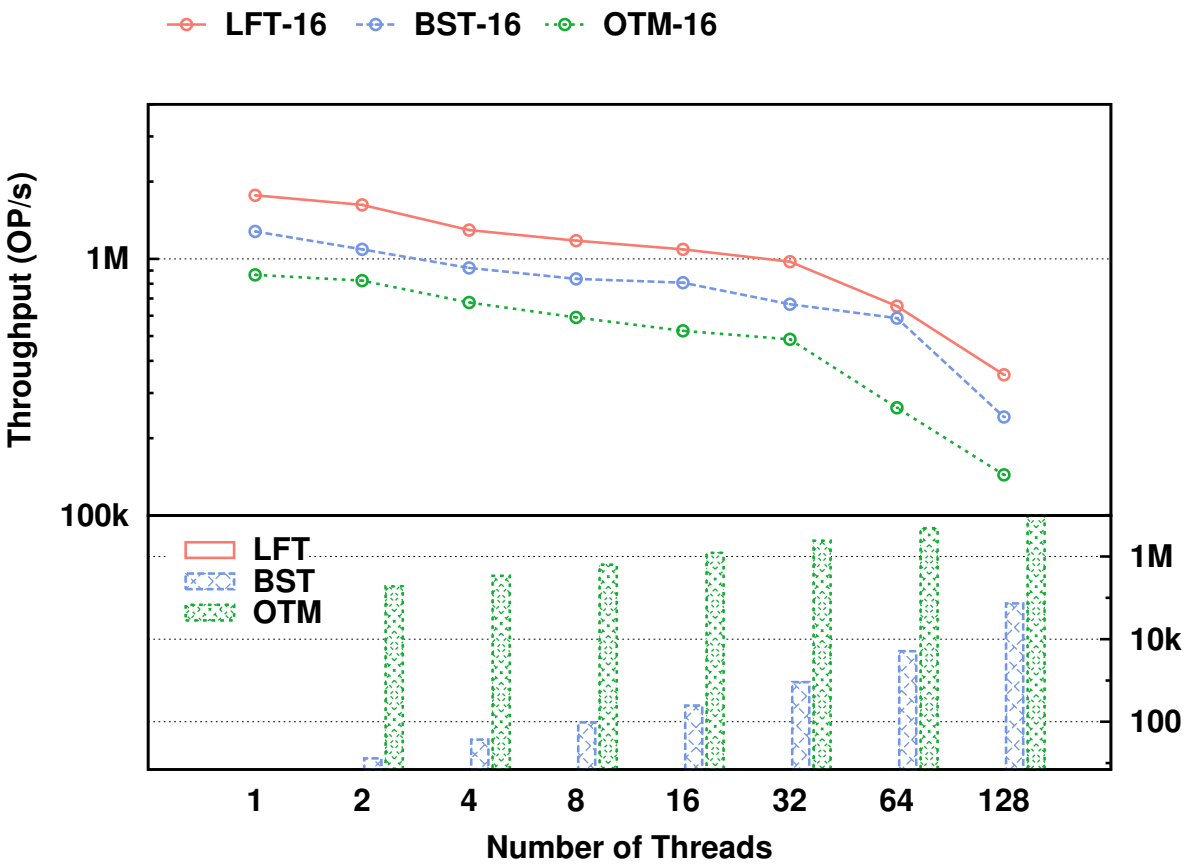
Throughput — Skiplist Mixed Workload



1M Key, 33% Insert, 33% Delete, 34% Find

- Average of 60% speedup over boosting
- 3 times speedup over STM
- No spurious aborts

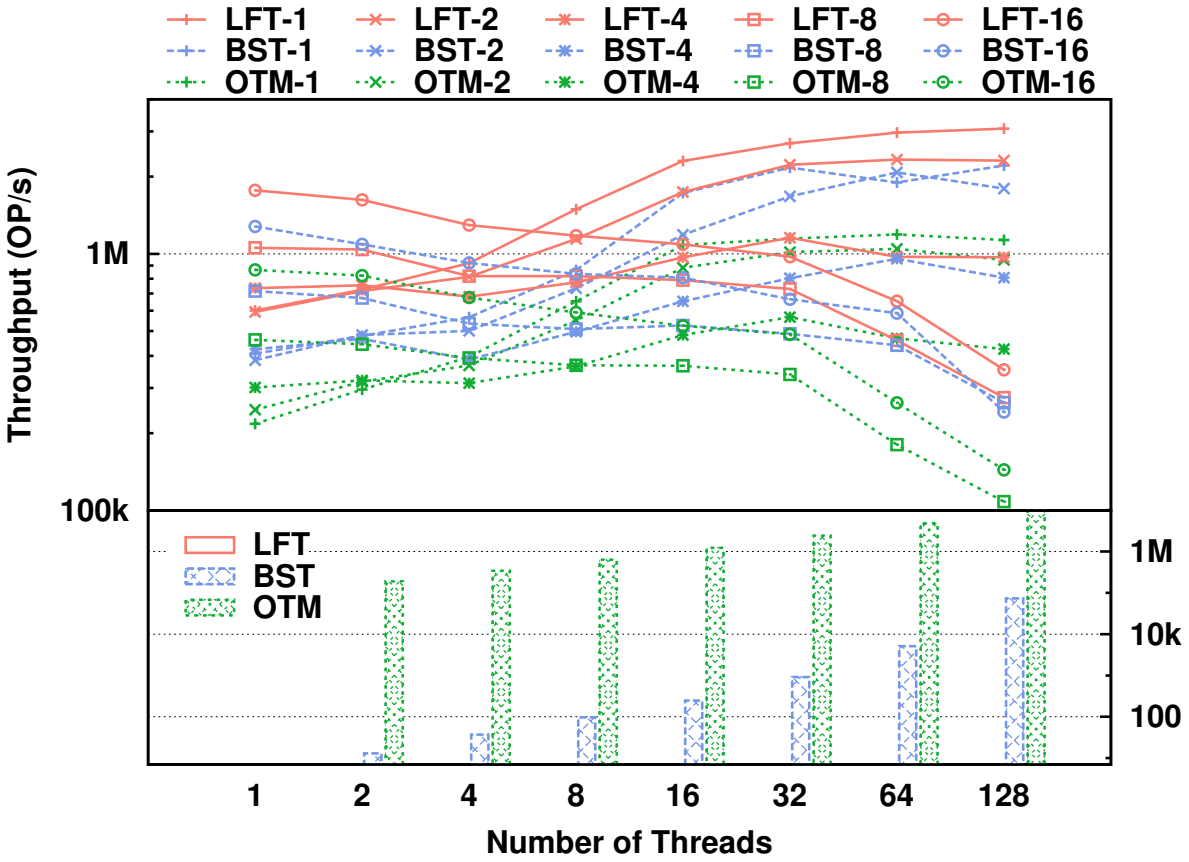
Throughput — Skiplist Mixed Workload



1M Key, 33% Insert, 33% Delete, 34% Find

- Average of 60% speedup over boosting
- 3 times speedup over STM
- No spurious aborts

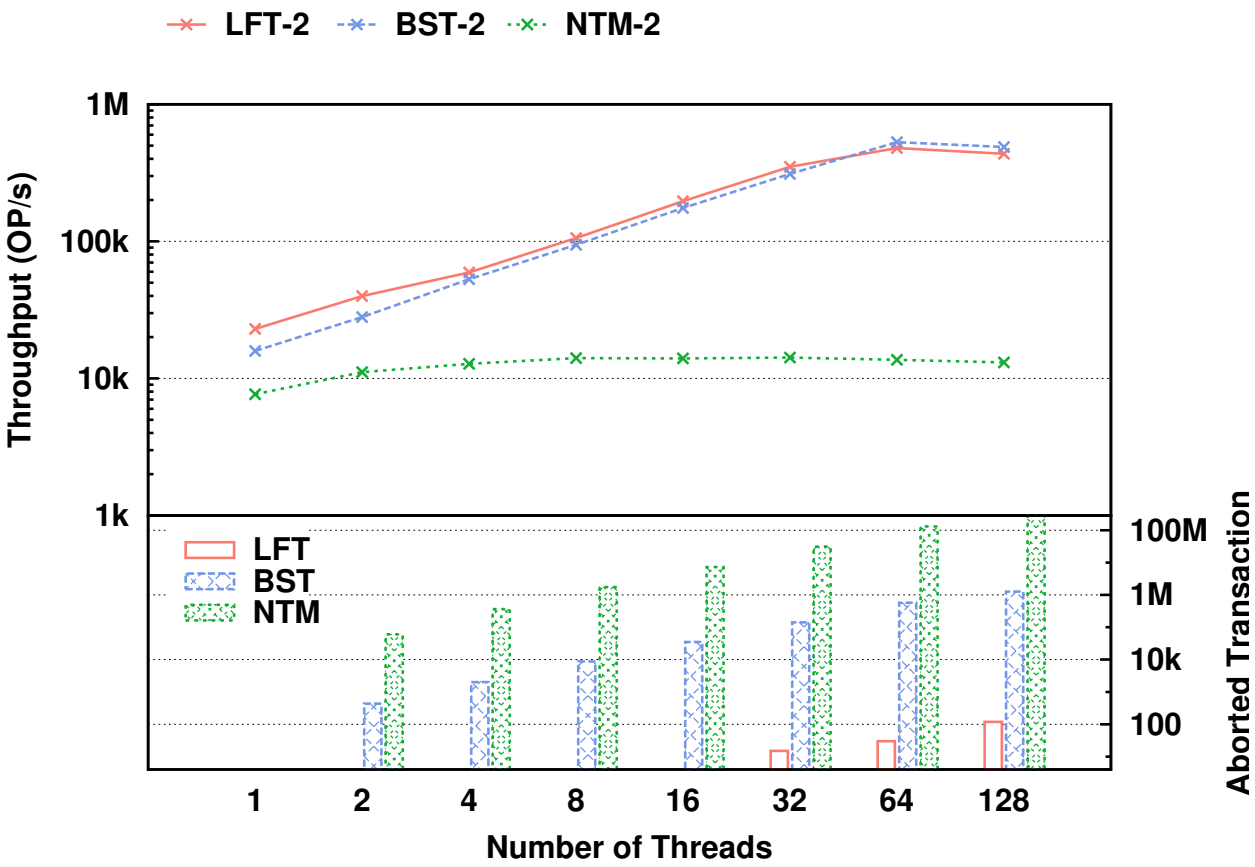
Throughput — Skiplist Mixed Workload



1M Key, 33% Insert, 33% Delete, 34% Find

- Average of 60% speedup over boosting
- 3 times speedup over STM
- No spurious aborts

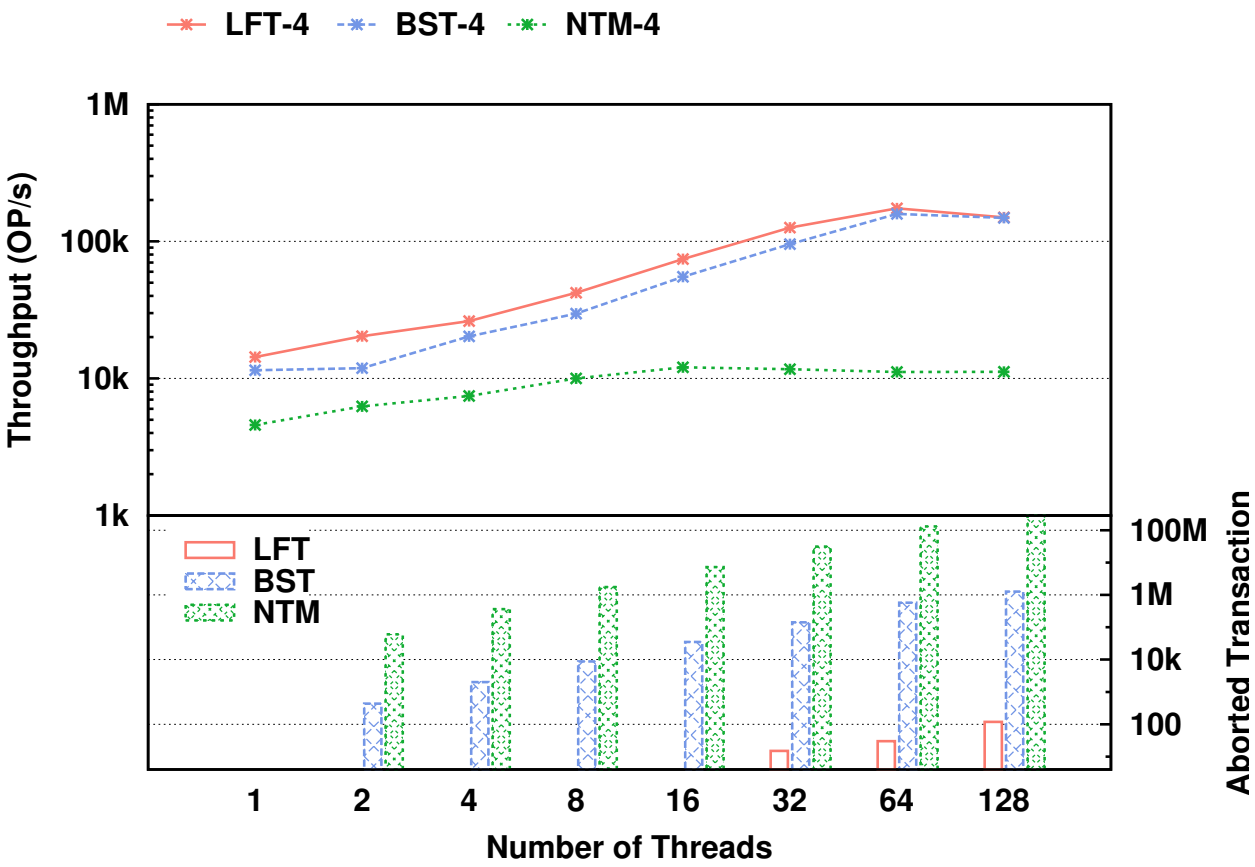
Throughput — Linked List Mixed Workload



10K Key, 33% Insert, 33% Delete, 34% Find

- Average of 40% speedup over boosting
- 10 times speedup over STM
- 2 to 3 orders of magnitude less spurious aborts

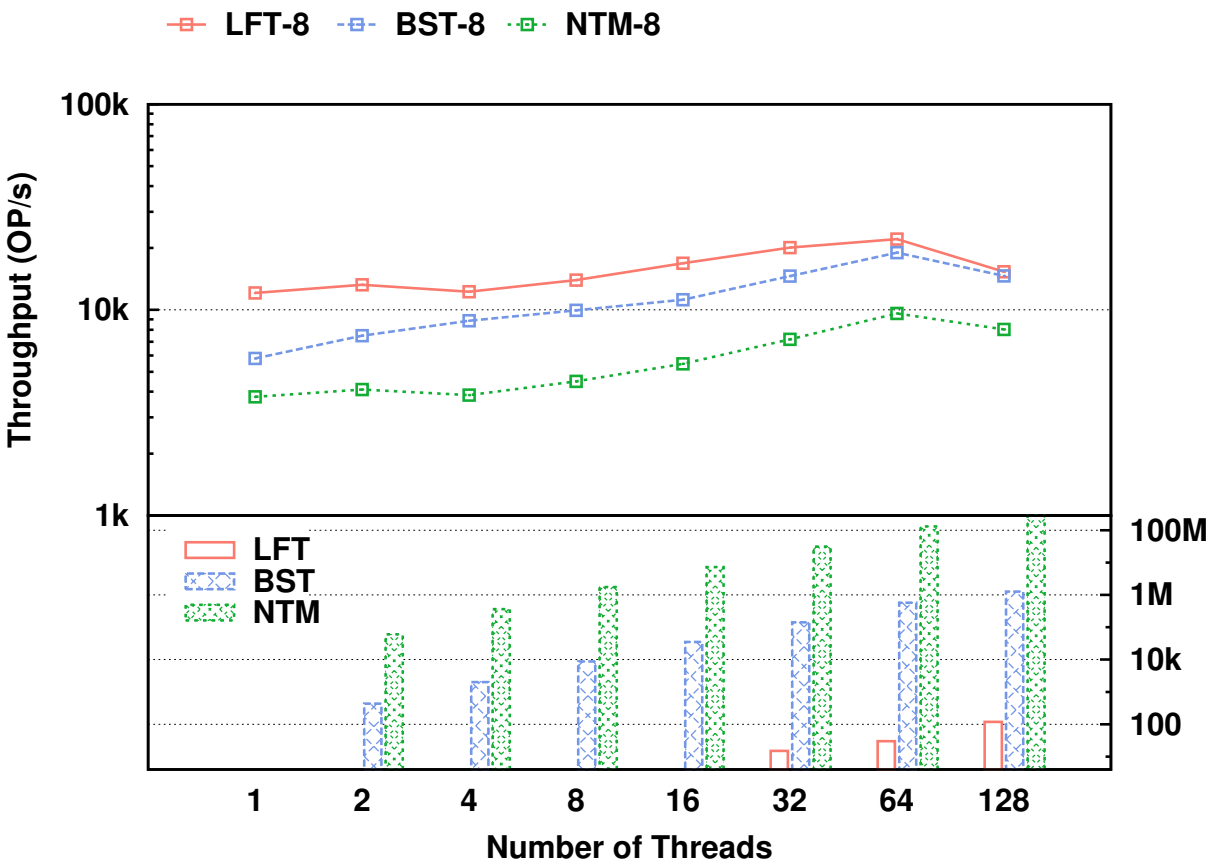
Throughput — Linked List Mixed Workload



10K Key, 33% Insert, 33% Delete, 34% Find

- Average of 40% speedup over boosting
- 10 times speedup over STM
- 2 to 3 orders of magnitude less spurious aborts

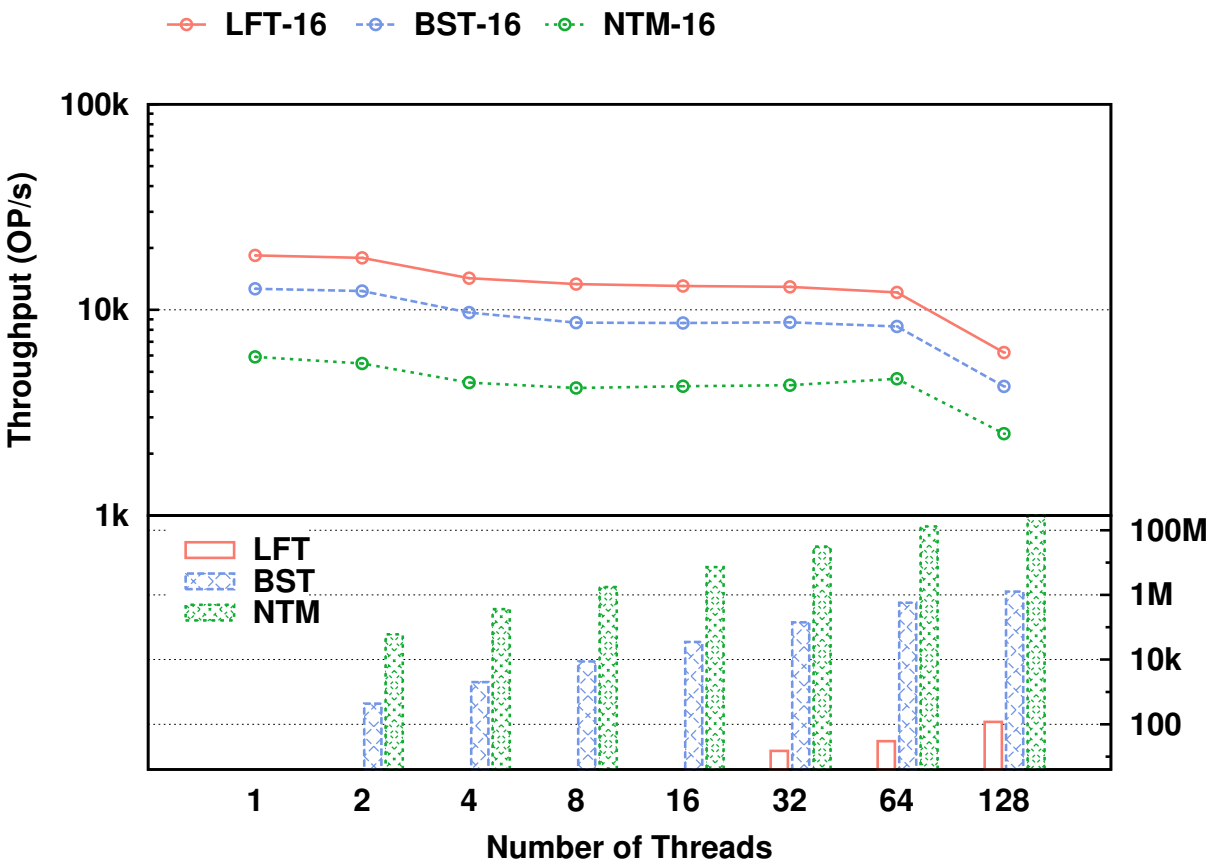
Throughput — Linked List Mixed Workload



10K Key, 33% Insert, 33% Delete, 34% Find

- Average of 40% speedup over boosting
- 10 times speedup over STM
- 2 to 3 orders of magnitude less spurious aborts

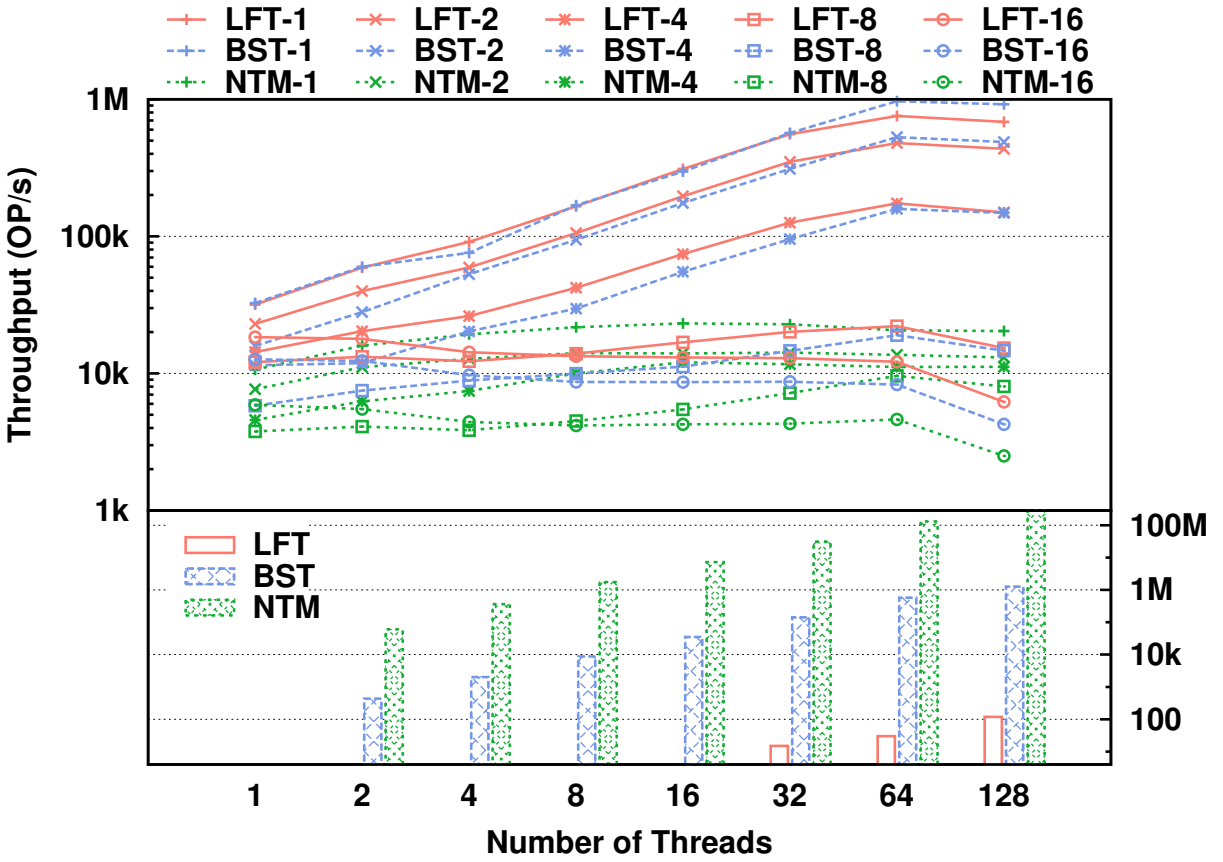
Throughput — Linked List Mixed Workload



10K Key, 33% Insert, 33% Delete, 34% Find

- Average of 40% speedup over boosting
- 10 times speedup over STM
- 2 to 3 orders of magnitude less spurious aborts

Throughput — Linked List Mixed Workload



10K Key, 33% Insert, 33% Delete, 34% Find

- Average of 40% speedup over boosting
- 10 times speedup over STM
- 2 to 3 orders of magnitude less spurious aborts

Summary

- LFTT characteristics
  - Built-in transaction support for lock-free sets
  - Excels at large transactions
  - Greater success rate with minimal spurious aborts
- Future work
  - Support map abstract data type
  - Support wait-free data structures
- Library of transactional data structures
  - Cross-container transactions
  - http://cse.eecs.ucf.edu/gitlab/deli/libtxd