High Performance C++ Concurrent Transactional Data Structures: Concept, Design, and Implementation

Deli Zhang and Damian Dechev

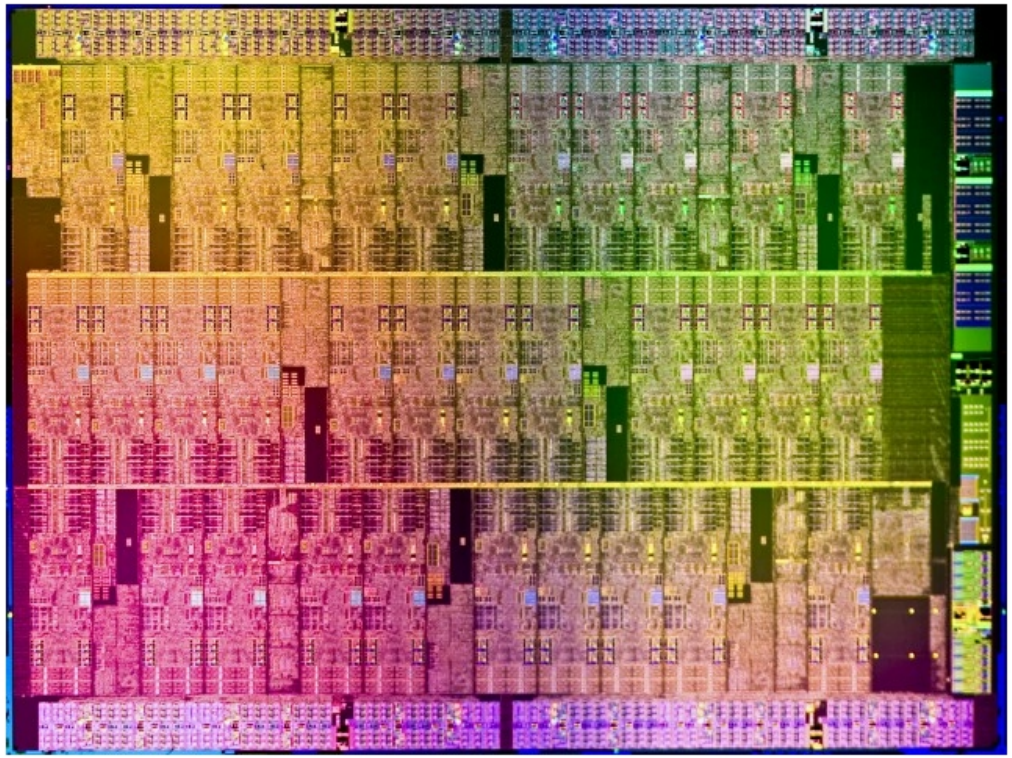
Microsoft University of Central Florida

Developed at University of Central Florida
Partially Supported by NSF Grants: NSF ACI-1440530 and NSF CCF-1218100.

Table of Contents

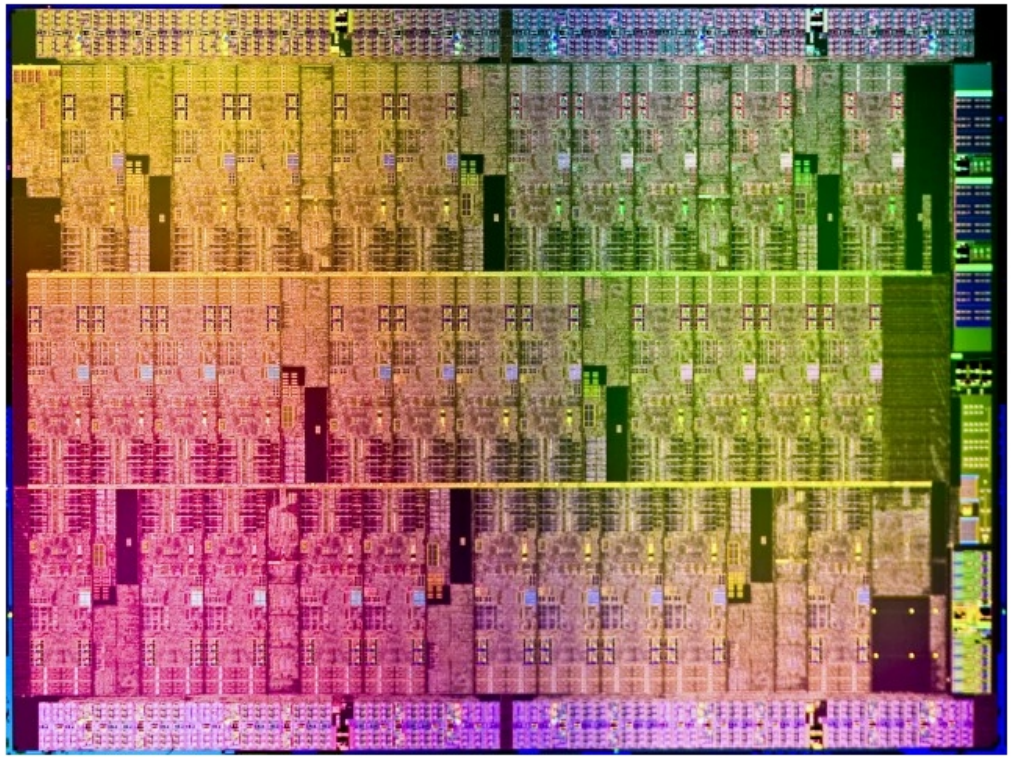
Many-core Processors Are Here

- Intel Xeon Phi 7290
- 72 x86 cores



Many-core Processors Are Here

- Intel Xeon Phi 7290
- 72 x86 cores
- Sunway SW26010
- 260 RISC cores

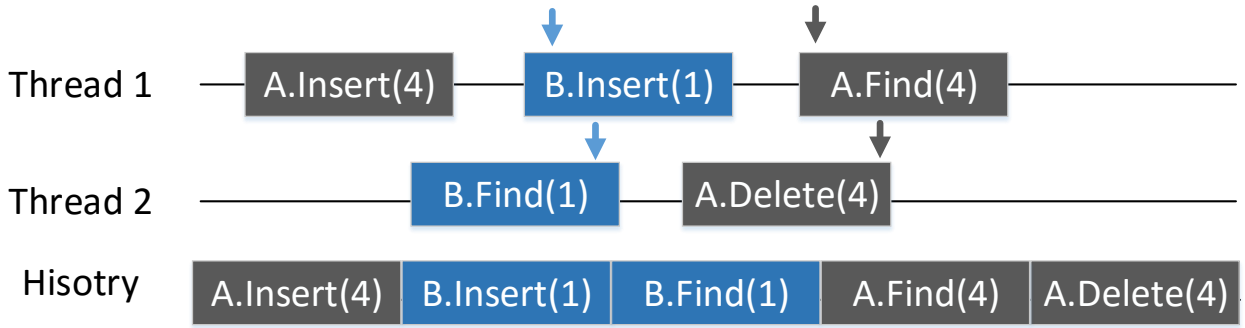


Non-blocking Data Structures

- Progress Guarantees
  - Wait-freedom
  - Lock-freedom
  - Obstruction-freedom
- Linearizability
  - Compositional
  - Container libraries
  - LibCDS, Tervel, TBB

Non-blocking Data Structures

- Progress Guarantees
  - Wait-freedom
  - Lock-freedom
  - Obstruction-freedom
- Linearizability
  - Compositional
  - Container libraries
  - LibCDS, Tervel, TBB



Linearizable Operations

The Catch: Composite Operations

Example

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

The Catch: 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);
}
```

The Catch: Composite Operations

Example

```
void ChangePhone(string name, string phone, string oldPhone){
  contactMap.Update(name, phone);
  phoneMap.Delete(oldPhone);
  phoneMap.Insert(phone, name);
}
```

The Catch: Composite Operations

Example

```
void ChangePhone(string name, string phone, string oldPhone){
  contactMap.Update(name, phone);
  phoneMap.Delete(oldPhone);
  phoneMap.Insert(phone, name);
}
```

- Composing linearizable operations is error-prone [Shacham et al., 2011]
  - Expose internal locks — break encapsulation
  - Manual composition — state explosion

The Catch: Composite Operations

Example

```
void ChangePhone(string name, string phone, string oldPhone){
  contactMap.Update(name, phone);
  phoneMap.Delete(oldPhone);
  phoneMap.Insert(phone, name);
}
```

- Composing linearizable operations is error-prone [Shacham et al., 2011]
  - Expose internal locks — break encapsulation
  - Manual composition — state explosion
- Software development requires composition
  - Modular design
  - Software reuse

Transactional Data Structures

- Desired Properties

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.

Transactional Data Structures

- Desired Properties

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.

- Strict serializability
  - Analogue of linearizability for transactions

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.

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.

Example

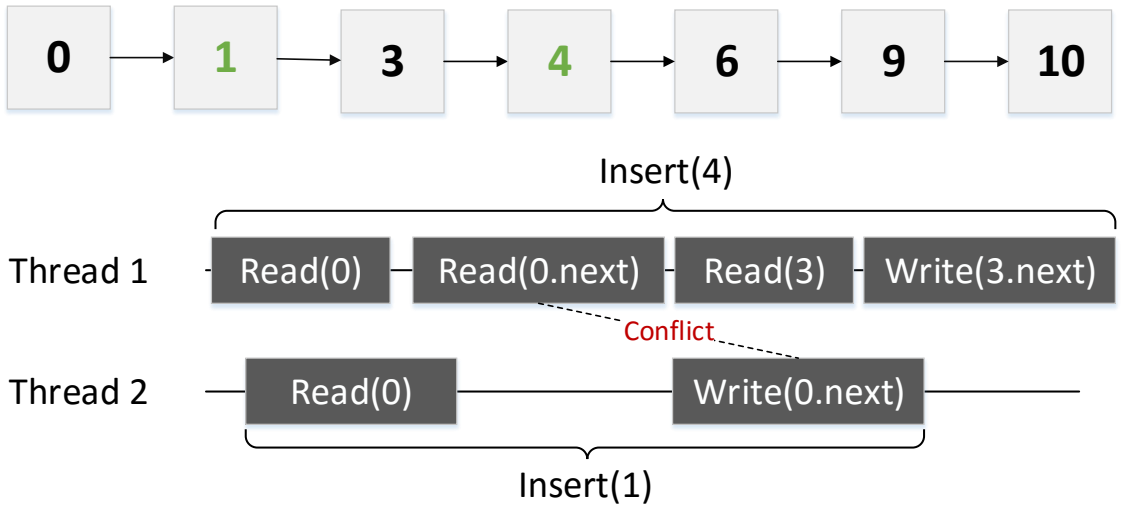
```
TM_BEGIN() {
    const Node *prev(m_head), *curr(TM_READ(prev->m_next));
    while (curr != NULL) {
        if (TM_READ(curr->m_val) >= val) break;
        prev = curr;
        curr = TM_READ(prev->m_next);
    }
    if (!curr || (TM_READ(curr->m_val) > val)) {
        Node* insert_point = const_cast<Node*>(prev);
        Node* i = (Node*)TM_ALLOC(sizeof(Node));
        i->m_val = val; i->m_next = const_cast<Node*>(curr);
        TM_WRITE(insert_point->m_next, i);
    }
}
TM_END;
```

Software Transactional Memory (STM)

- Memory instrumentation exhibits large overhead

Software Transactional Memory (STM)

- 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.

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.

Set Specification

Method	Inverse
add$(x)/\mathit{false}$	noop()
add$(x)/\mathit{true}$	remove$(x)/\mathit{true}$
remove$(x)/\mathit{false}$	noop()
remove$(x)/\mathit{true}$	add$(x)/\mathit{true}$
contains$(x)/_$	noop()

Commutativity

insert $(x)/\_ \Leftrightarrow$ insert $(y)/\_, x \neq y$

remove$(x)/_ \Leftrightarrow$ remove$(y)/_, x \neq y$

insert $(x)/\_ \Leftrightarrow$ remove$(y)/_, x \neq y$

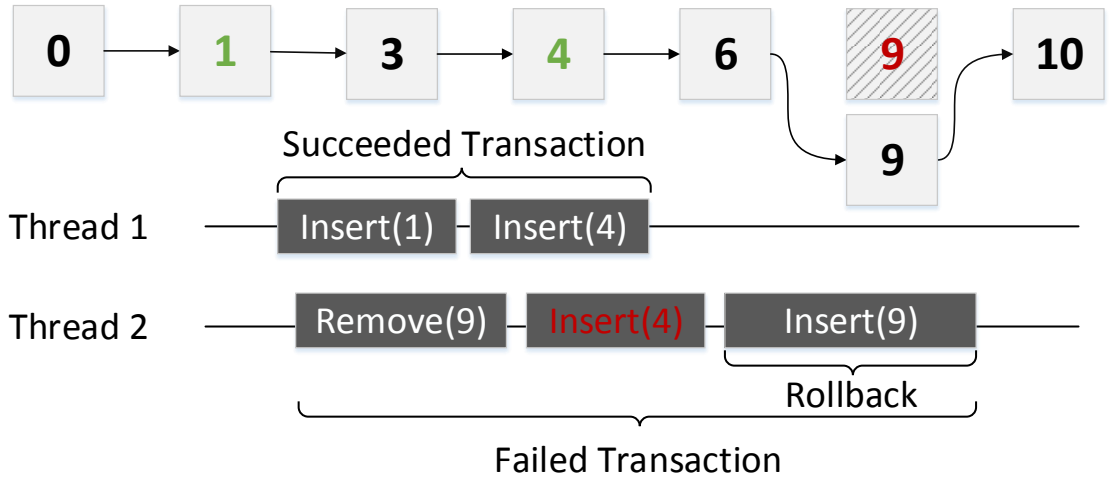
add$(x)/\mathit{false} \Leftrightarrow$ remove$(x)/\mathit{false} \Leftrightarrow$ contains$(x)/_$

Transactional Boosting [Herlihy and Koskinen, 2008]

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

Transactional Boosting [Herlihy and Koskinen, 2008]

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



Methodology Overview

- Development Goal
  - Efficient transactions for legacy lock-based data structures
  - High-performance transactions for existing lock-free data structures
  - New search data structures optimized for concurrency and transaction
  - Composable transactions among arbitrary data structures

Methodology Overview

- Development Goal
  - Efficient transactions for legacy lock-based data structures
  - High-performance transactions for existing lock-free data structures
  - New search data structures optimized for concurrency and transaction
  - Composable transactions among arbitrary data structures
- Methodologies
  - Multi-resource Lock (MRLock) [2, 5]
  - Lock-free Transactional Transformation (LFTT) [4]
  - Multi-dimension List (MDList) [3, 1]
  - Framework for cross-container transactions (Libtxd)
  - https://ucf-cs.github.io/tlds

Implementation Techniques

- CompareAndSwap (CAS) synchronization primitive
- CAS-based retry loop
- Descriptor object
- Cooperative execution (helping)

Implementation Techniques

- CompareAndSwap (CAS) synchronization primitive
- CAS-based retry loop
- Descriptor object
- Cooperative execution (helping)

Example

```
Desc* new_desc = new Desc();
Node* n = LocateNode();
do{
    Desc* curr_desc = n->desc;
    if(!IsTaskDone(curr_desc)) {
        HelpTask(curr_desc);
        continue;
    }
}while(!CAS(&n->desc, curr_desc, new_desc))
```

Transaction Execution as Resource Allocation

Definition

Given a pool of k resources that require exclusive access, each thread may request $1 \leq h \leq k$ resources, and a thread remains blocked until all required resources are available.

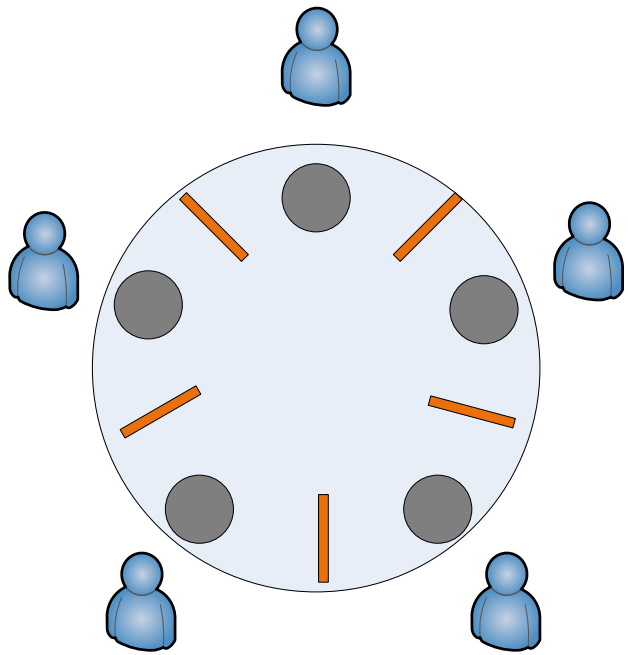
- Data items as resources

Transaction Execution as Resource Allocation

Definition

Given a pool of k resources that require exclusive access, each thread may request $1 \leq h \leq k$ resources, and a thread remains blocked until all required resources are available.

- Data items as resources
- Generalizes the Dining Philosophers Problem



Acquiring Multiple Locks

- Locking protocols — one lock at a time
  - Two-phase locking
  - Resource hierarchy
  - Time-stamp locking

Acquiring Multiple Locks

- Locking protocols — one lock at a time
  - Two-phase locking
  - Resource hierarchy
  - Time-stamp locking
  - Prone to conflict and retry

Acquiring Multiple Locks

- Locking protocols — one lock at a time
  - Two-phase locking
  - Resource hierarchy
  - Time-stamp locking
  - Prone to conflict and retry
- Batch locking — multiple locks in a batch
  - Extend TATAS
  - Multi-resource lock (MRLock)

Extended TATAS

```
typedef uint64 bitset;

void lock(biteset* l,biteset r){
  biteset b;
  do{
    b = *l;
    if(b & r) //detect conflict
      continue;
  }while(CAS(l, b, b | r) != b);
}
```

- A bitset is an array of bits
- Represent a resource by one bit
- Detecting conflict by bitwise AND
- Handle acquisition in batch

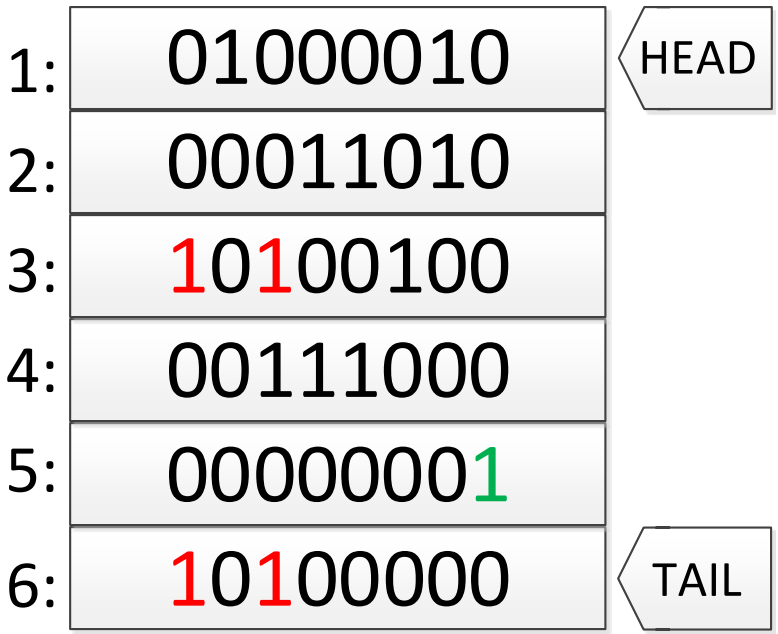
Extended TATAS

```
typedef uint64 bitset;

void lock(biteset* l,biteset r){
  biteset b;
  do{
    b = *l;
    if(b & r) //detect conflict
      continue;
  }while(CAS(l, b, b | r) != b);
}
```

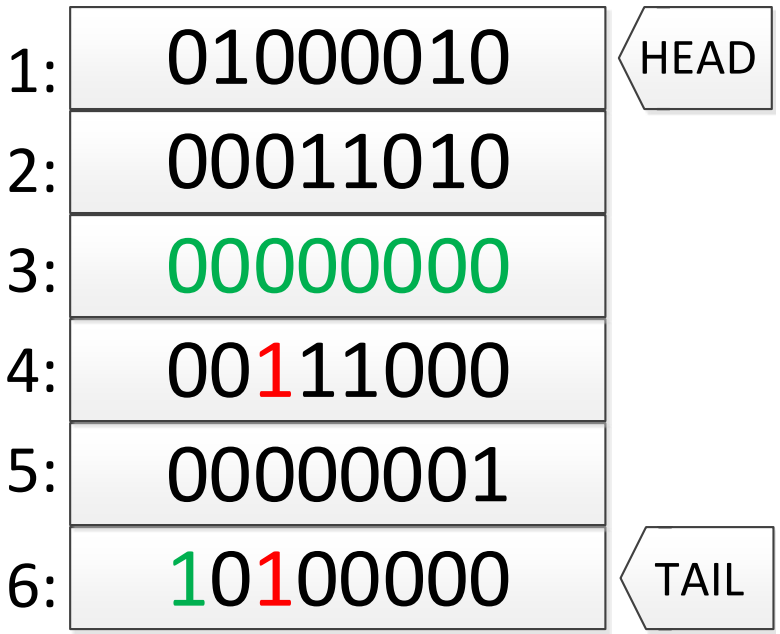
- A bitset is an array of bits
- Represent a resource by one bit
- Detecting conflict by bitwise AND
- Handle acquisition in batch
- Drawbacks
  - No fairness guarantee
  - Heavy contention
  - Limited number of resources

Queue-based Multi-resource Lock



- First to guarantee FIFO fairness
- Unbounded number of resources
- Excels at high levels of contention

Queue-based Multi-resource Lock



- First to guarantee FIFO fairness
- Unbounded number of resources
- Excels at high levels of contention

Data Structure

```
struct cell{
  atomic<uint32> seq;
  bitset bits;
}
struct mrlock{
  cell* buffer;
  uint32 siz;
  atomic<uint32> head;
  atomic<uint32> tail;
}
void init(mrlock& l, uint32 siz){
  l.buffer = new cell[siz];
  l.siz = siz;
  l.head.store(0);
  l.tail.store(0);
  for(uint32 i = 0; i < siz; i++){
      l.buffer[i].bits.set();
      l.buffer[i].seq.store(i);
  }
}
```

- Declaration
  - *Sequence number* as sentinel
  - `bits` as resource flags
  - Atomic queue `head` and `tail`
- Initialization
  - Allocate adjacent buffer cells
  - `bits` are initialized to 1s
  - `seq` are initialized to cell index

Lock Acquire

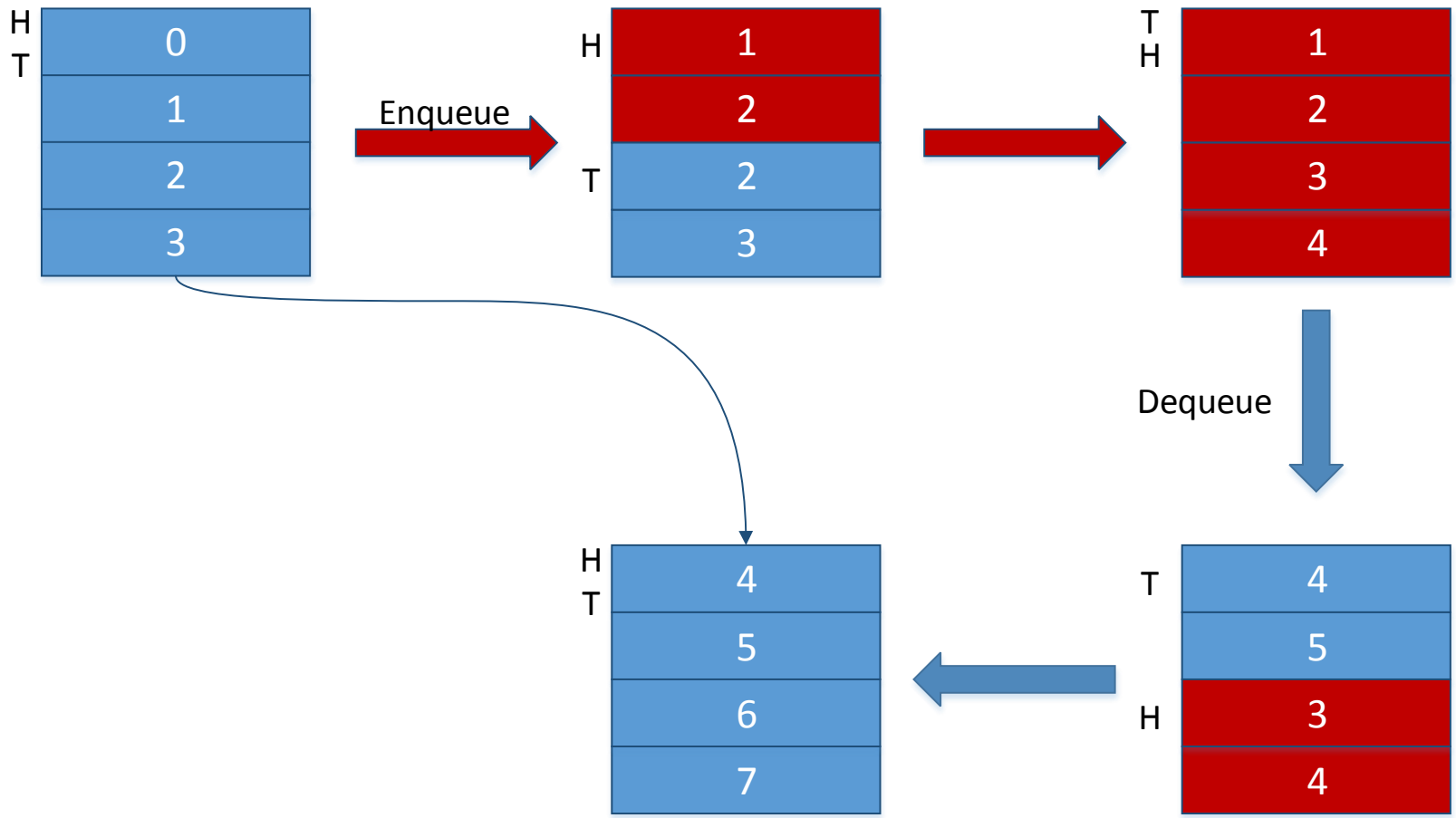
```
function ACQUIRE(mrlock* l, bitset r)
    loop
        pos ← l.tail, c ← ReadCell(l, pos), seq ← c.seq
        if seq - pos == 0 then
            if CAS(&l.tail, pos, pos + 1) succeeds then
                break
    c.bits ← r, c.seq ← pos + 1

    spin_pos ← l.head
    while spin_pos != pos do
        if IsDequeued(spin_pos) or NoConflict(spin_pos, r) then
            spin_pos++
    return pos
```

Lock Release

```
function RELEASE(mrlock* l, handle pos)
    ReadCell(l, pos).bits ← 0
    pos ← l.head
    while ReadCell(l, pos).bits == 0 do
        c ← ReadCell(l, pos)
        seq ← c.seq
        if seq - pos - 1 == 0 then
            if CAS(&l.head, pos, pos + 1) succeeds then
                c.bits ← ∼0
                c.seq ← pos + l.siz
        pos ← l.head
```

Sequence Number



Updating flow of sequence numbers

Correctness

- Concurrent update of the ring buffer is safe

Theorem

The head always precedes the tail; the tail is larger than head by at most N, where N equals to the size of the buffer.

Correctness

- Concurrent update of the ring buffer is safe

Theorem

The head always precedes the tail; the tail is larger than head by at most N, where N equals to the size of the buffer.

- Non-atomic update of bitset is safe
  - Bitsets are initialized to 1s
  - Then written to specific request value by one thread

Theorem

In the presence of a single writer, intermediate values of the bitset during the write operation represent some **supersets** *of requested resources.*

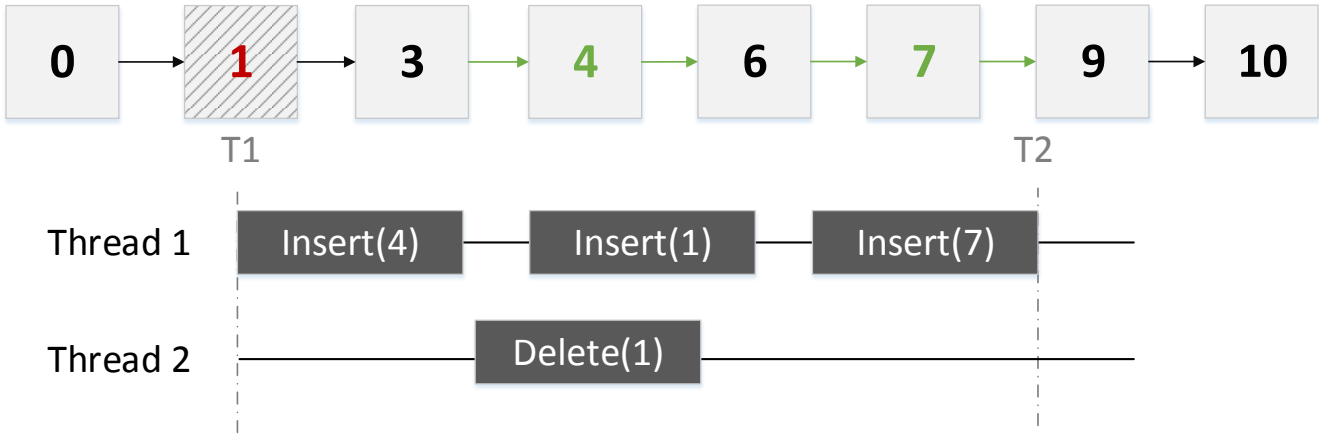
Transaction Using MRLock

Example

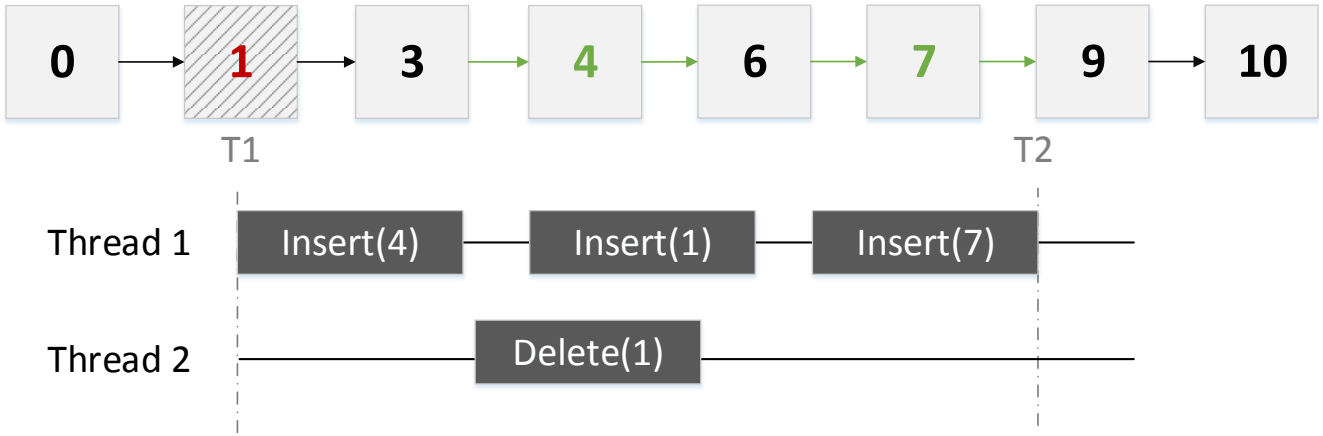
```
FineGrainedLockingList<int> a;
MRLock mrlock;
ResourceAllocator res(mrlock);
vector<int> resources = {2,7,5,8};
lockable = res.CreateLockable(resrouces);

lockable.Lock();
a.Insert(2);
a.Delete(7);
a.Find(8);
a.Insert(5);
lockable.Unlock();
```

Challenges for Lock-free Transactions



Challenges for Lock-free Transactions



- Optimistic synchronization
- Buffering write operations
- Rollback failed transactions

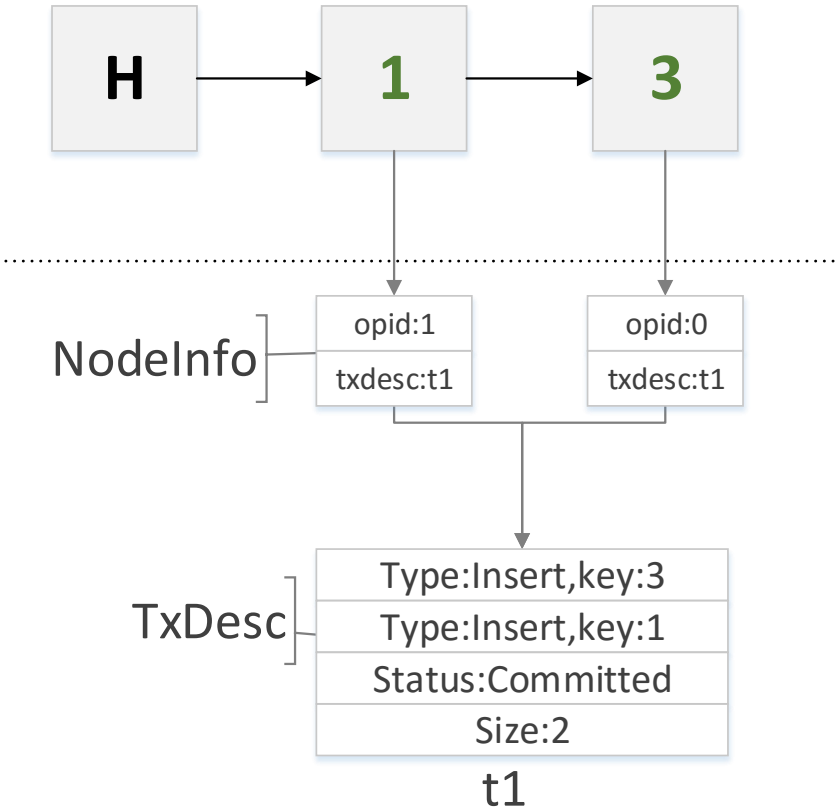
LFTT Overview

- Lock-free Transactional Transformation (LFTT)
  - Lock-free semantic conflict detection
  - Logical status interpretation eliminates rollbacks
  - Cooperative transaction execution minimizes aborts
  - Significant performance gains

LFTT Overview

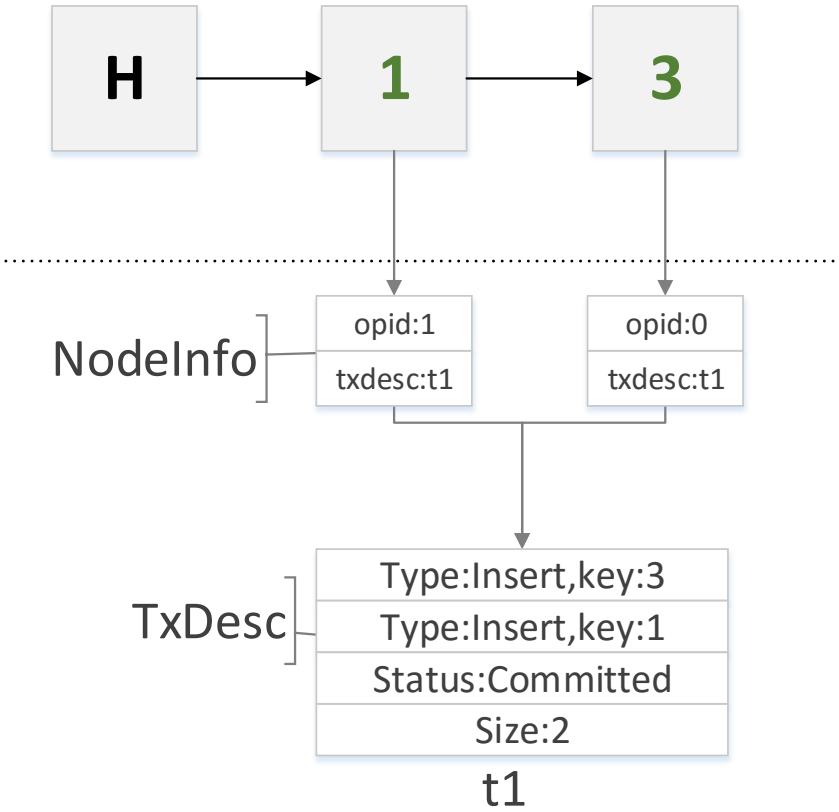
- Lock-free Transactional Transformation (LFTT)
  - Lock-free semantic conflict detection
  - Logical status interpretation eliminates rollbacks
  - Cooperative transaction execution minimizes aborts
  - Significant performance gains
- Applicable Data Structures
  - Abstract Data Types
    - Set: Insert(k), Delete(k), Find(k)
    - Map: Insert(k,v), Delete(k), Find(k), Update(k,v)
  - Node-based Linked Data Structures
    - Linked list, binary search trees, skiplist, MDList
    - Common fields: *key*, *val*, *next*[]

Node-based Conflict Detection



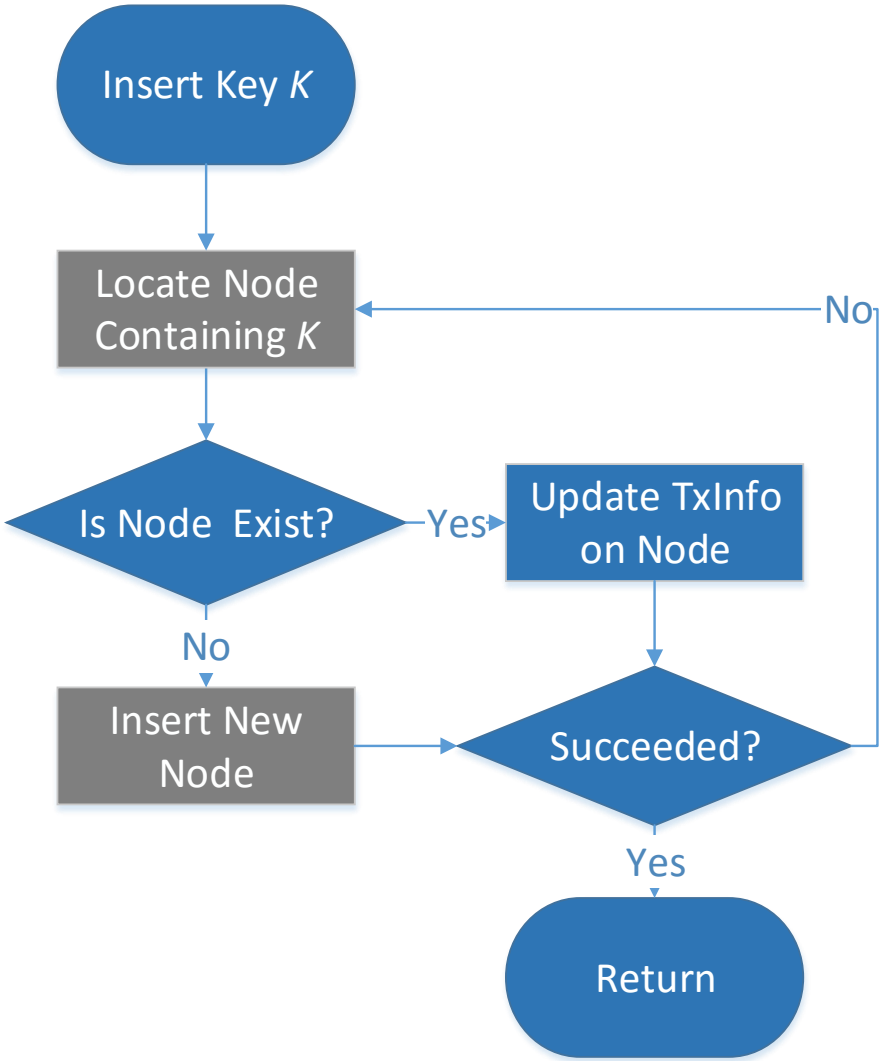
- Embed NodeInfo as a monitors
- TxDesc contains operation context and status

Node-based Conflict Detection



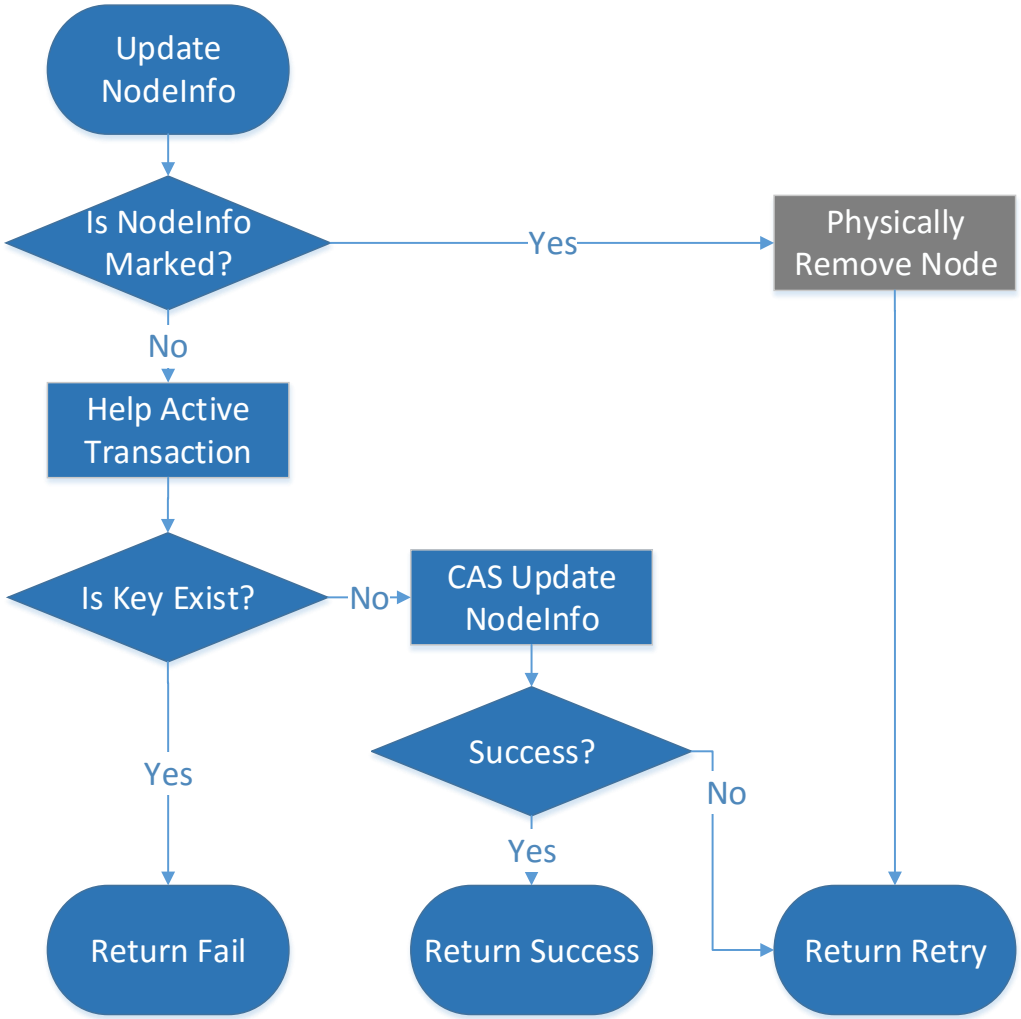
- Embed NodeInfo as a monitors
- TxDesc contains operation context and status
- Eager detection
- Key access = node access

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

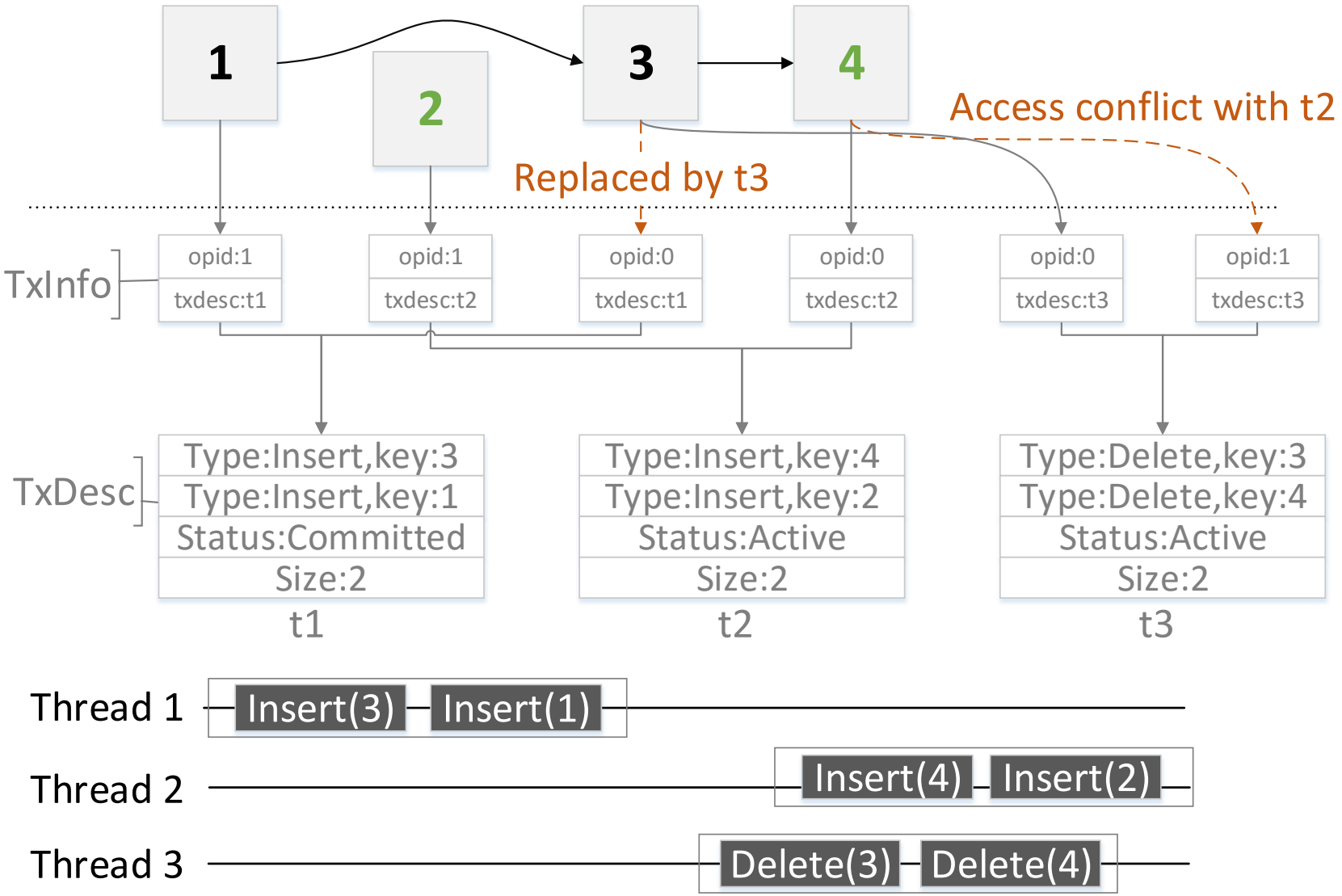
Cooperative Transaction Execution

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

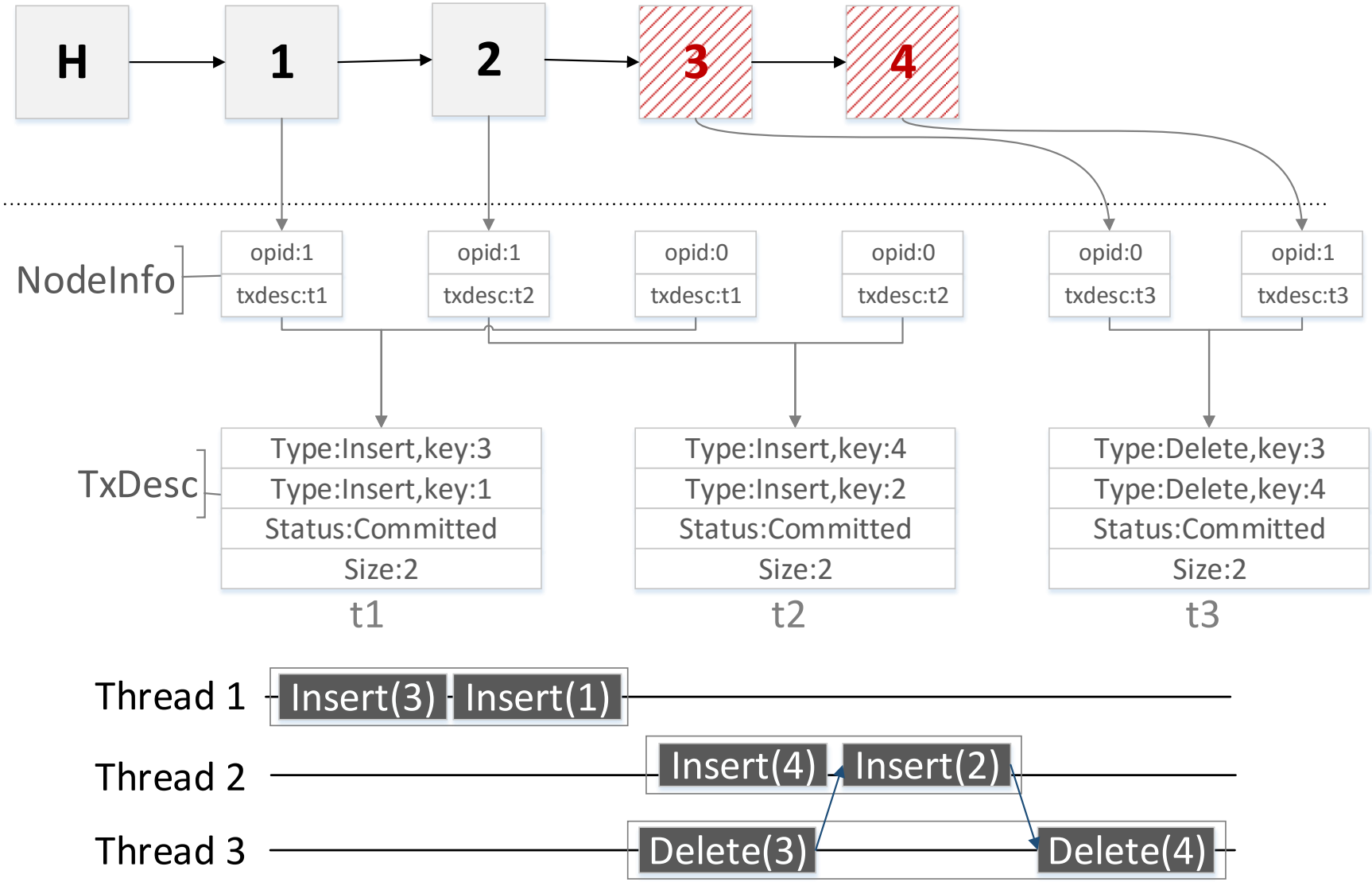
Cooperative Transaction Execution

- 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



LFTT Transaction Usage

Example

```
TxList set1;
TxSkiplist set2;
TxMDList set3;

TxDesc* desc = new TxDesc({
    new Set<int>::InsertOp(3, set1),
    new Set<int>::DeleteOp(6, set2),
    new Set<int>::FindOp(5, set1),
    new Set<int>::InsertOp(7, set3)});
desc->Execute();
```

Concurrent Dictionary

- Abstract Data Type
  - Supports INSERT, DELETE, and FIND
  - Keys are totally ordered
- Typical Implementations
  - Binary Search Trees (BST)
  - Skip-lists
  - Hash Tries

Concurrent Dictionary

- Abstract Data Type
  - Supports INSERT, DELETE, and FIND
  - Keys are totally ordered
- Typical Implementations
  - Binary Search Trees (BST)
  - Skip-lists
  - Hash Tries
- Improve baseline performance
  - Offset transactional synchronization overhead

Lock-free BST

- Node not found $\neq$ Key not found
  - Embed logical ordering [Drachsler et al. 2014]
  - Use external (leaf-oriented) trees [Ellen et al. 2010]
- Balancing induces bottleneck
  - Relaxed balancing [Bronson et al. 2010]
  - Background balancing [Crain et al. 2013]

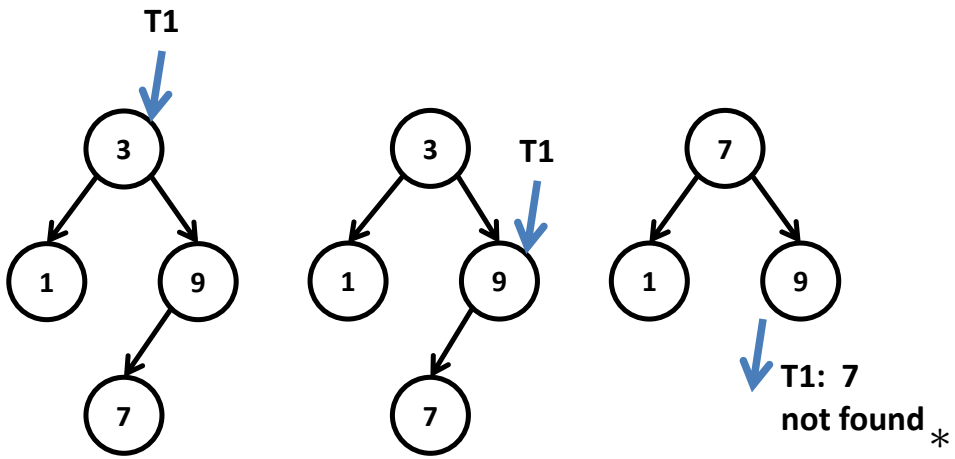
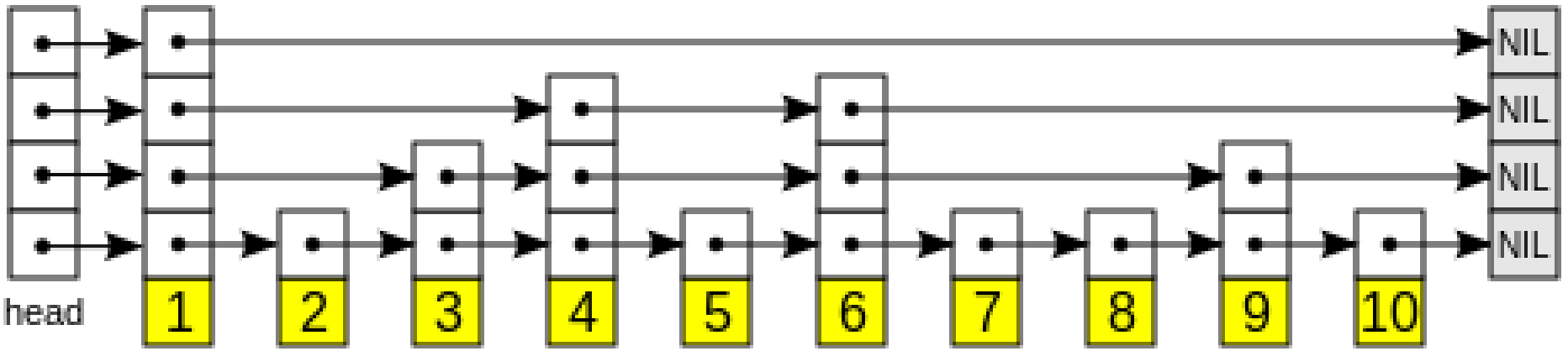


*Illustration from Drachsler et al. 2014

Lock-free Skip-list

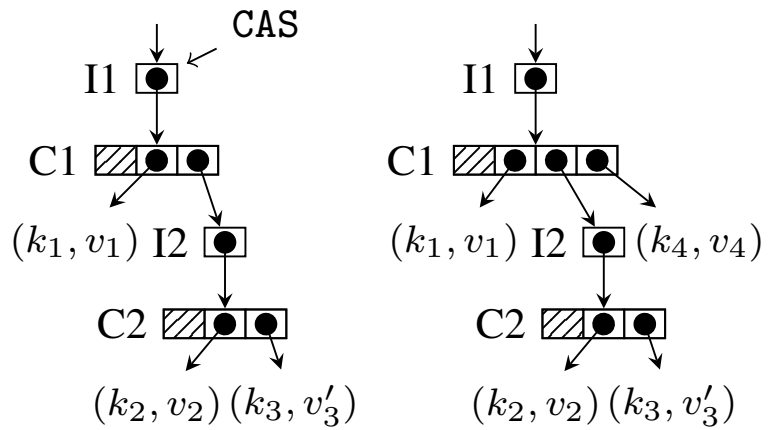
- Probabilistically balanced alternative
  - Bottom level linked list
  - Redundant short-cut links
  - Exponentially lower probability
- Concurrency limitation [Fraser 2004, Crain et al. 2013]
  - INSERT/DELETE update multiple nodes



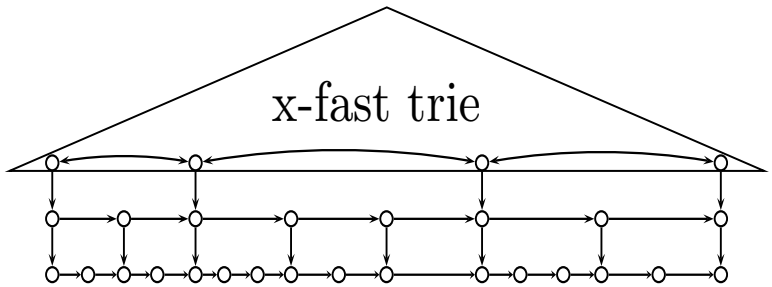
† Illustration from wikipedia

Lock-free Tries

- K-ary prefix tree
  - Bounded key space U
  - Fast sequential search $\log \log U$
  - Value stored externally
- Ctrie [Prokopec et al. 2012]
  - A hash array mapped trie
  - Hot spot: Intermediate (I) nodes
- SkipTrie [Oshman and Shavit 2013]
  - Y-fast trie with skip-list
  - Require double-word CAS
  - Hot spot: inserting prefixes



Ctrie Insert



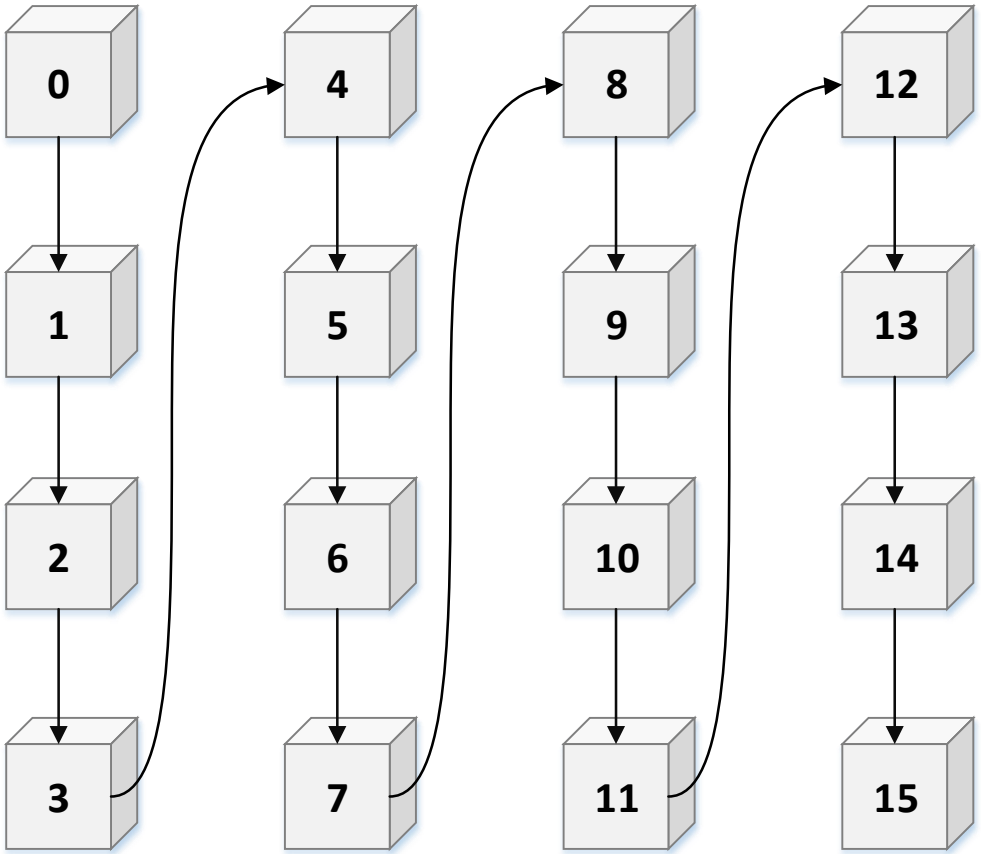
SkipTrie‡

‡ Illustration from Prokopec et al. 2012, and Oshman and Shavit 2013

What makes an efficient lock-free dictionary?

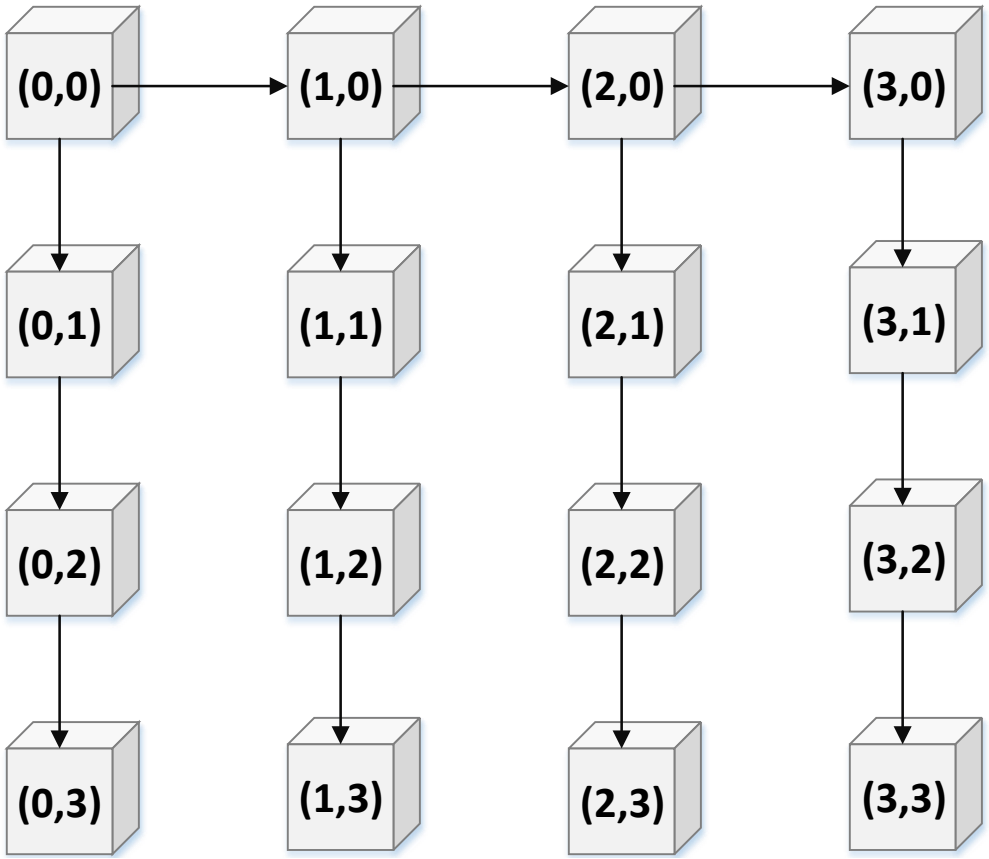
- To maximize disjoint access parallelism
  - Low branching factor — avoids hot spots
  - Localized updates — reduces access conflicts
  - Fixed key positions — simplifies predecessor query
- Proposed Multi-dimensional List (MDList)
  - Lower level nodes have smaller branching factor
  - INSERT/DELETE modifies at most 2 adjacent nodes
  - Unique layout per logical ordering
  - No balancing/randomization required
  - Memory efficient

Ordered Linked List



- Re-arranged into columns
- Partitioned sub-lists
- $\mathcal{O}(N)$ FIND

Ordered 2D List



- Short-cut links on top
- Vector coordinates (d_0, d_1)
- Lexicographical ordering
- Worst-case $\mathcal{O}(\sqrt{N})$ FIND

Multi-dimensional List

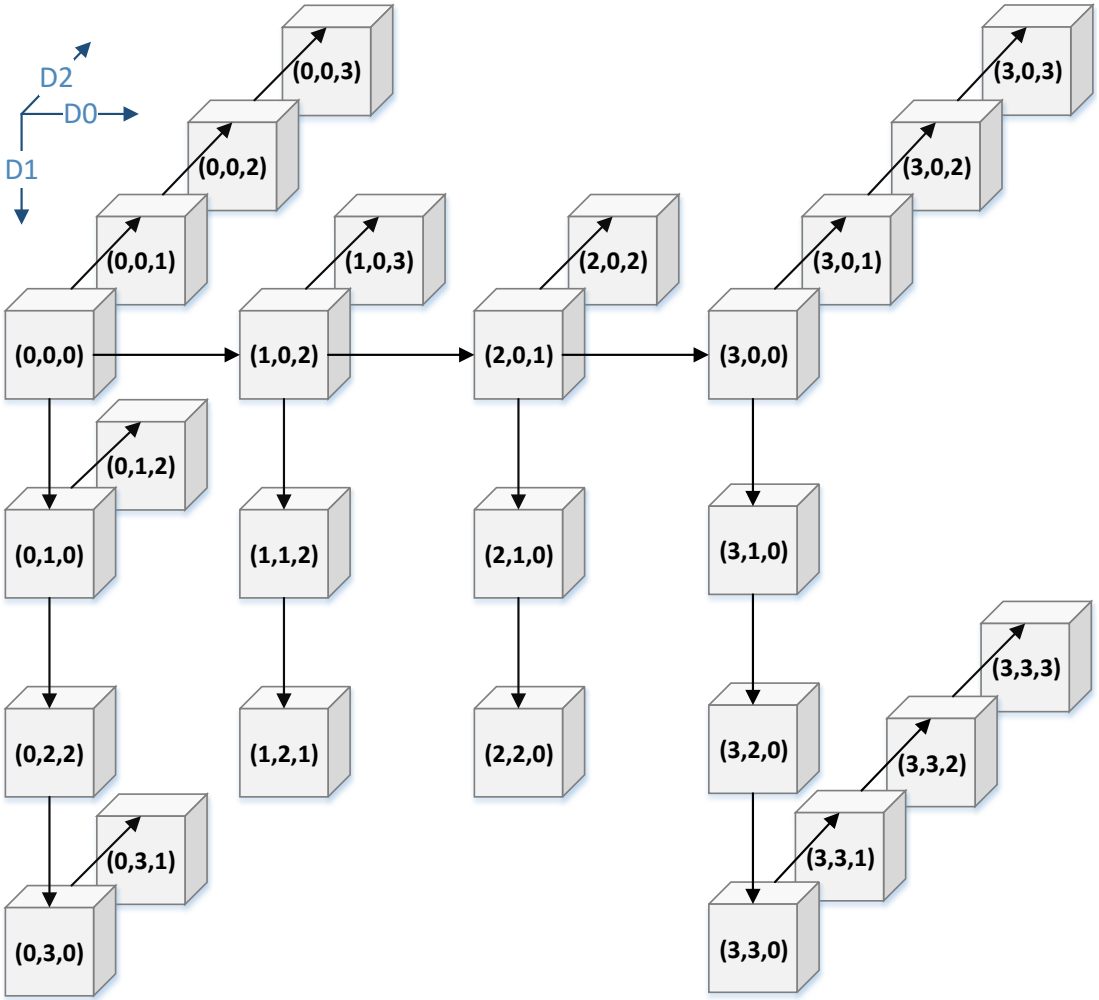
Definition

A D-dimensional list is a tree in which each node is implicitly assigned a dimension of $d \in [0, D)$. The root node's dimension is 0. A node of dimension d has no more than $D - d$ children, where the mth child is assigned a dimension of $d' = d + m - 1$.

Definition

Given a non-root node of dimension d with coordinate $\mathbf{k} = (k_0, ..., k_{D-1})$ and its parent with coordinate $\mathbf{k}' = (k'_0, ..., k'_{D-1})$ in an ordered D-dimensional list: $k_i = k'_i, \ \forall \ i \in [0, d) \wedge k_d > k'_d$.

Coordinate Mapping



- $k \mapsto (d_0, d_1, ..., d_{D-1})$
- *Injective* and *monotonic*
- Preferably uniform
- Base conversion: choose $base = \lceil \sqrt[D]{U} \rceil$

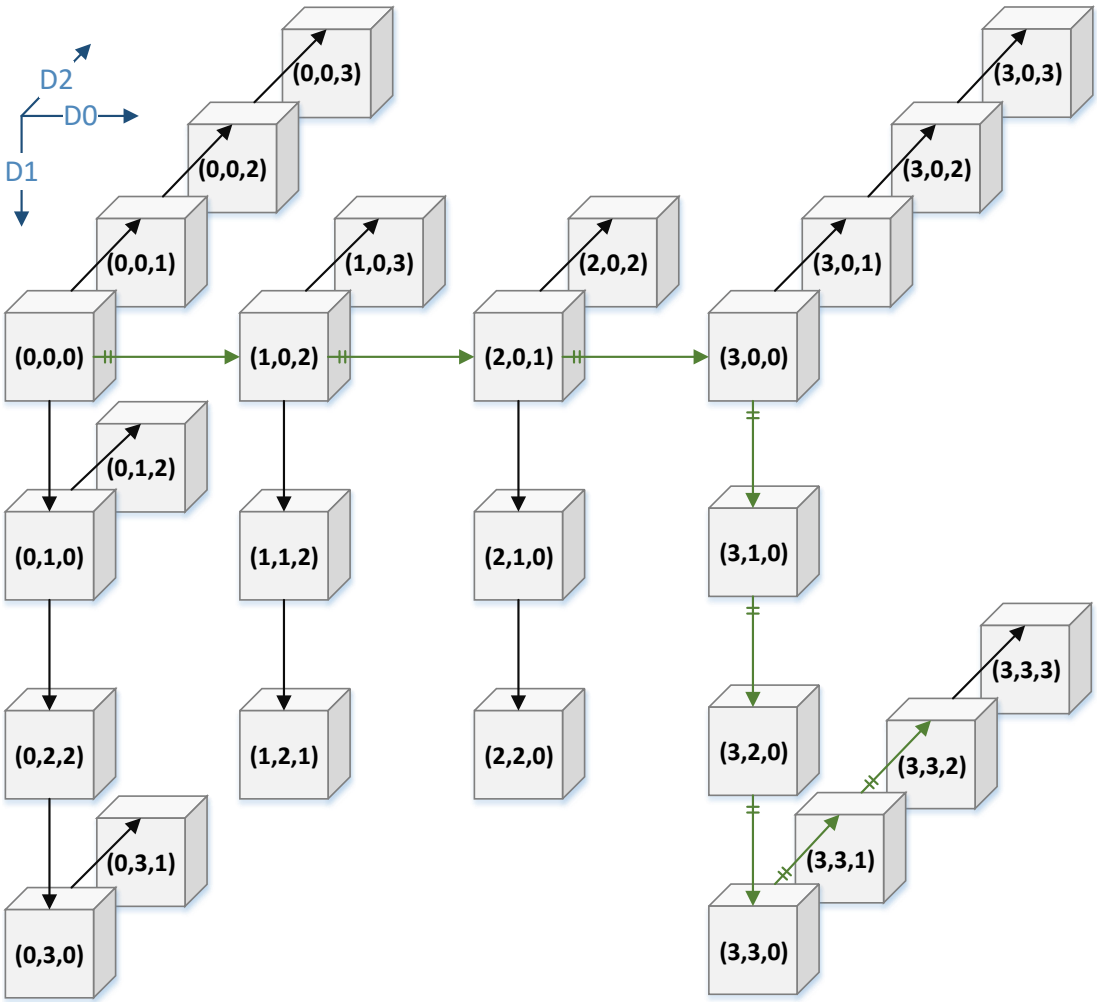
Example

$U = 64, D = 3, base = 4$

$(63)_{10} = (333)_4$

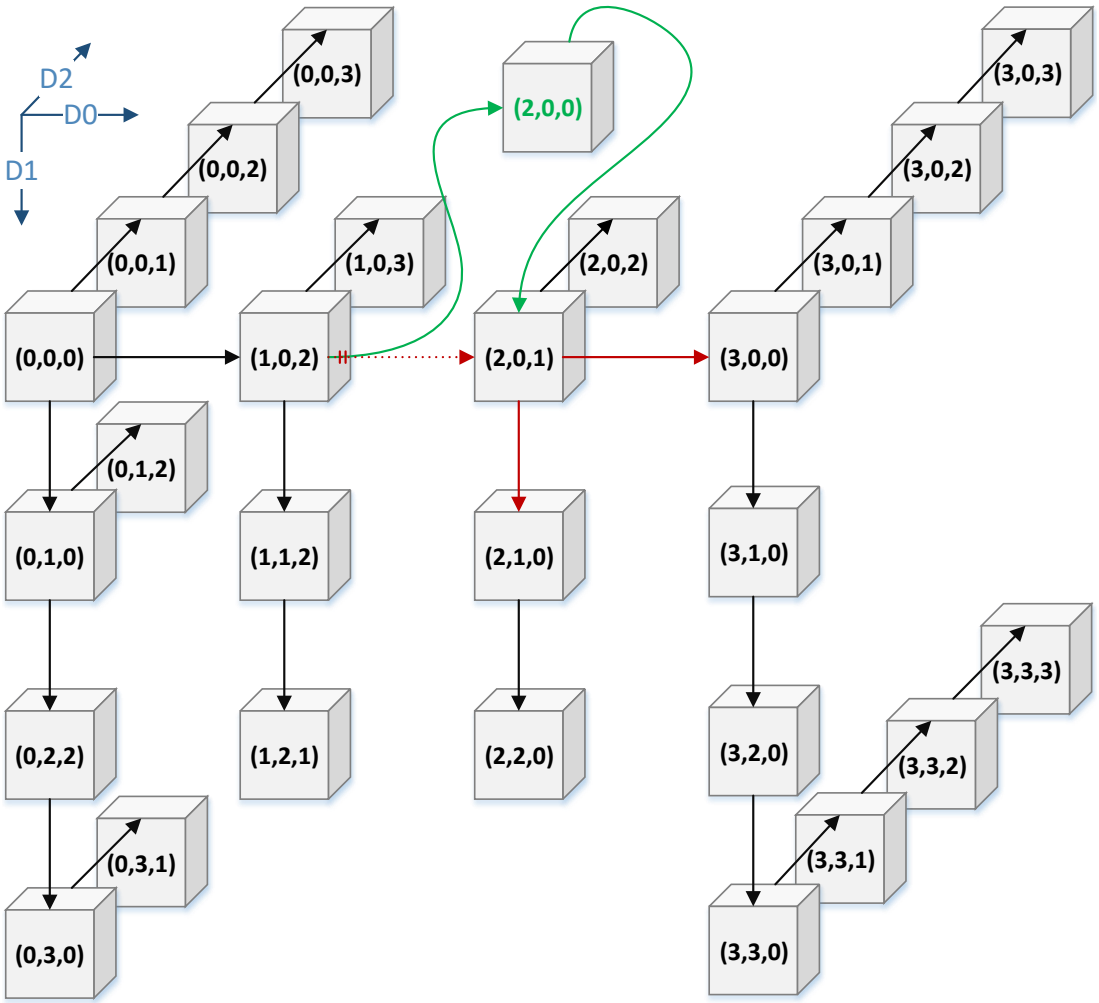
$(34)_{10} = (202)_4$

Find Operation



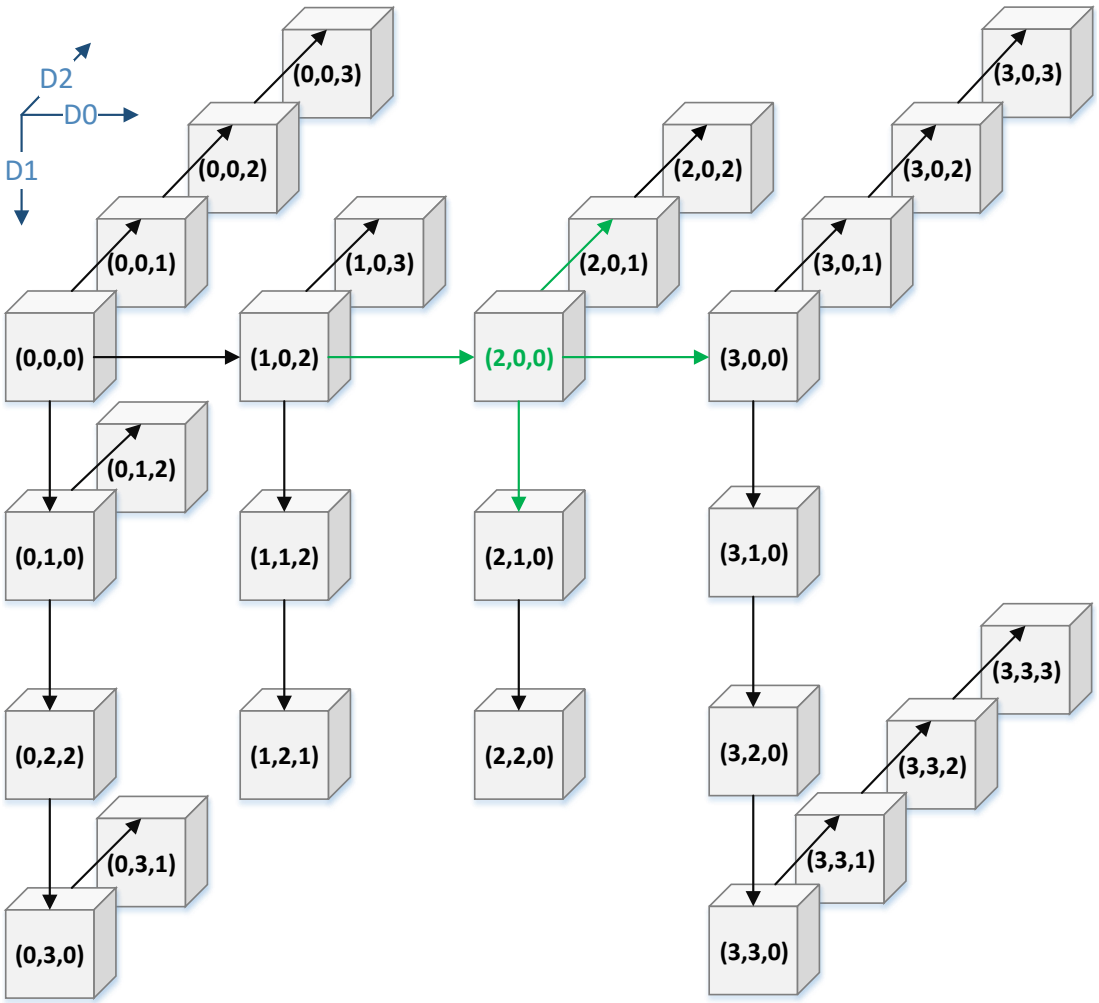
- Recursively traverse sub-lists
- Comparing coordinates from $d = 0$
- Worst-case $\mathcal{O}(D \sqrt[D]{U})$ time
- Choose $D = \log U$ then $\mathcal{O}(\log U \sqrt[\log U]{U}) = \mathcal{O}(\log U)$

Insert — 2 Steps



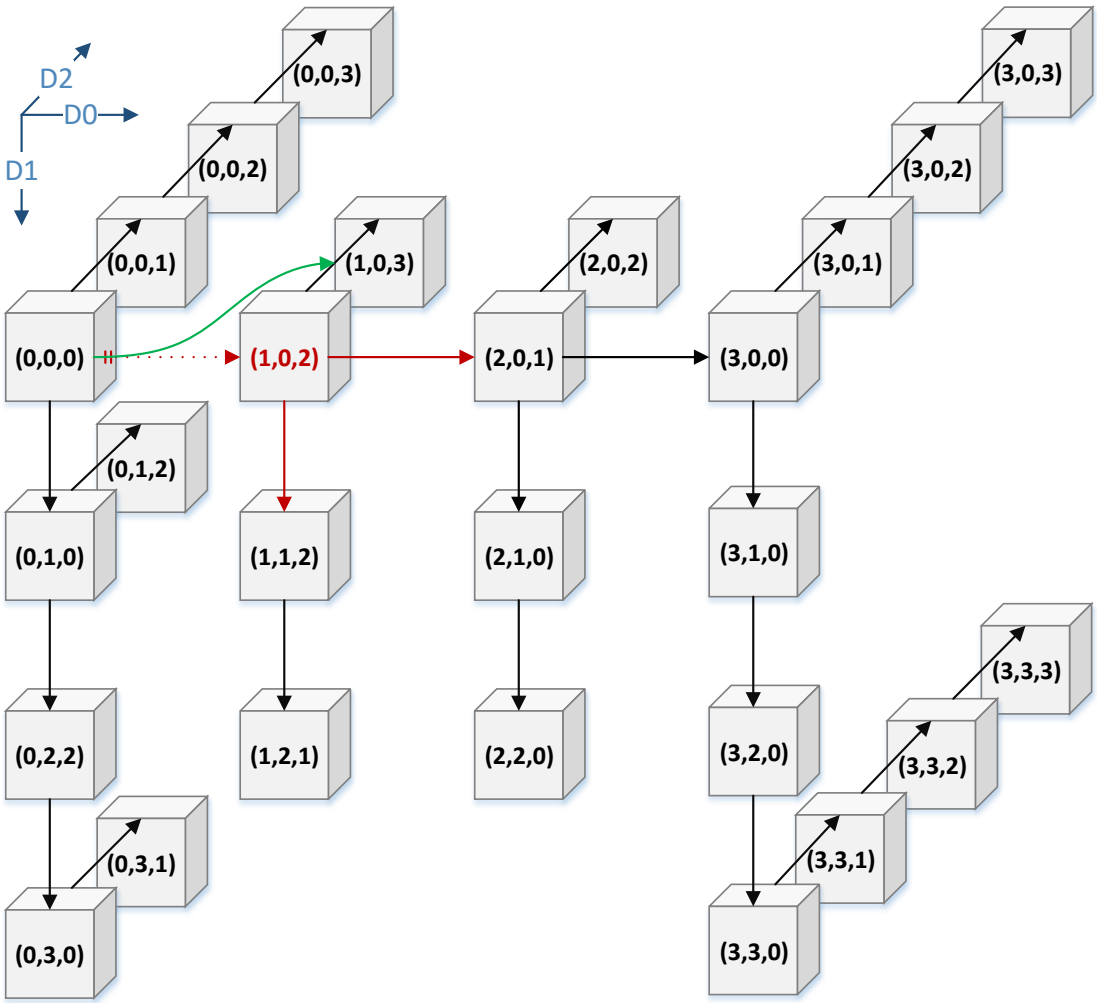
1. Node Splicing
   - Predecessor query locate $pred, curr, d_p, d_c$
   - CAS updates $pred.child[d_p]$

Insert — 2 Steps



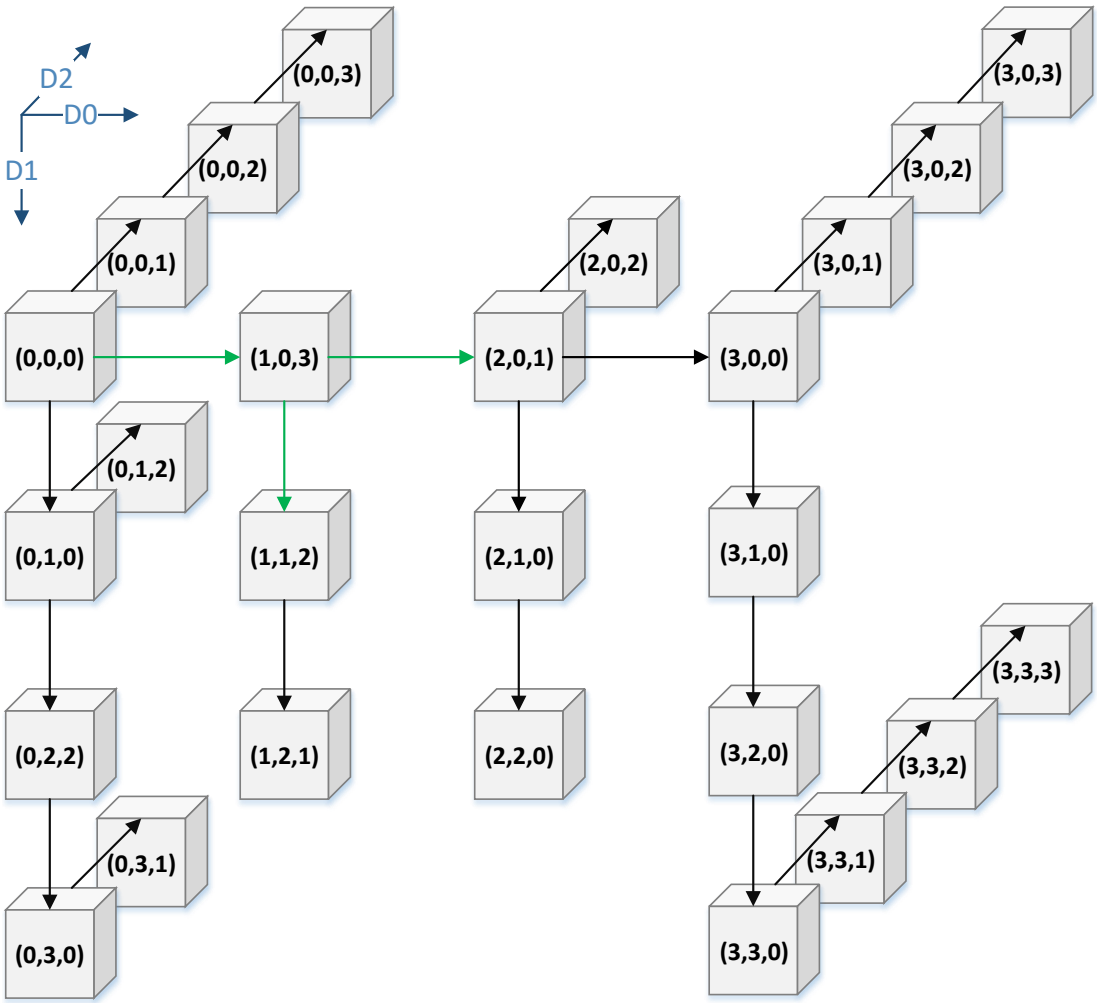
1. Node Splicing
   - Predecessor query locate $pred, curr, d_p, d_c$
   - CAS updates $pred.child[d_p]$
2. Child adoption
   - Necessary if $d_c \neq d_p$
   - $curr.child[d_p : d_c]$ transferred to new node
   - Use descriptor for helping in case process is delayed

Normal Delete — 2 steps



1. Prompt the next node
   - Examine coordinates from $D - 1$
   - Swing *pred*'s pointer

Normal Delete — 2 steps



1. Prompt the next node
   - Examine coordinates from $D-1$
   - Swing *pred*'s pointer
2. Child transfer
   - Inverse of child adoption
   - Potential conflicts

Normal Delete Does Not Work

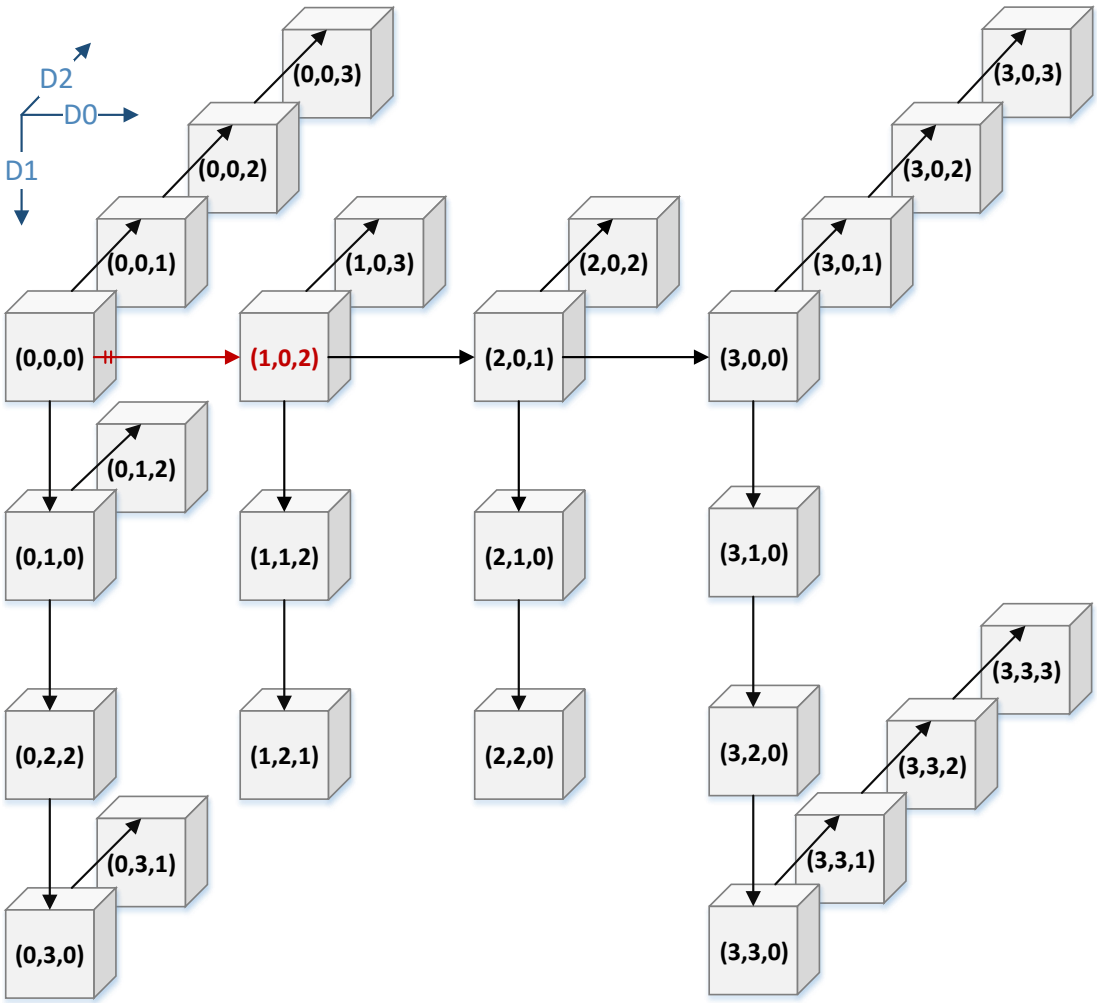
- INSERT may demote a node (i.e., increase its dimensionality)
- DELETE may promote a node (i.e., decrease its dimensionality)

Synchronization Issue

Due to the helping mechanism, several threads may execute child adoption on the same node. Without additional synchronization, we cannot know if all of them have finished. Data races may arise among ongoing child adoption and child transfer processes.

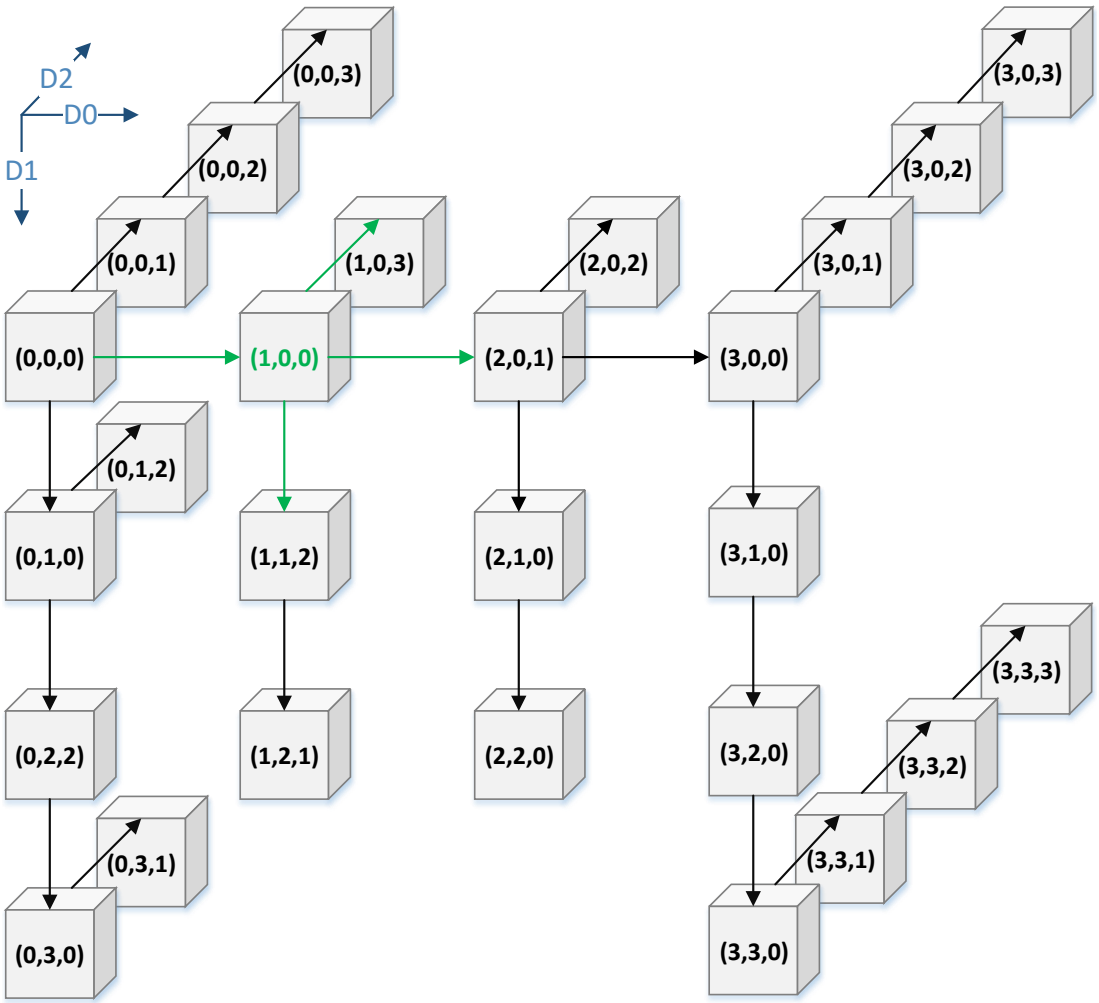
- Solution: keep dimensionality change *unidirectional*

Asymmetrical Delete — Decoupled Physical Removal



1. Mark for logical deletion
   - Still valid for routing

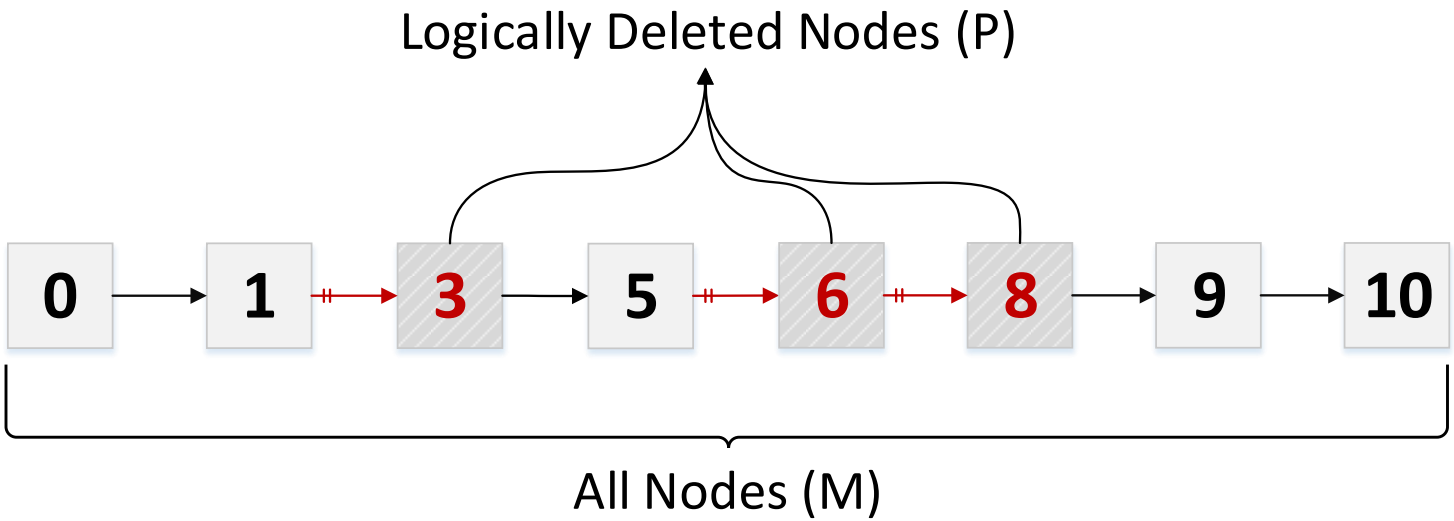
Asymmetrical Delete — Decoupled Physical Removal



1. Mark for logical deletion
   - Still valid for routing
2. INSERT purge marked node
   - Adopt children in $[d_p, D)$

- Reuse child adoption
- Unifies help protocol
- Simplifies synchronization

Abstract State



- The abstract state of the dictionary $S = M \setminus P$
- Linearization points
  - INSERT: when CAS updates predecessor's child pointer
  - DELETE: when CAS marks predecessor's child pointer
  - FIND: when predecessor's child pointer is read

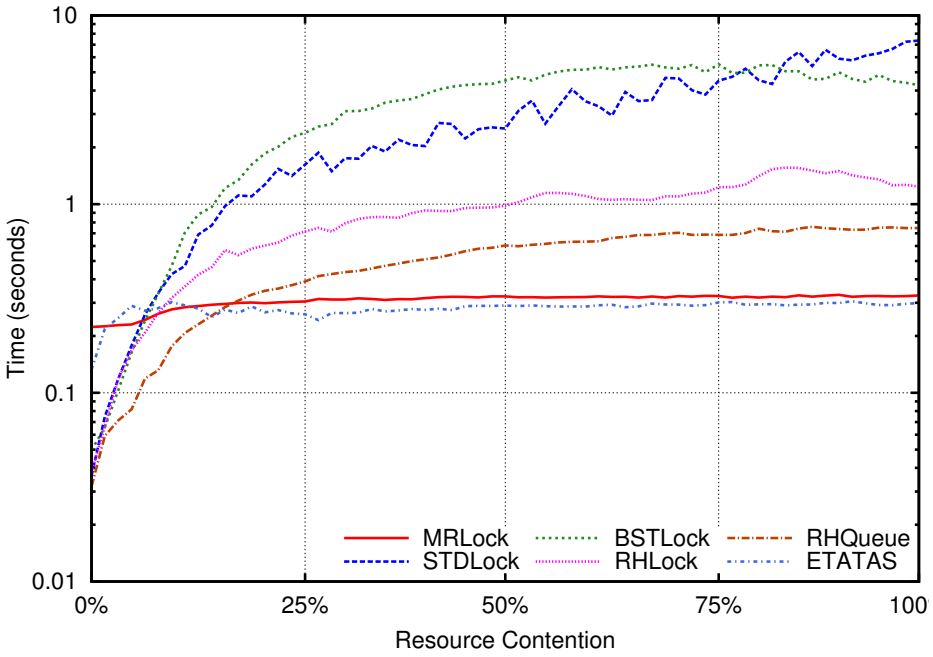
Environment

- Hardware
  - 64-core NUMA (4 AMD Opteron @2.1GHz)
  - 6-core (12 w/ Hyper-threading) SMP (Intel Xeon @2.9GHz)
- Micro-benchmark
  - GCC 4.7 w/ O3
  - MRLock: randomly acquire up to 64 or 1024 resources
  - LFTT/MDList: Write-dominated, read-dominated, and mixed workloads

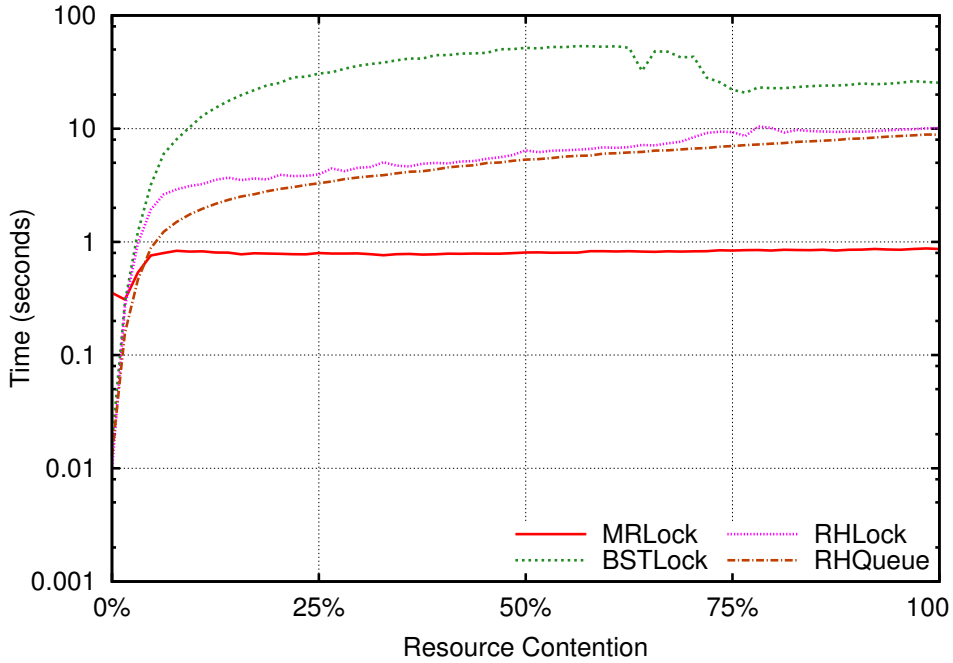
Alternatives

- Two-phase locking
  - std::lock function with std::mutex (STDLock)
  - boost::lock function with boost::mutex (BSTLock)
- Resource hierarchy
  - with std::mutex (RHSTD)
  - with tbb::queue_mutex (RHQueue)
- Extended TATAS (ETATAS)

16 Threads



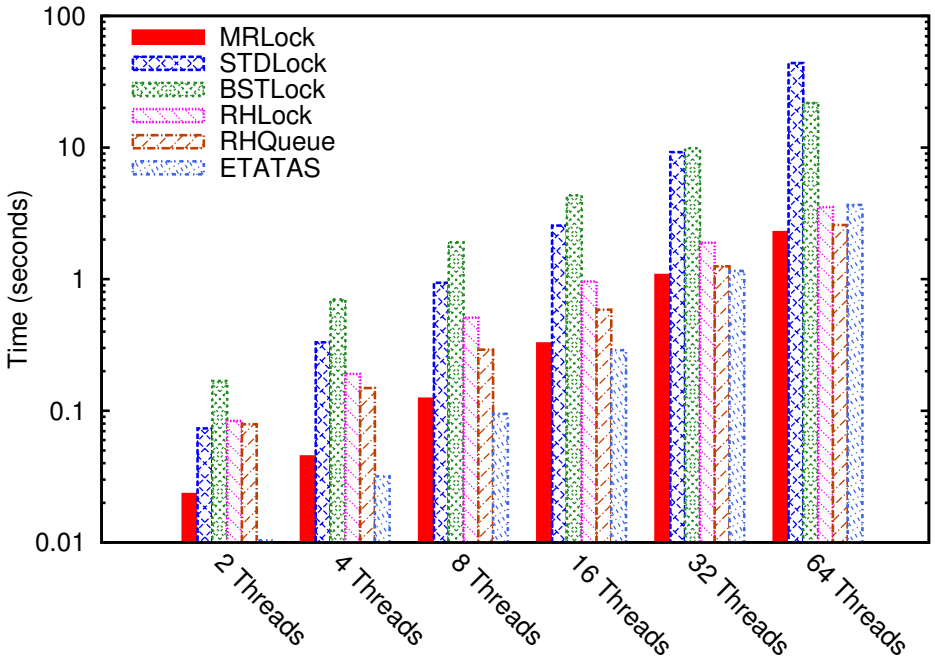
64 Resources



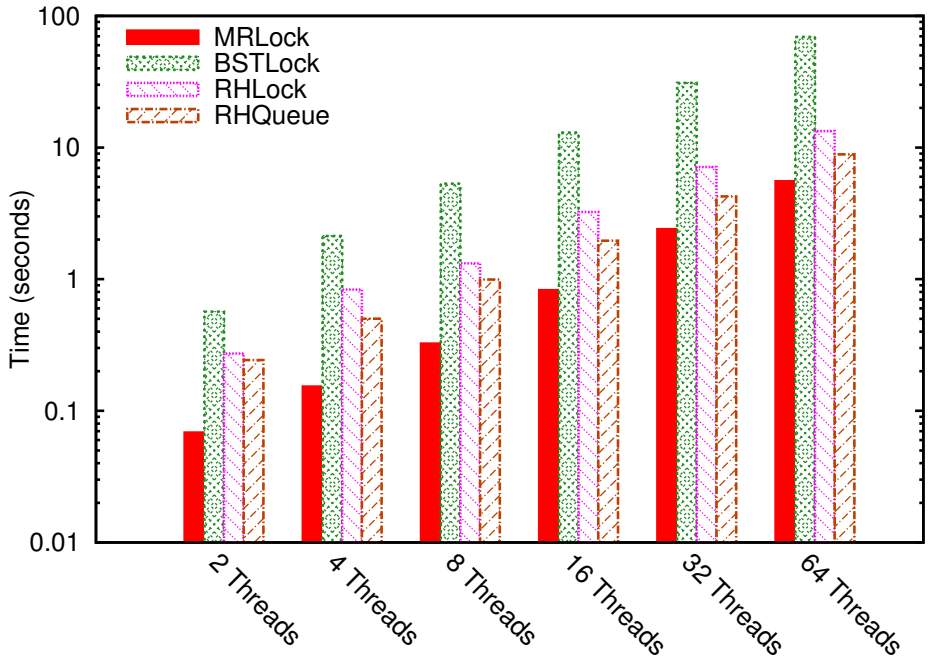
1024 Resources

- Outperforms RHQueue by up to 3 times for 64 resources
- Outperforms RHQueue by up to 5 times for 1024 resources
- Execution time independent from resource contention

Thread Scale



Contention 32/64 (50%)



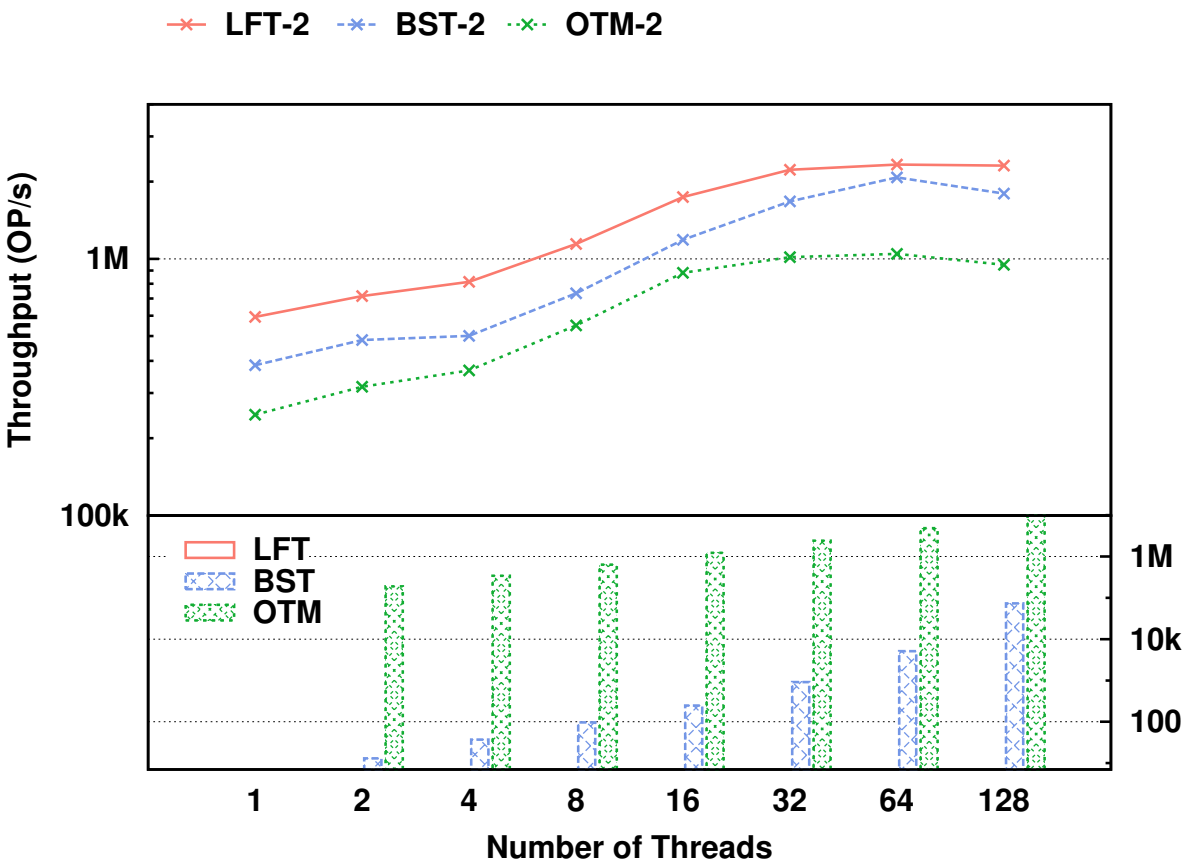
Contention 128/1024 (12.5%)

- Begins to outperform extended TATAS when thread increases
- Constantly outperforms alternatives for 1024 resources

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)

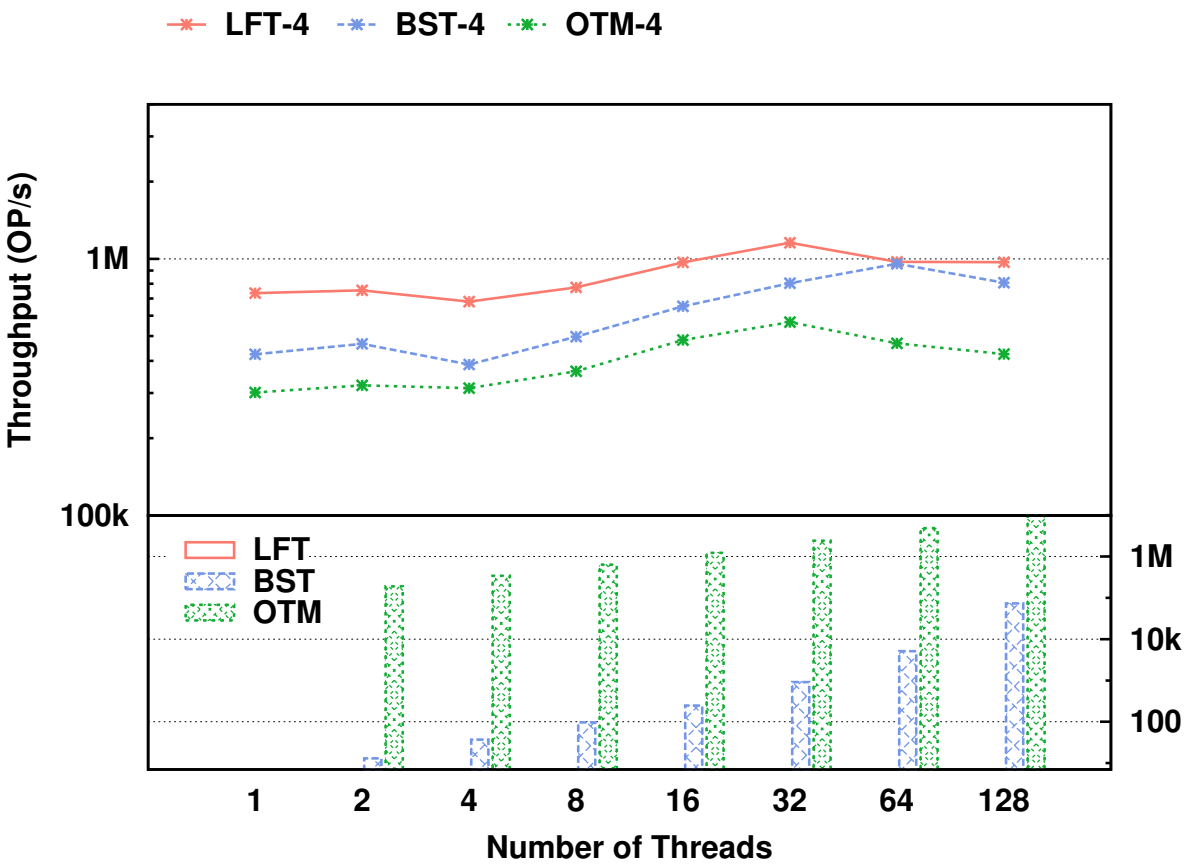
Throughput — Skiplist Mixed Workload



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

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

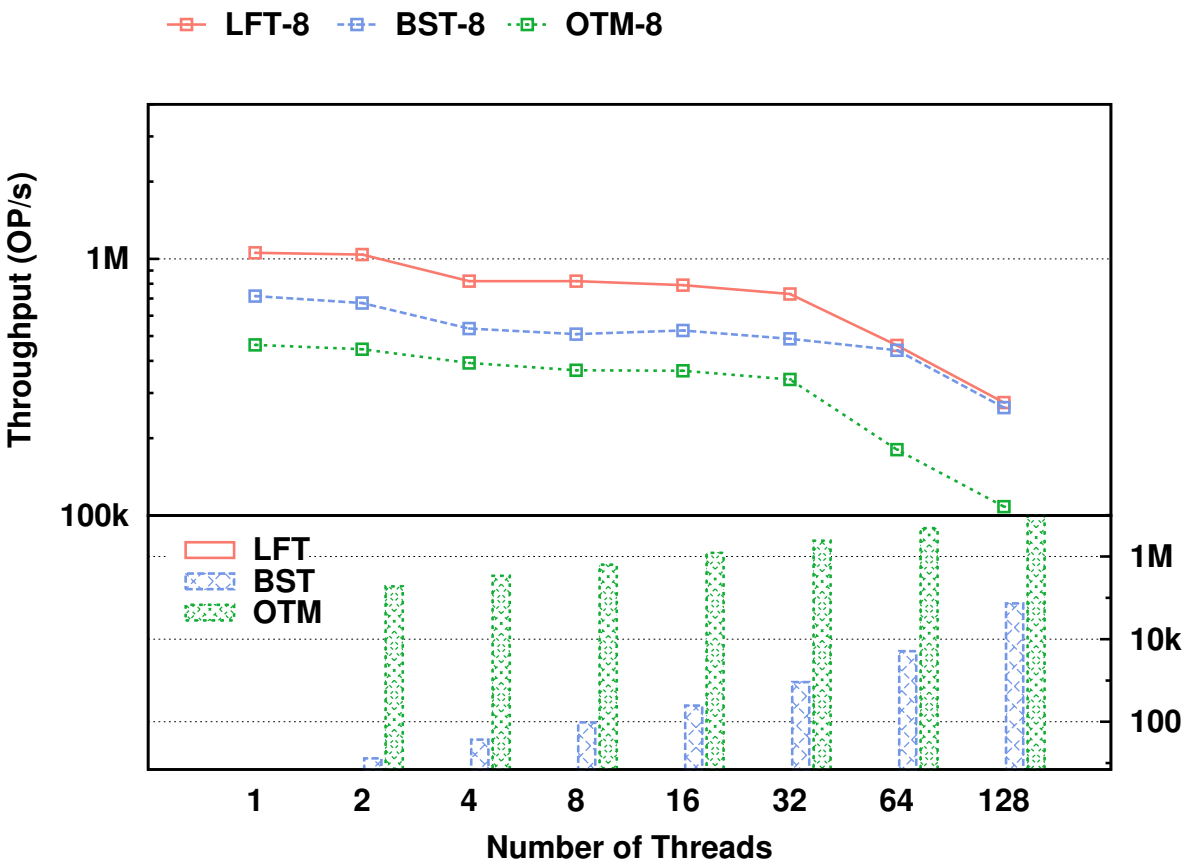
Throughput — Skiplist Mixed Workload



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

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

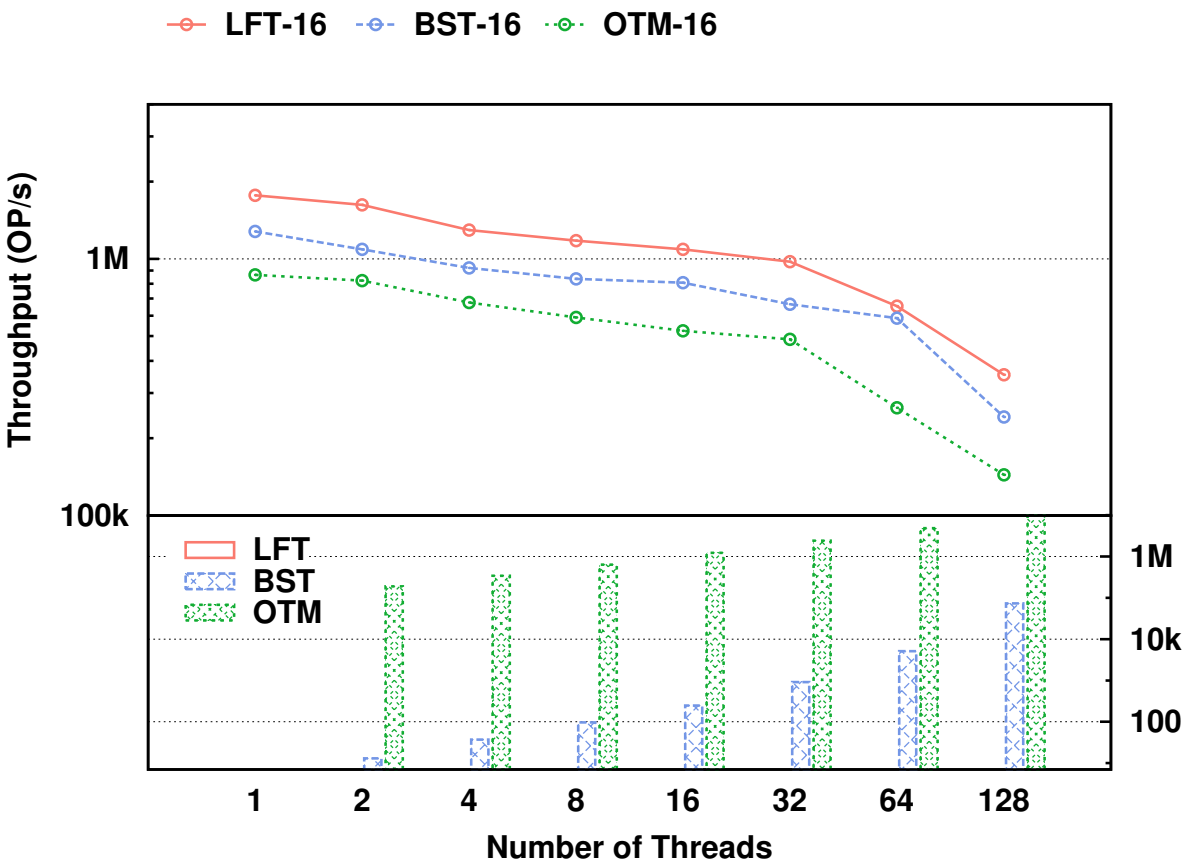
Throughput — Skiplist Mixed Workload



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

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

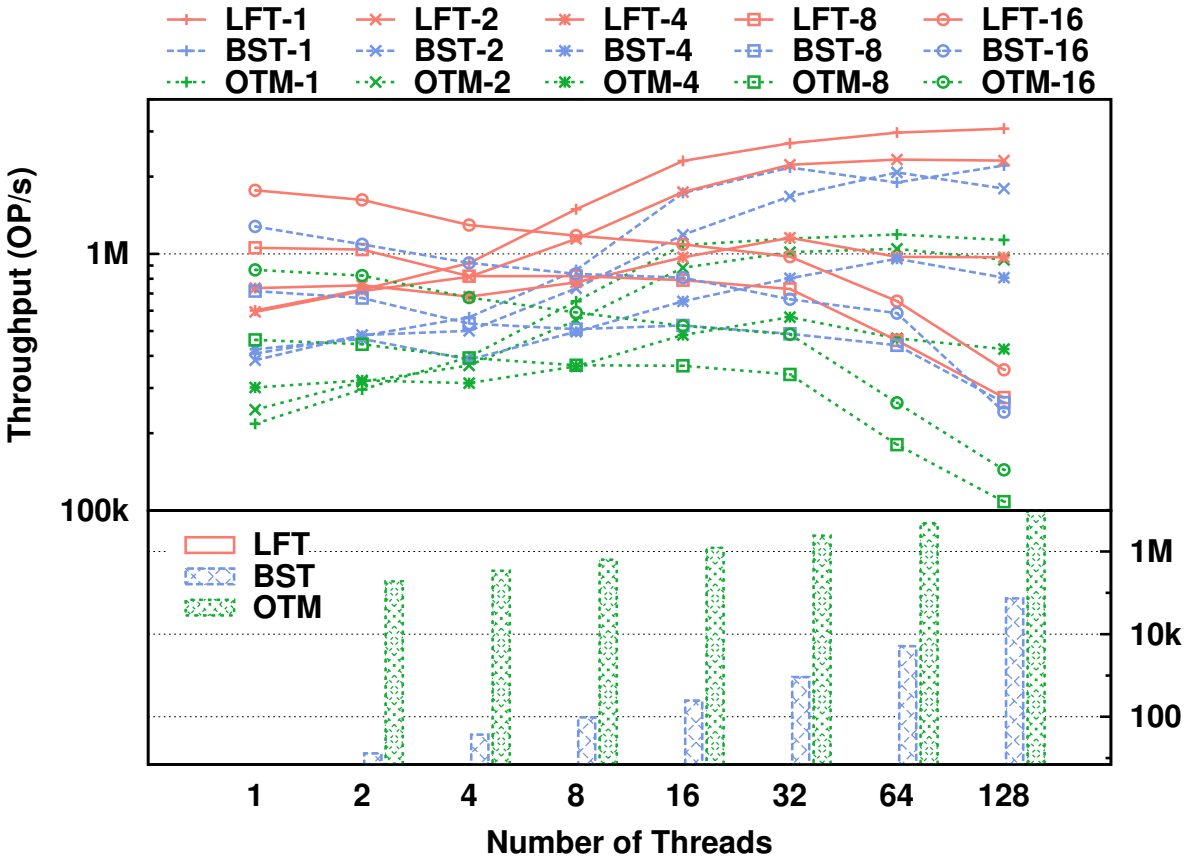
Throughput — Skiplist Mixed Workload



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

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

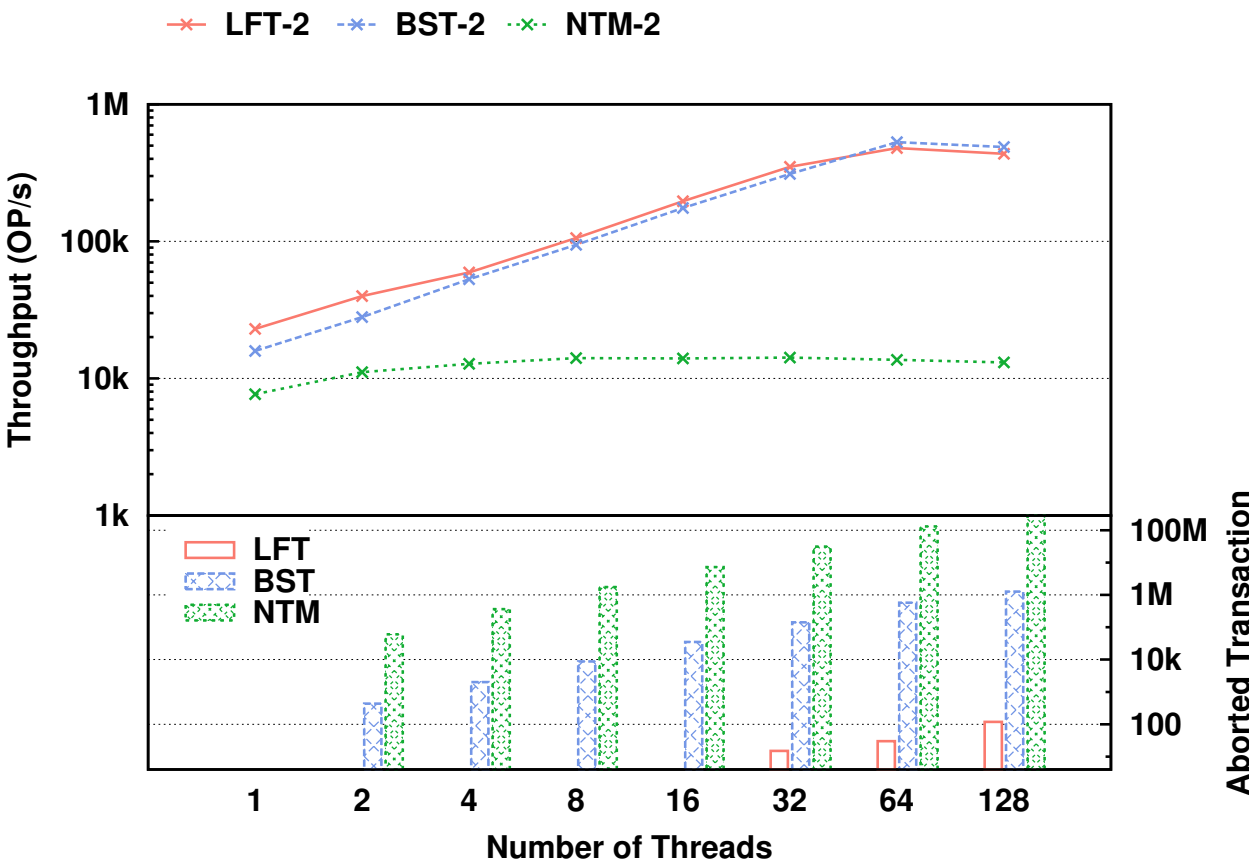
Throughput — Skiplist Mixed Workload



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

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

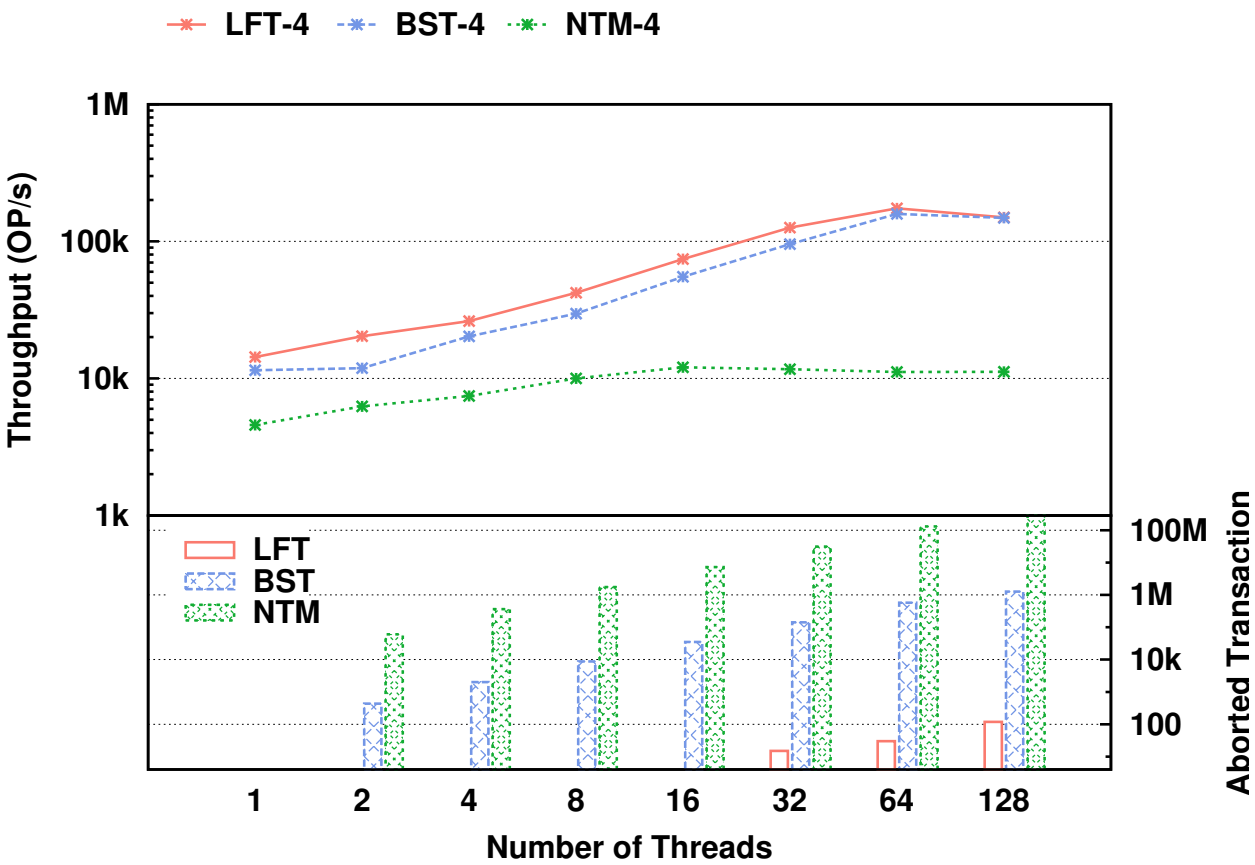
Throughput — Linked List Mixed Workload



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

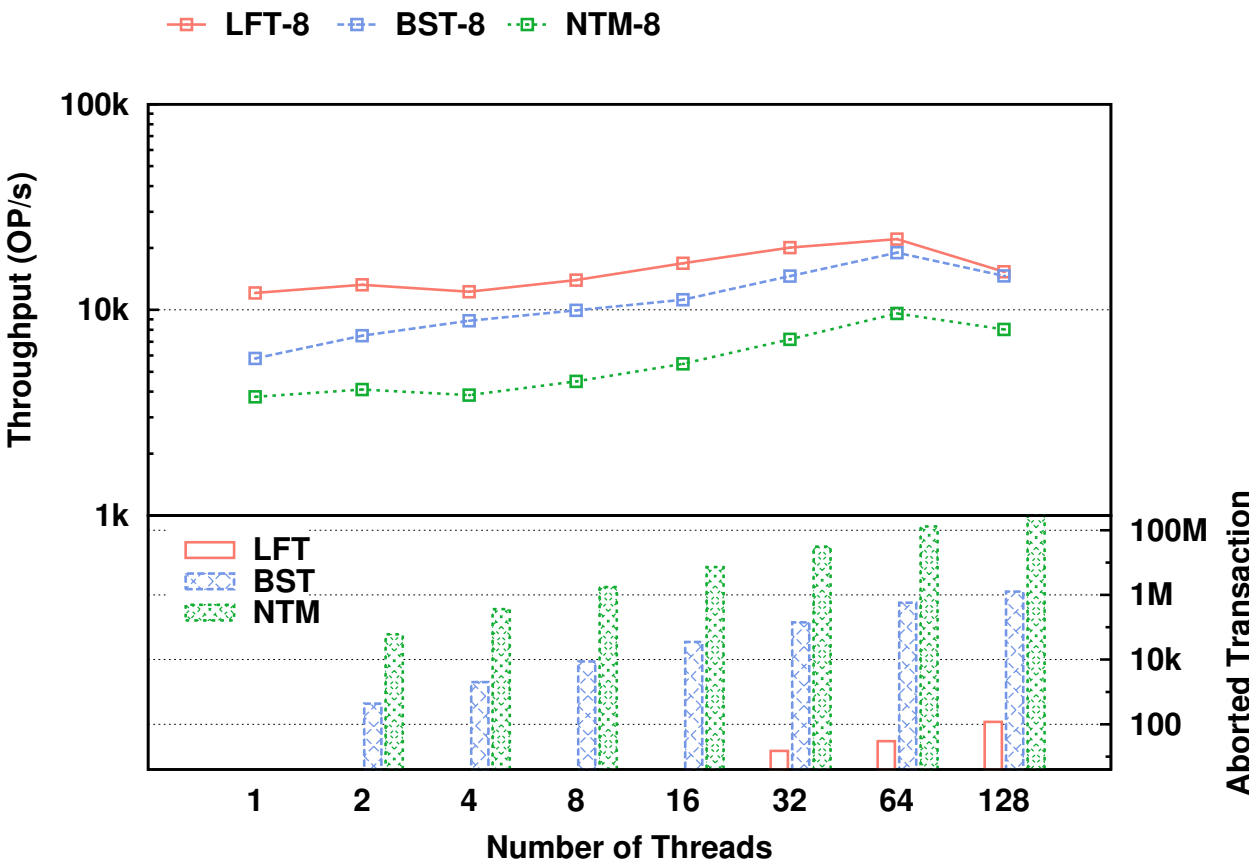
Throughput — Linked List Mixed Workload



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

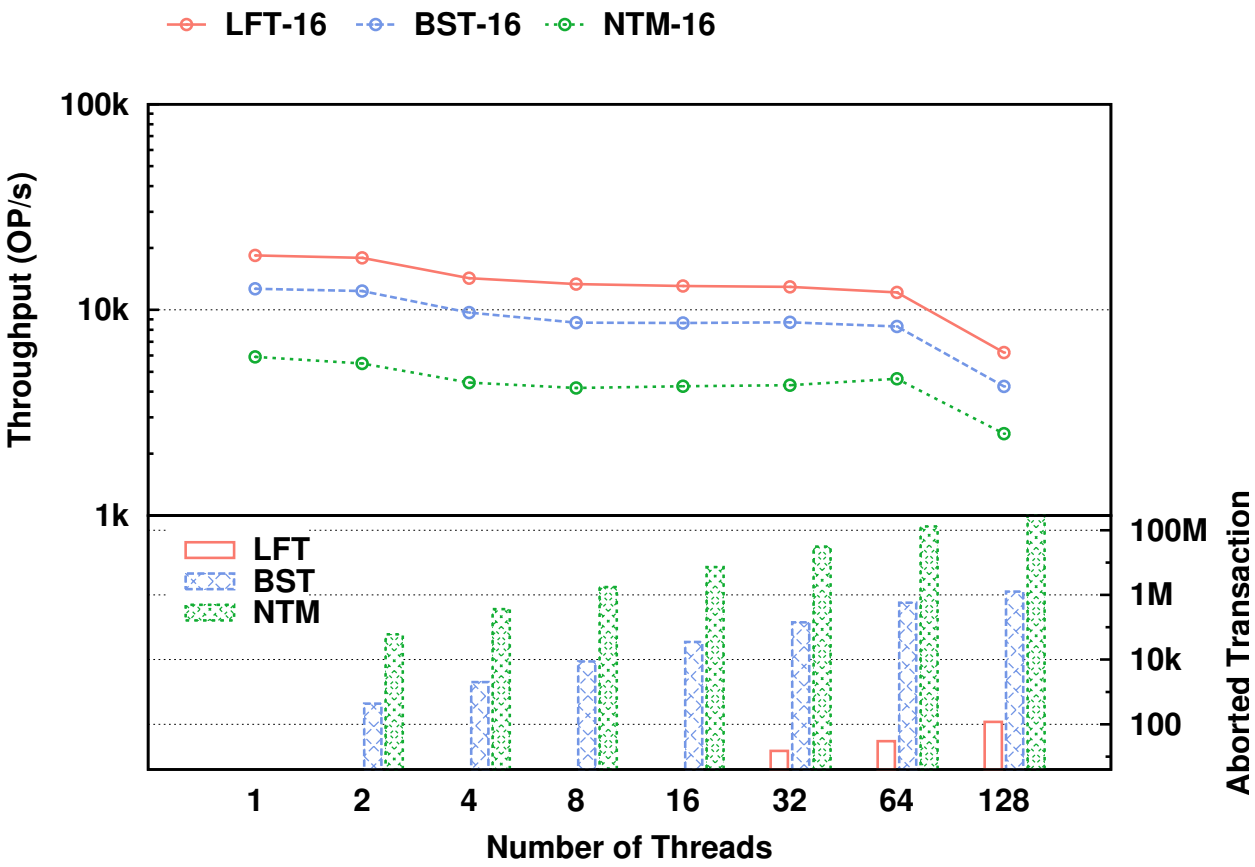
Throughput — Linked List Mixed Workload



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

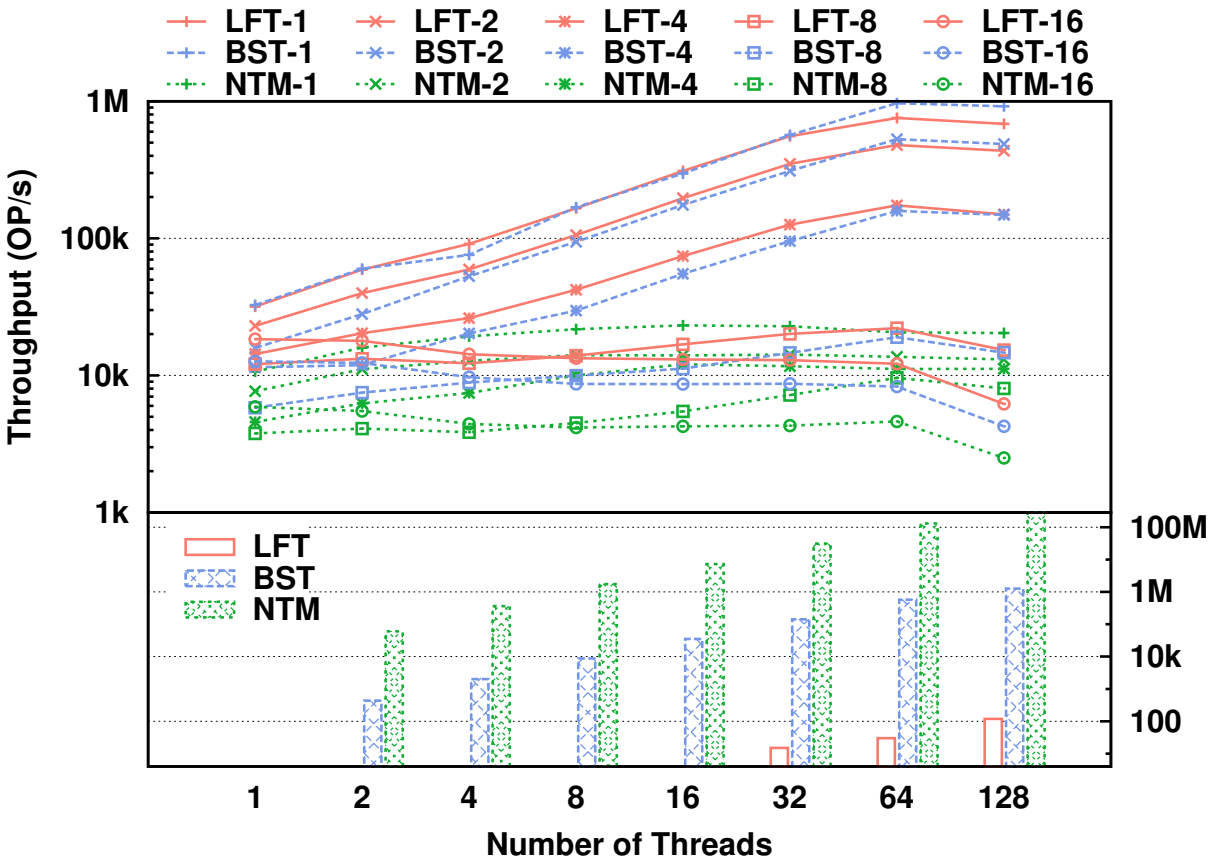
Throughput — Linked List Mixed Workload



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

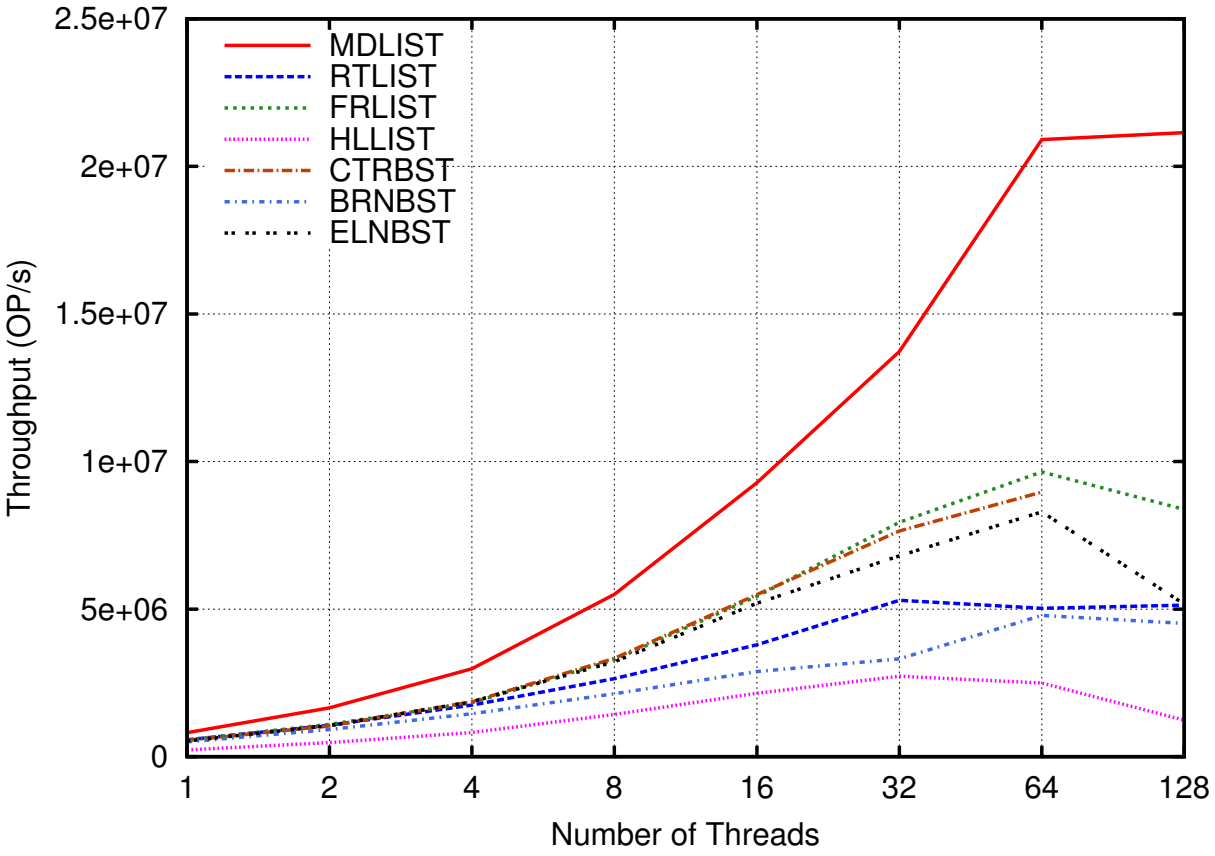
- Average of 40% speedup over boosting
- 10 times speedup over STM
- 2 to 3 orders of magnitude less spurious aborts

Alternatives

- BST
  - Lock-based relaxed AVL by Bronson et al. (BRNBST)
  - Lock-free unbalanced BST by Ellen et al. (ELNBST)
  - RCU-based Citrus tree by Arbel et al. (CTRBST)
- Skip-list (max tower height 30)
  - Lock-free skip-list by Fraser (FRLIST)
  - Lock-free skip-list by Herlihy (HLLIST)
  - Lock-free rotating skip-list by Dick et al. (RTLIST)
- MDList (dimension 16)

‡BRNBST, ELNBST, and HLLIST are based on C++ implementation by Wicht et al. 2012

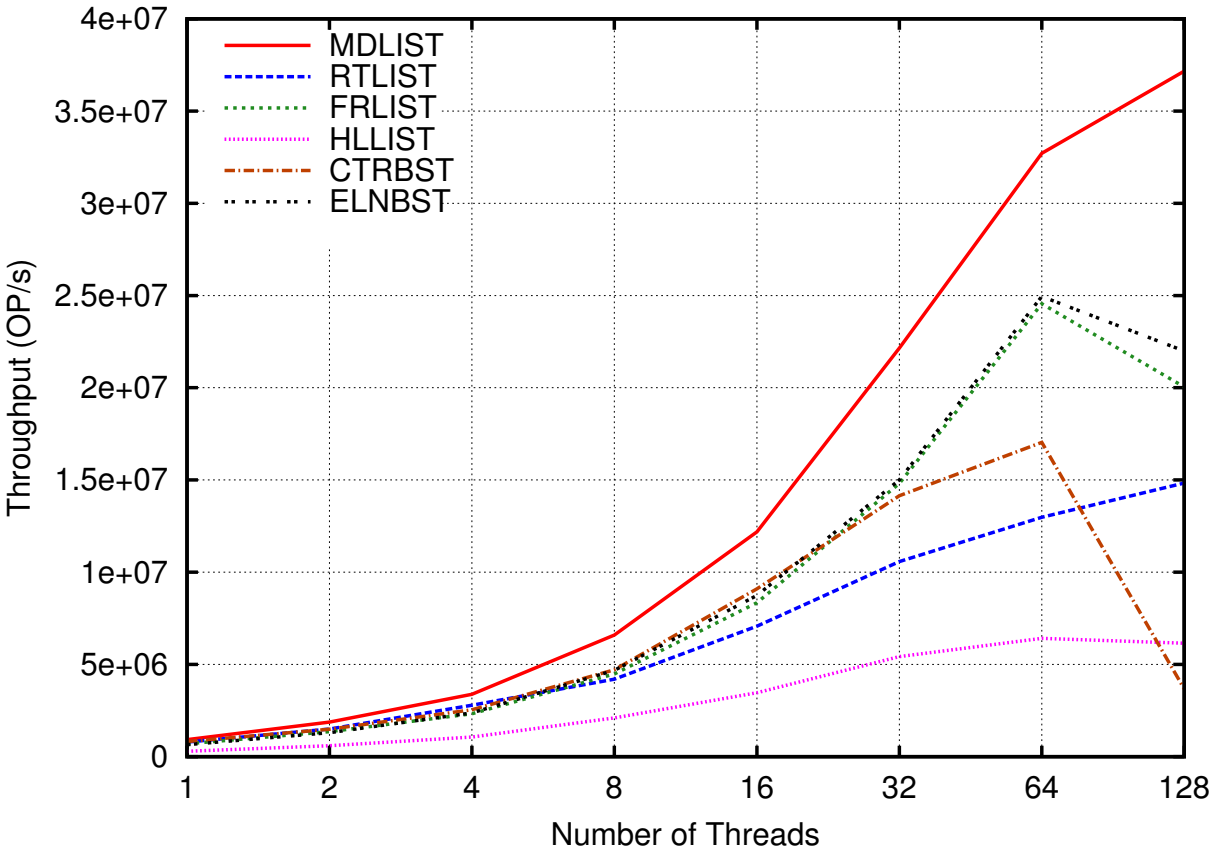
Throughput — NUMA Write-dominated Workload



1G Keys, 50% Insert 50% Delete

- As much as 100% speedup for 64 threads
- Optimized by localized node modifications

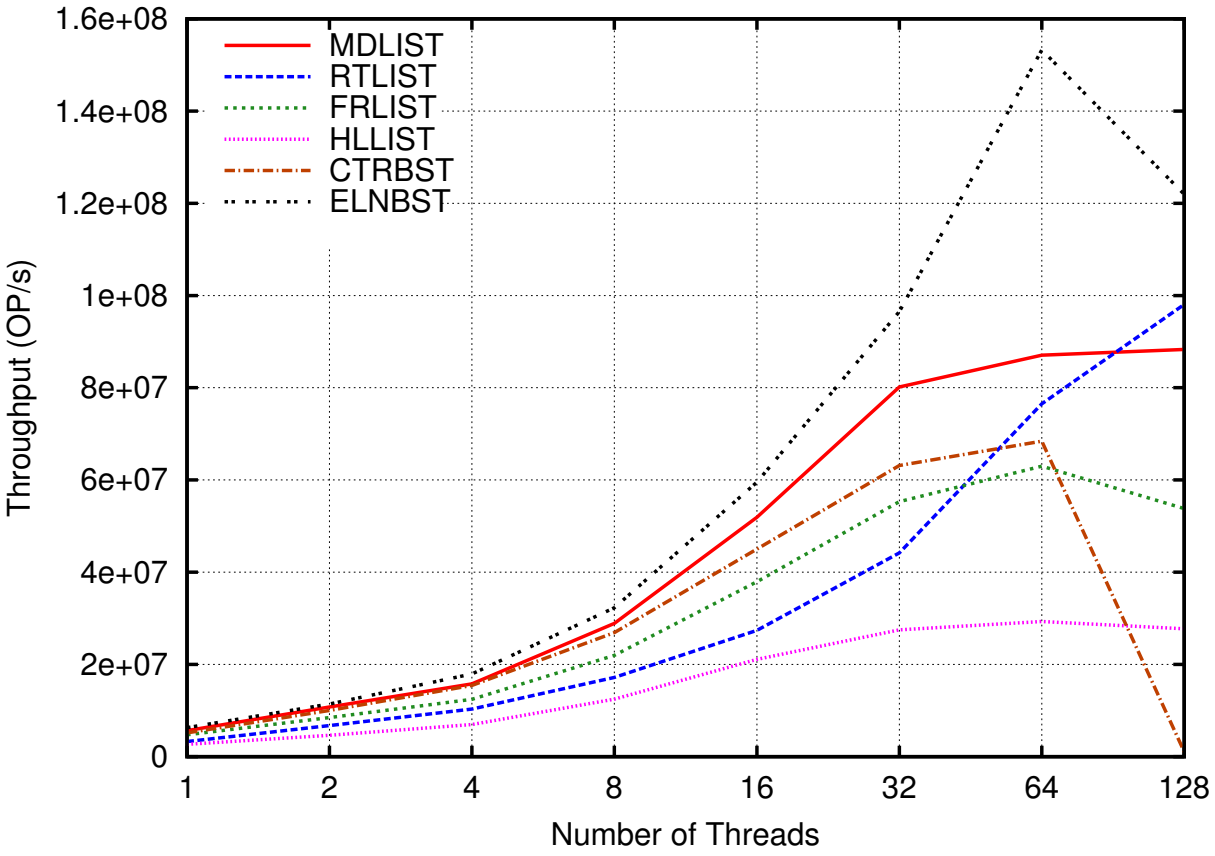
Throughput — NUMA Mixed Workload



1M Keys, 20% Insert 10% Delete 70% Find

- As much as 50% speedup for 64 threads

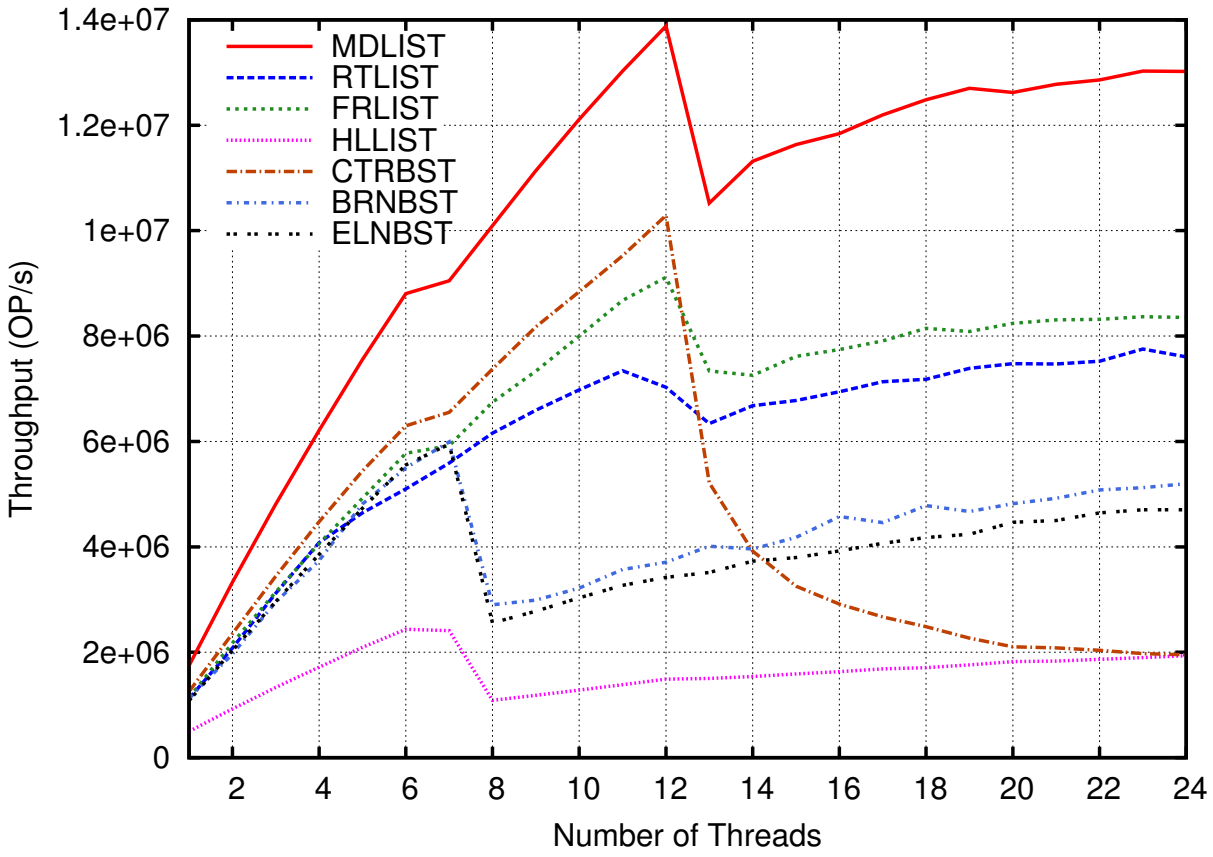
Throughput — NUMA Read-dominated Workload



1K Keys, 9% Insert 1% Delete 90% Find

- BSTs have shallow depth
- 16 dimension is too much for 1000 keys

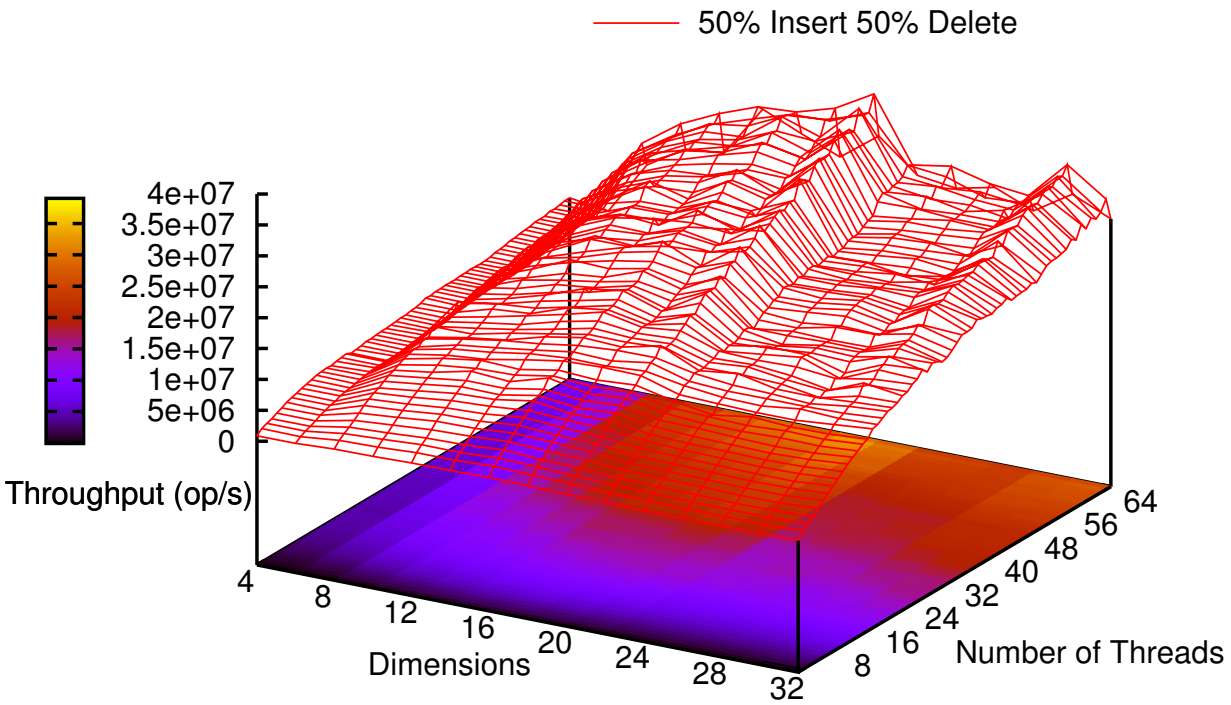
Throughput — SMP Mixed Workload



1M Keys, 30% Insert 20% Delete 50% Find

- 40% speedup for 12 threads
- Able to exploit hyper-threading

Dimension Sweep



1M Keys, 50% Insert 50% Delete

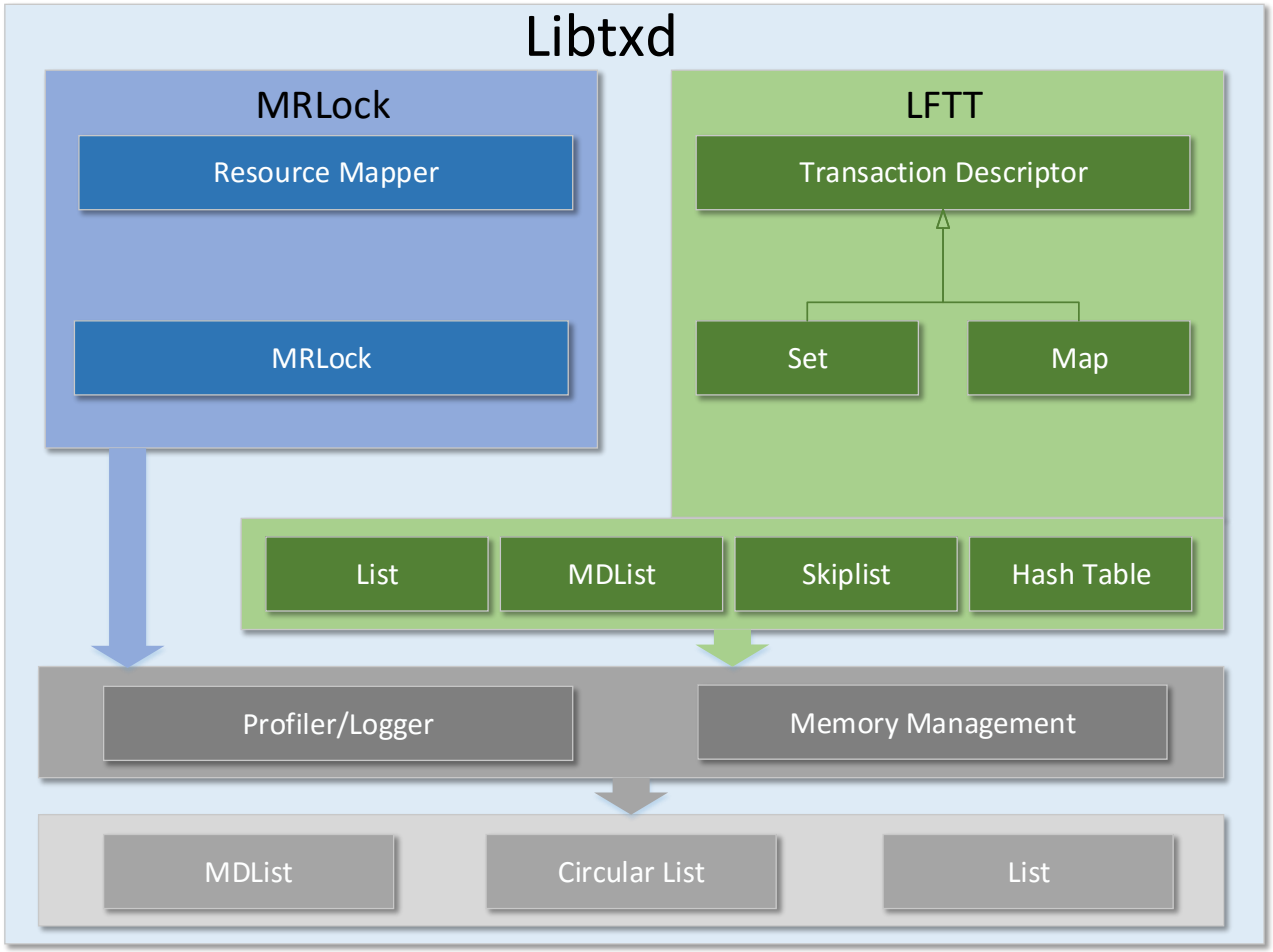
- For any number of threads, max throughput occurs when $D = 20$
- Performance of MDList is workload dependent and concurrency independent

Scalability Summary

- MRLock characteristics
  - Excels at medium to high levels of resources contention
  - Suitable for large pool of resources
- LFTT characteristics
  - Excels at large transactions
  - Improved success rate with minimal spurious aborts
- MDList characteristics
  - Excels at high level of concurrency and large key space
  - Optimized for write operations

Libtxd Architecture

- Unmanaged containers
- Utilities
- Managed containers
- LFTT component
- MRLock component



Libtxd Utilities

- Memory Management
  - Reference counting — Descriptors
  - Epoch based garbage collection — Nodes
- Lock-free Profiler
  - Existing loggers/profilers are blocking
  - Helps debugging and optimization

Future Development

- Support wait-free data structures
  - Wait-free update of TxInfo
  - Wait-free memory management
- Support non-linked data structures
  - Associate data item with TxInfo
- Automatic code transformation
  - Identify code blocks with user comments
  - Template-based source to source transformation

Thank You!

In Refereed Journals

- Deli Zhang and Damian Dechev. "A Lock-Free Priority Queue Design Based on Multi-Dimensional Linked Lists". In: *IEEE Transactions on Parallel and Distributed Systems* 27.3 (2016), pp. 613–626. ISSN: 1045-9219. DOI: 10.1109/TPDS.2015.2419651.
- Deli Zhang, Brendan Lynch, and Damian Dechev. "Queue-Based and Adaptive Lock Algorithms for Scalable Resource Allocation on Shared-Memory Multiprocessors". In: *International Journal of Parallel Programming (IJPP)* 43 (5 Aug. 2014), pp. 721–751. DOI: 10.1007/s10766-014-0317-6.

In Conference Proceedings

- Deli Zhang, and Damian Dechev. "An Efficient Lock-free Logarithmic Search Data Structure Based on Multi-dimensional List". In: *Proceedings of the 36th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, June 2016.
- Deli Zhang, and Damian Dechev. "Lock-free Transactions Without Rollbacks for Linked Data Structures". In: *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*. ACM, July 2016.
- Deli Zhang, Brendan Lynch, and Damian Dechev. "Fast and Scalable Queue-Based Resource Allocation Lock on Shared-Memory Multiprocessors". In: *Proceedings of the 17th International Conference on Principles of Distributed Systems*. Springer, Dec. 2013, pp. 266–280. DOI: 10.1007/978-3-319-03850-6_19.

Resources

- https://ucf-cs.github.io/tlds
- dezhan@microsoft.com, dechev@cs.ucf.edu