Algorithmic Improvements for Fast Concurrent Cuckoo Hashing

Presentation by Nir David

Xiaozhou Li , David G. Andersen , Michael Kaminsky , Michael J. Freedman

Overview

- Background and Related Work
  - Hash Tables
  - Concurrency Control Mechanisms
  - Naive use of concurrency control fails
- Principles to Improve Concurrency
- Concurrent Cuckoo Hashing
  - Cuckoo Hashing
  - Prior Work in Concurrent Cuckoo
  - **Algorithmic Optimizations**
  - Fine-grained Locking
- Optimizing for Intel TSX
- Evaluation

Concurrent hash table

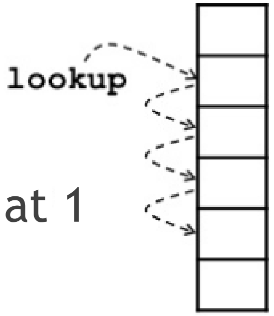
Provides: **Lookup**, **Insert** and **Delete** operations.

- On **Lookup**, a value is returned for the given key, or “does not exist” if the key cannot be found.
- On **Insert**, the hash table returns success, or an error code to indicate whether the hash table is full or the key is already exists.
- **Delete** simply removes the key’s entry from the hash table.

Several definitions before we go further

- **Open Addressing**: a method for handling collisions.

  A collision is resolved by **probing**, or searching through alternate locations in the array until either the target record is found, or an unused array slot is found.



- **Linear probing -** in which the interval between probes is fixed — often at 1

- **Quadratic probing -** in which the interval between probes increases linearly

$$H+1^2, H+2^2, H+3^2, H+4^2, \ldots, H+k^2$$

Linear Probing: $f(i) = i$

```
Insert(k,x)  // assume unique keys
1. index =  hash(key) % table_size;
2.  if (table[index]== NULL)
        table[index]=
        new key_value_pair(key, x);
3. Else  {
   • index++;
   • index = index % table_size;
   • goto 2;
   }
```

Linear Probing Example

89 mod 10 = 9

18 mod 10 = 8

49 mod 10 = 9

58 mod 10 = 8

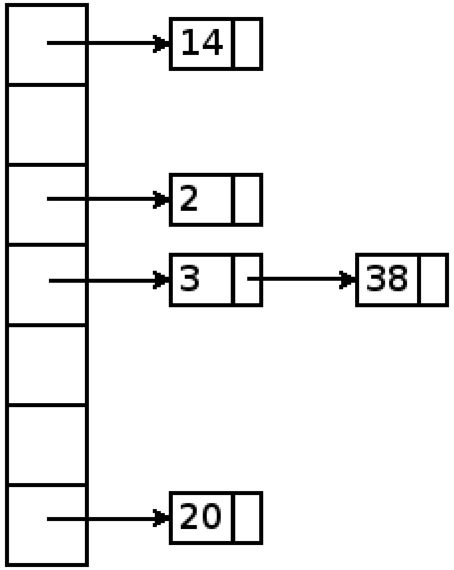
- Insert 89, 18, 49, 58, 69

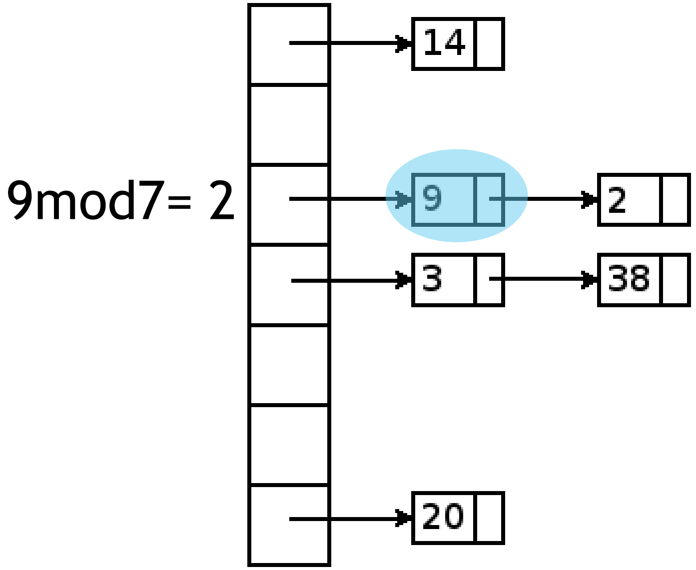
	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Several definitions before we go further

- **Chaining:** another possible way to resolve collisions.

  Each slot of the array contains a link to a singly-linked list containing key-value pairs with the same hash.





High-performance single-thread hash tables

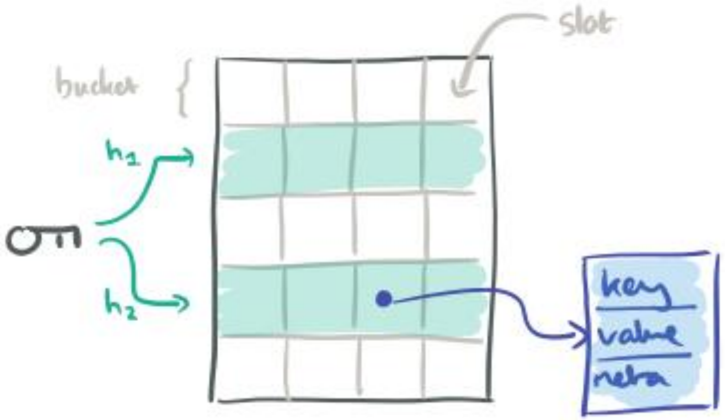
- Google's **dense_hash_map**
  - It uses open addressing with quadratic probing
- C++11 introduces an **unordered_map**
  - implemented as a separate chaining hash table
- The performance of these hash tables does not scale with the number of cores in the machine, because only one writer or one reader is allowed at the same time.

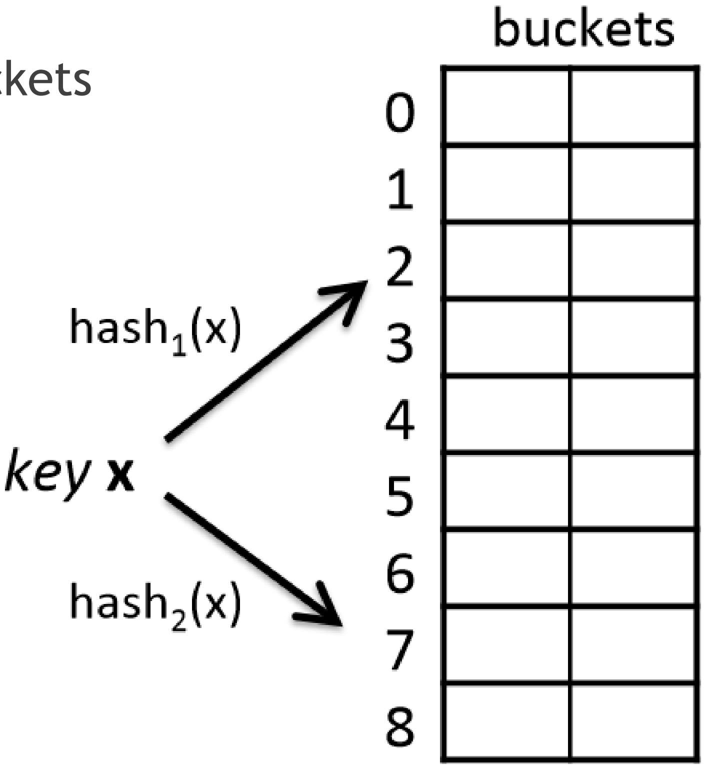
Multiple-reader, single-writer hash tables

- A middle ground between no thread safety and full concurrency.
- Single-writer tables can be extended to permit many concurrent readers.
- **Cuckoo hashing** is an open-addressed hashing technique with high memory efficiency and O(1) amortized insertion time and retrieval.
- The paper's work builds upon one such hash table design.

Cuckoo hashing

- Each bucket has *b* slots for items (b-way set-associative)
- Each key is mapped to two random buckets
  - Stored in one of them

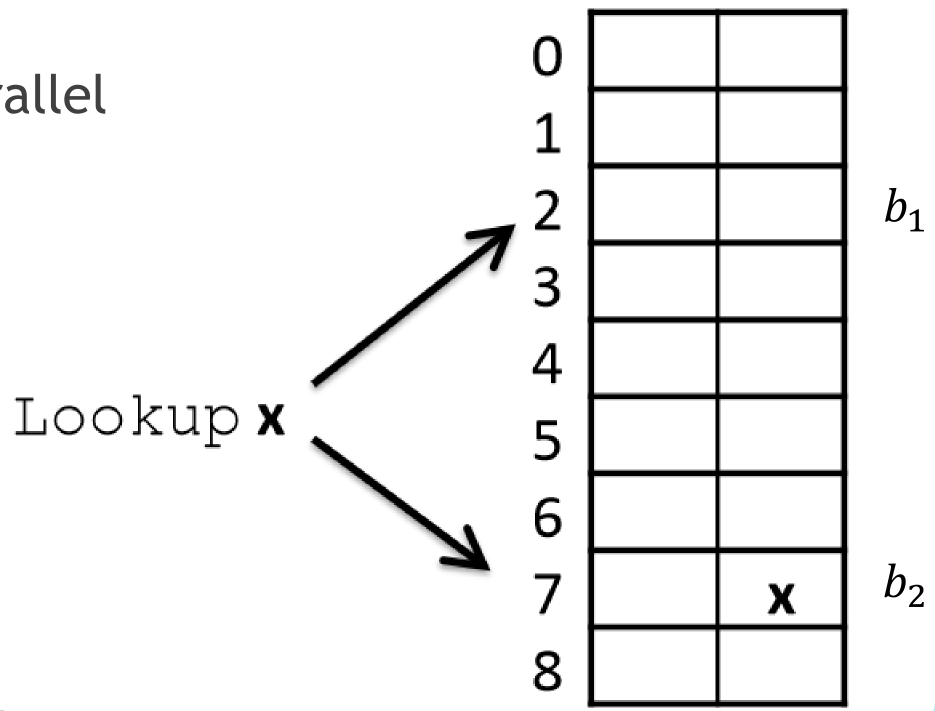




Cuckoo hashing

- **Lookup:** read 2 buckets b_1, b_2 in parallel
  - constant time in the worst case

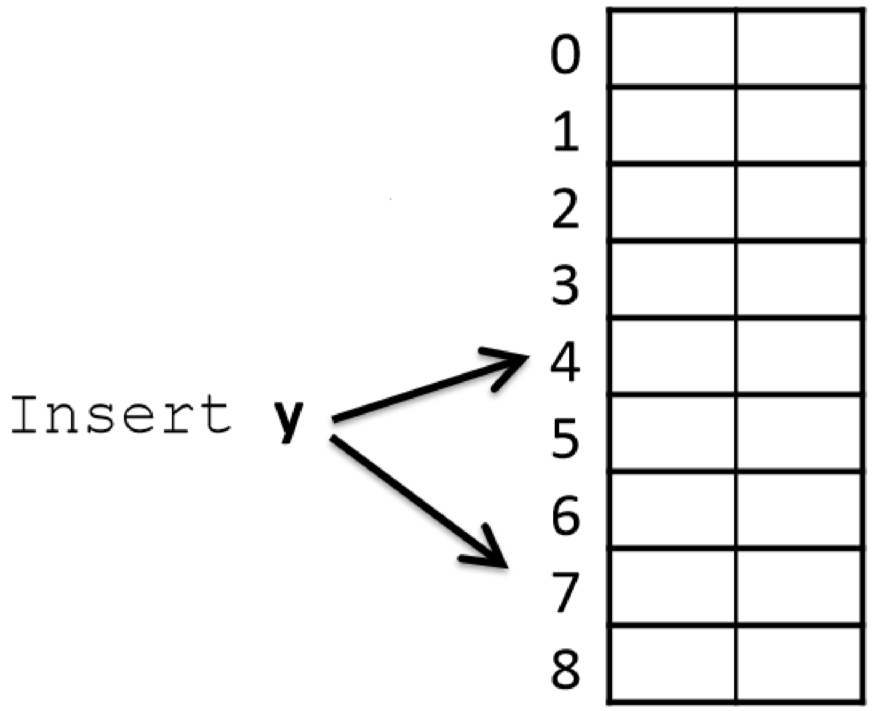
1. Compute 2 hashes of key to find b_1, b_2 that could be used to store the key
2. Examine all of the slots within each of those buckets to determine if the key is present



Always checking up to *2b* keys.

Cuckoo hashing

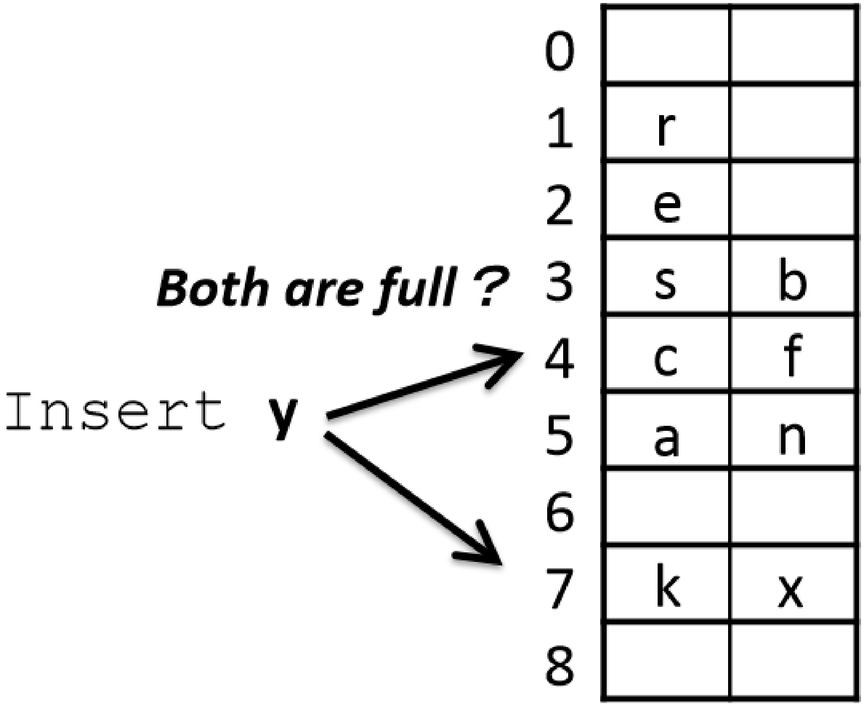
- **Insert may need ”cuckoo move”**
- Write to an empty slot in one of the two buckets



Cuckoo hashing

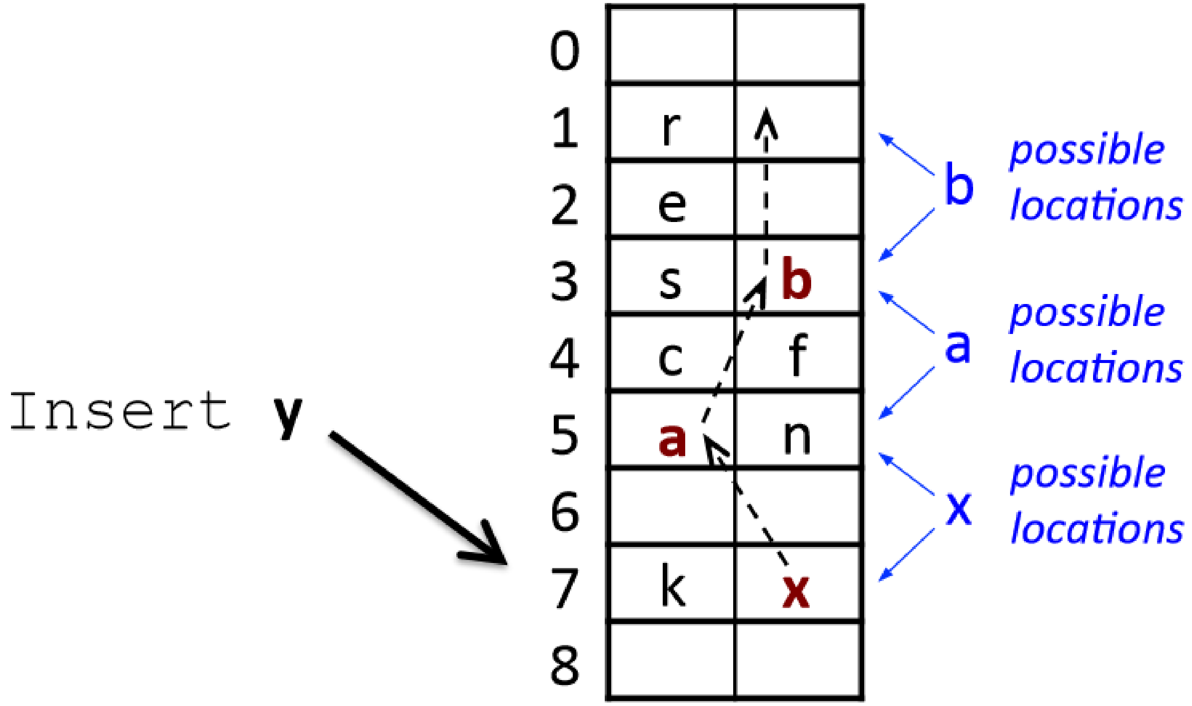
Insert:

- when both cells are already full, it will be necessary to move other keys to their second locations (or back to their first locations) to make room for the new key.



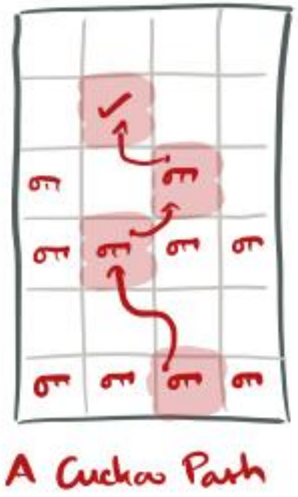
Insert may need "cuckoo move"

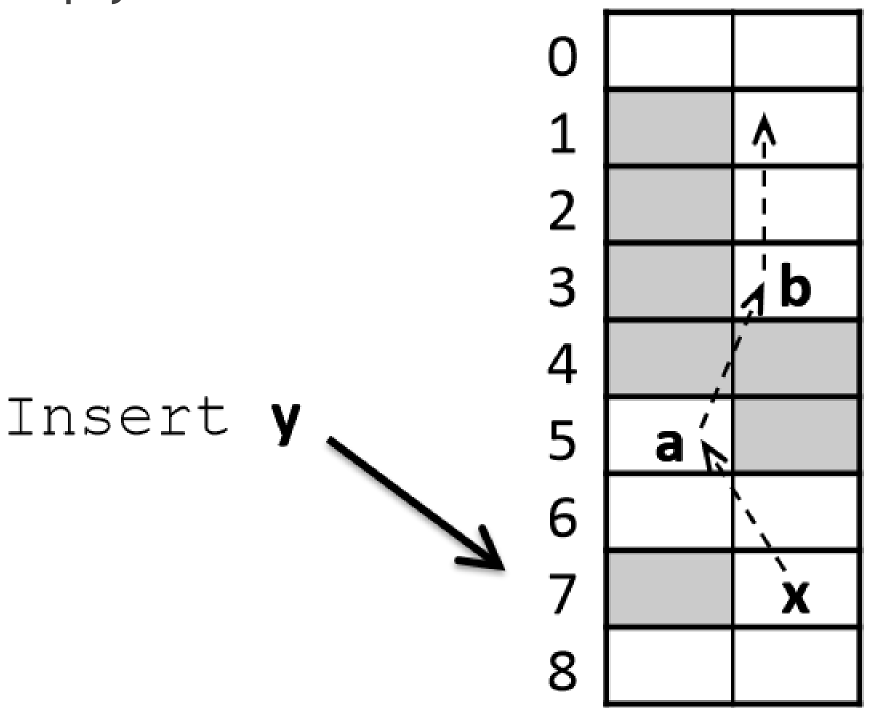
- **Insert:** move keys to alternate buckets



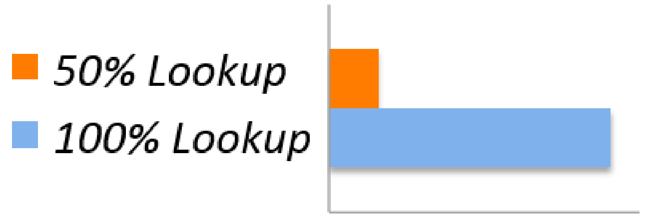
Insert may need ”cuckoo move”

- **Insert:** move keys to alternate buckets
  - find a **“cuckoo path”** to an empty slot
  - move hole backwards





Review



- **Benefits**
  - support concurrent reads
  - memory efficient for small objects

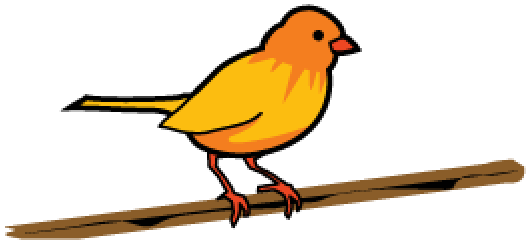
    over 90% space utilized when set-associativity ≥ 4

- **Limits**
  - Inserts are serialized

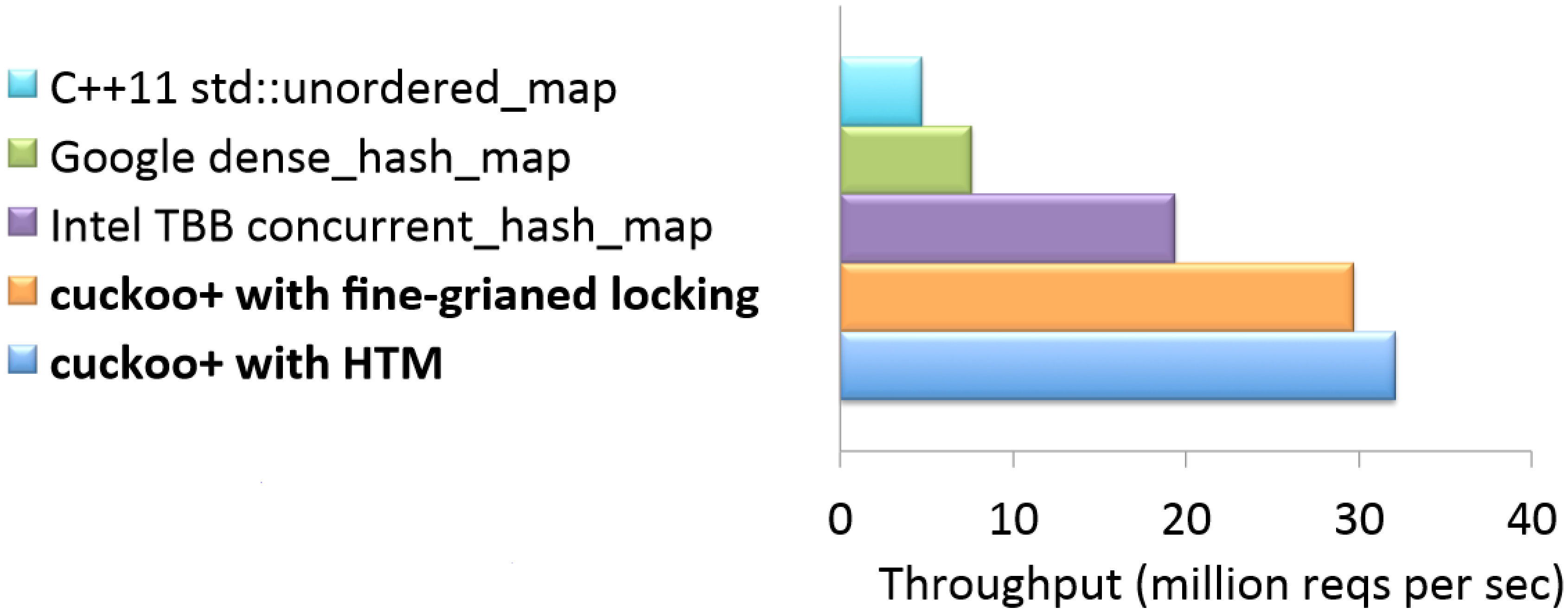
    poor performance for write-heavy workloads

Scalable concurrent hash tables

- The Intel TBB library provides a **concurrent_hash_map** that allows multiple threads to concurrently access and update values.
- This hash table is also based upon the classic separate chaining design.



Preview of the results on a quad-core machine



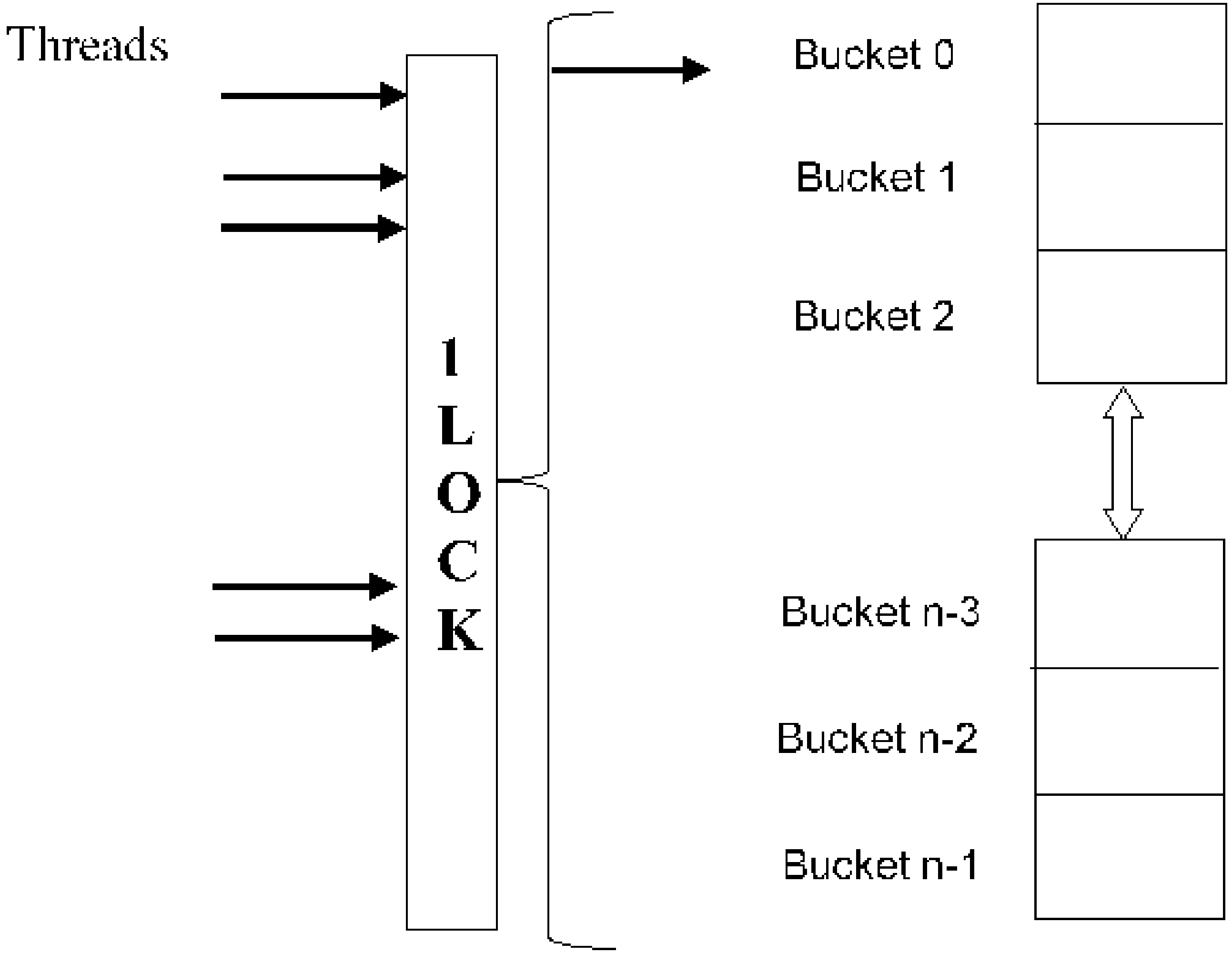
Concurrency Control Mechanisms

- **Locking:**
  - Coarse-grained locking
  - Fine-grained locking
- **Hardware Transactional Memory (HTM)**
  - Intel Transactional Synchronization Extensions (TSX)
  - Hardware support for lock elision

Coarse-grained locking

- The simplest form of locking is to wrap a single lock around **the whole** shared data structure
- Only one thread can hold the lock at the same time
- Hard to mess up
- This tends to be pessimistic, why?

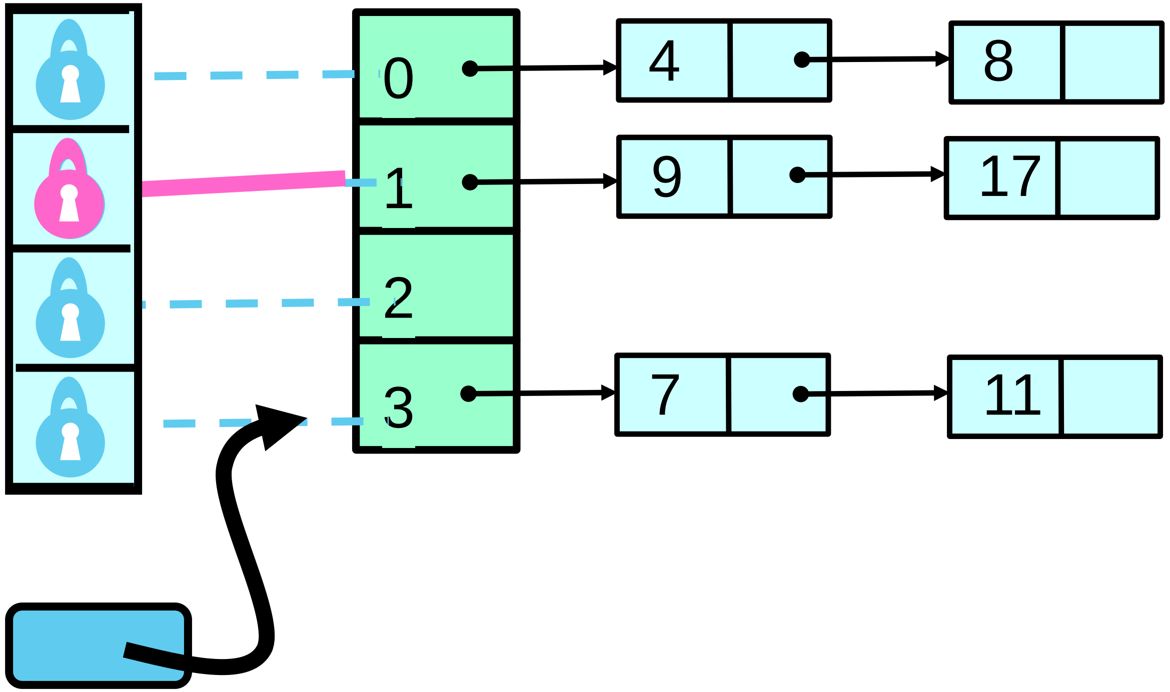
Coarse-grained locking



Fine-grained locking

- Splitting the coarse-grained lock into multiple locks
- Each lock is responsible for protecting a region of the data
- Multiple threads can operate on different regions of the data at the same time
- It can improve the overall performance of a concurrent system
- However, it must be carefully designed and implemented to behave correctly without deadlock, livelock, starvation, etc.

Fine-grained locking



Each lock associated with one bucket

Hardware Transactional Memory (HTM)

- All shared memory accesses and their effects are applied **atomically**
- Threads no longer need to take locks when accessing the shared data structures
- Yet the system will still guarantee thread safety

Intel Transactional Synchronization Extensions (TSX)

- An extension to the Intel 64 architecture that adds transactional memory support in hardware
- Allows the processor to determine dynamically whether threads need to serialize through lock-protected critical sections
- Serialize only when required
- The program can declare a region of code as a transaction

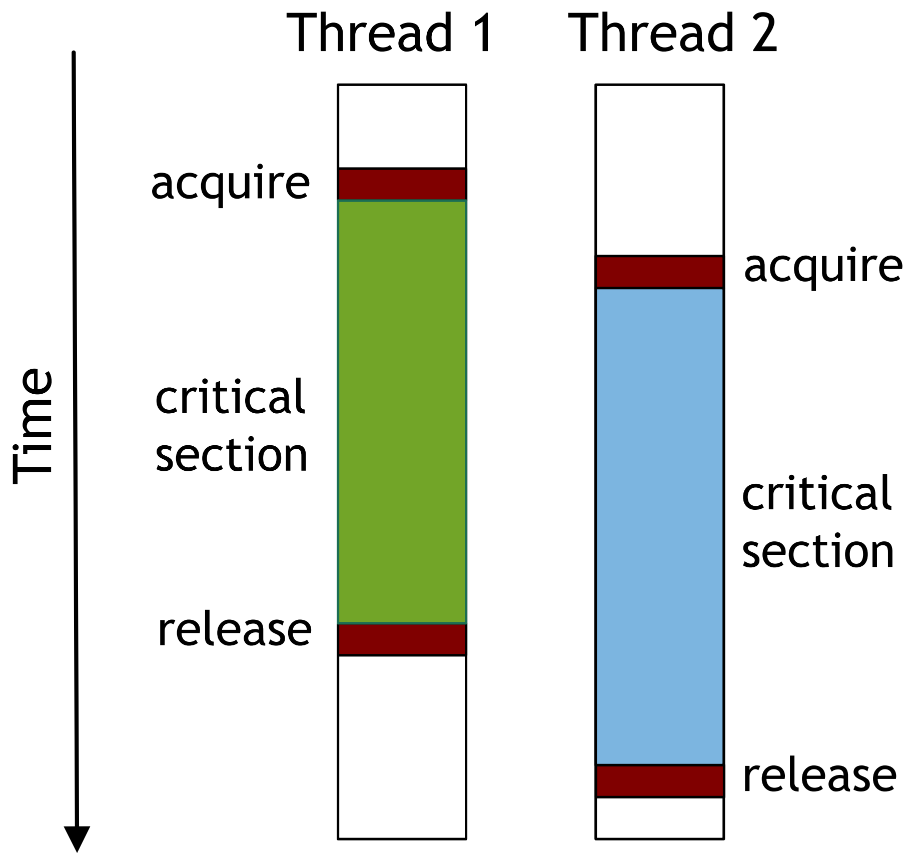
Naive use of concurrency control fails

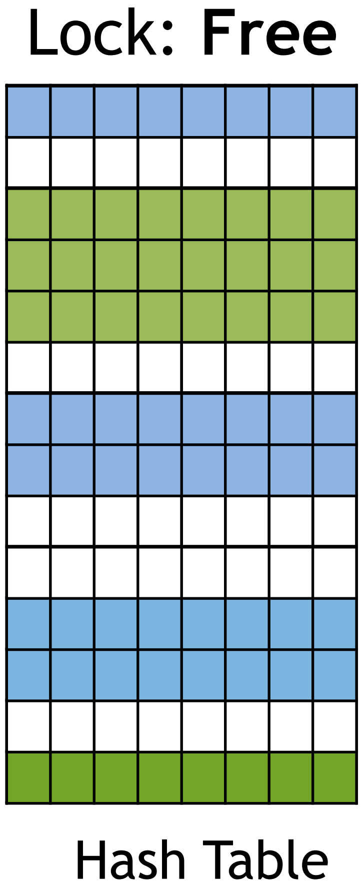
- The hardware provides no guarantees as to whether a transactional region will ever successfully commit.
- Therefore, any transaction implemented with TSX needs a **fallback path**.

The simplest fallback mechanism is “**lock elision**”:

- The program executes a lock-protected region speculatively as a transaction.
- Only falls back to use normal locking if the transaction does not succeed.

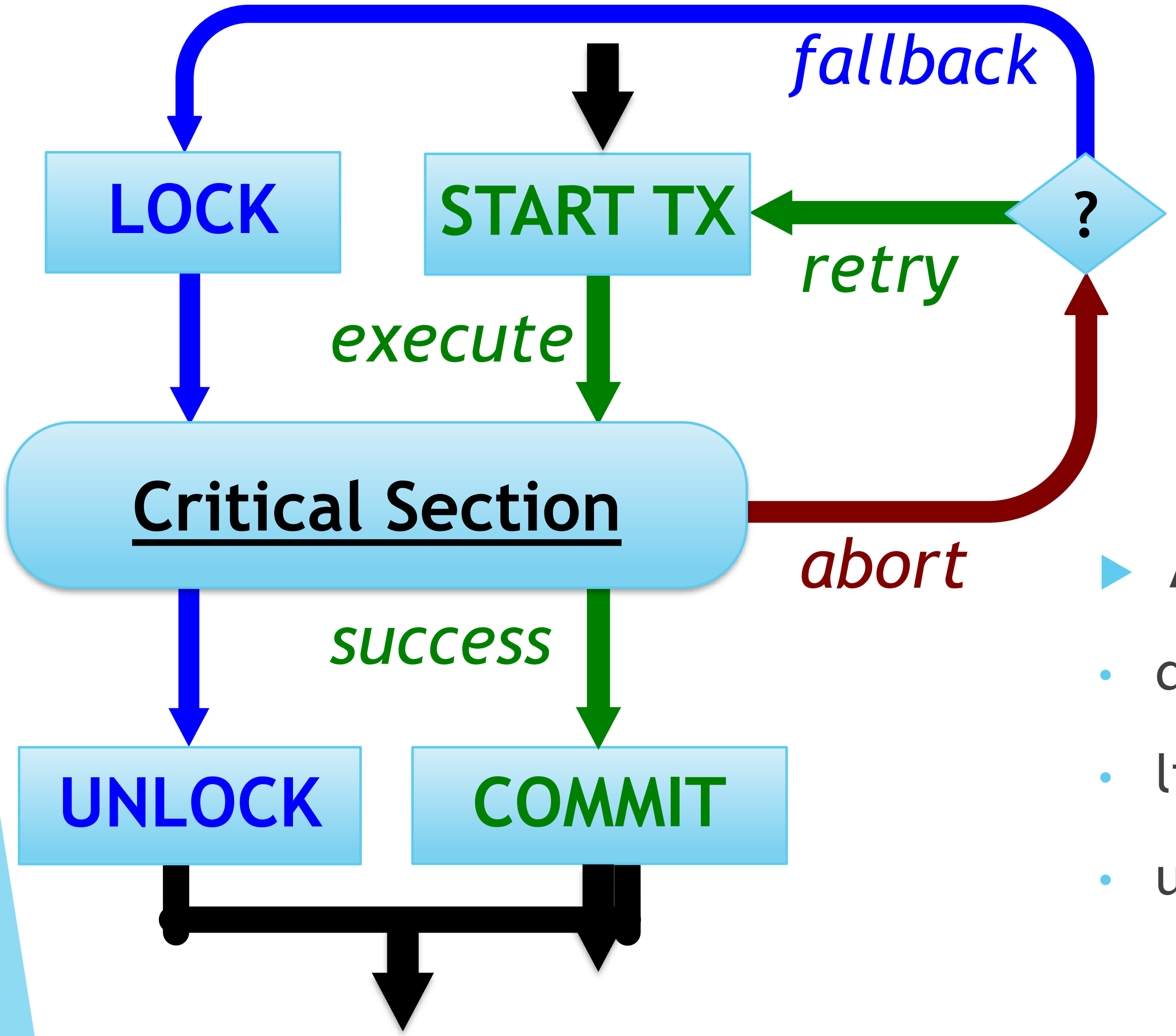
Lock elision





No serialization if no data conflicts

Implement lock elision with Intel TSX



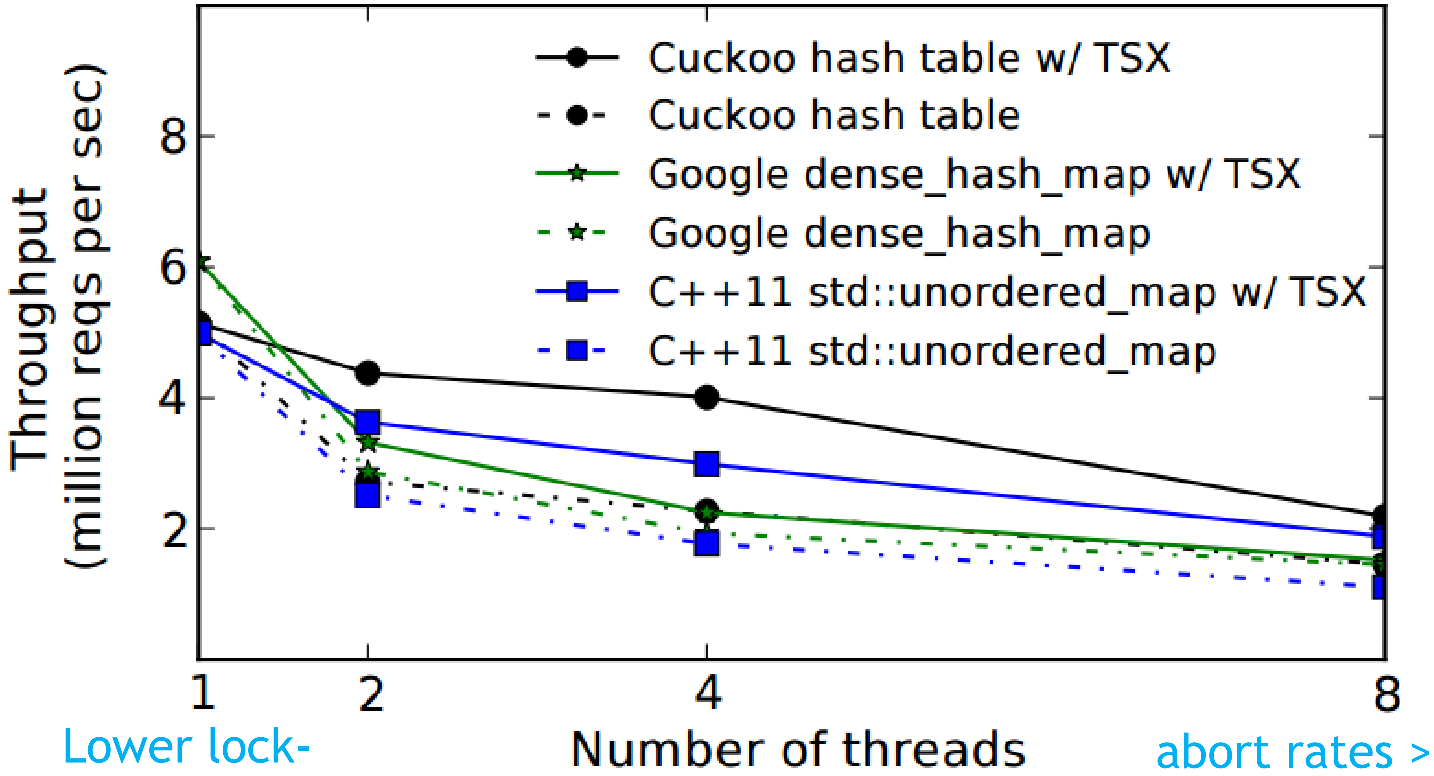
optimized to make better decisions

- **Abort reasons:**
  - data conflicts
  - limited HW resources
  - unfriendly instructions

Lock elision

- Advantages:
  - Simple: No code modifications
- Disadvantages:
  - Serial fallback: **All** concurrent hardware transactions must abort

Insert throughput vs. number of threads



Lower lock-contention

abort rates > 80%

Extensive lock-contention

Principles to Improve Concurrency

- Given that naive global locking with or without HTM support fails to provide scalable performance, what must be done? How to reduce lock contention?

1. Avoid unnecessary or unintentional access to common
   - Make globals thread-local
2. Minimize the size and execution time of critical sections
   - Lock later, Perform searches outside of a critical section
3. Optimize the concurrency control mechanism
   - e.g. Use spinlocks and lock striping for short critical sections

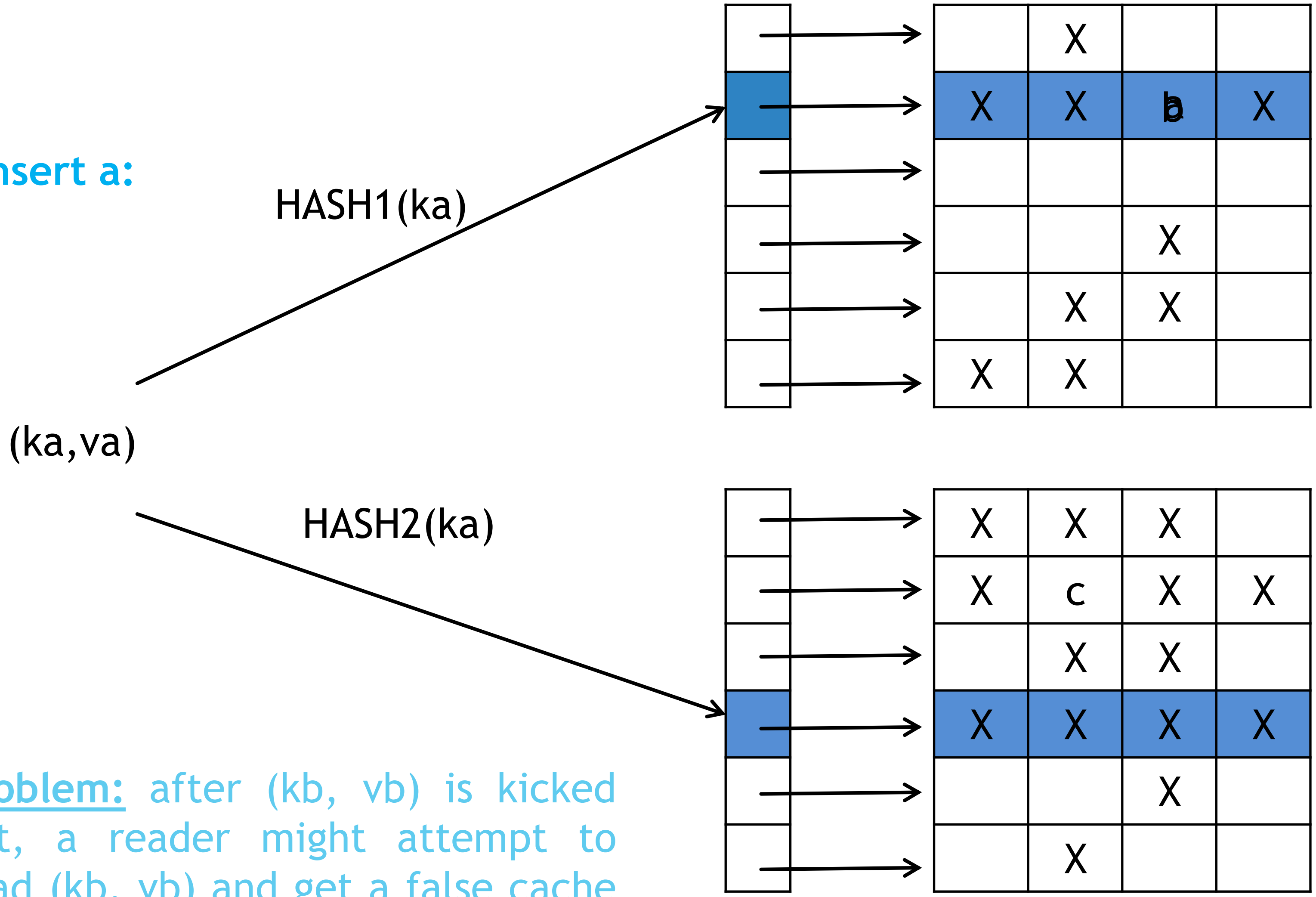
Cuckoo Hashing

- We have already explained it (slides 7-13).

  Let’s do a really short review ☺

Cuckoo Hashing

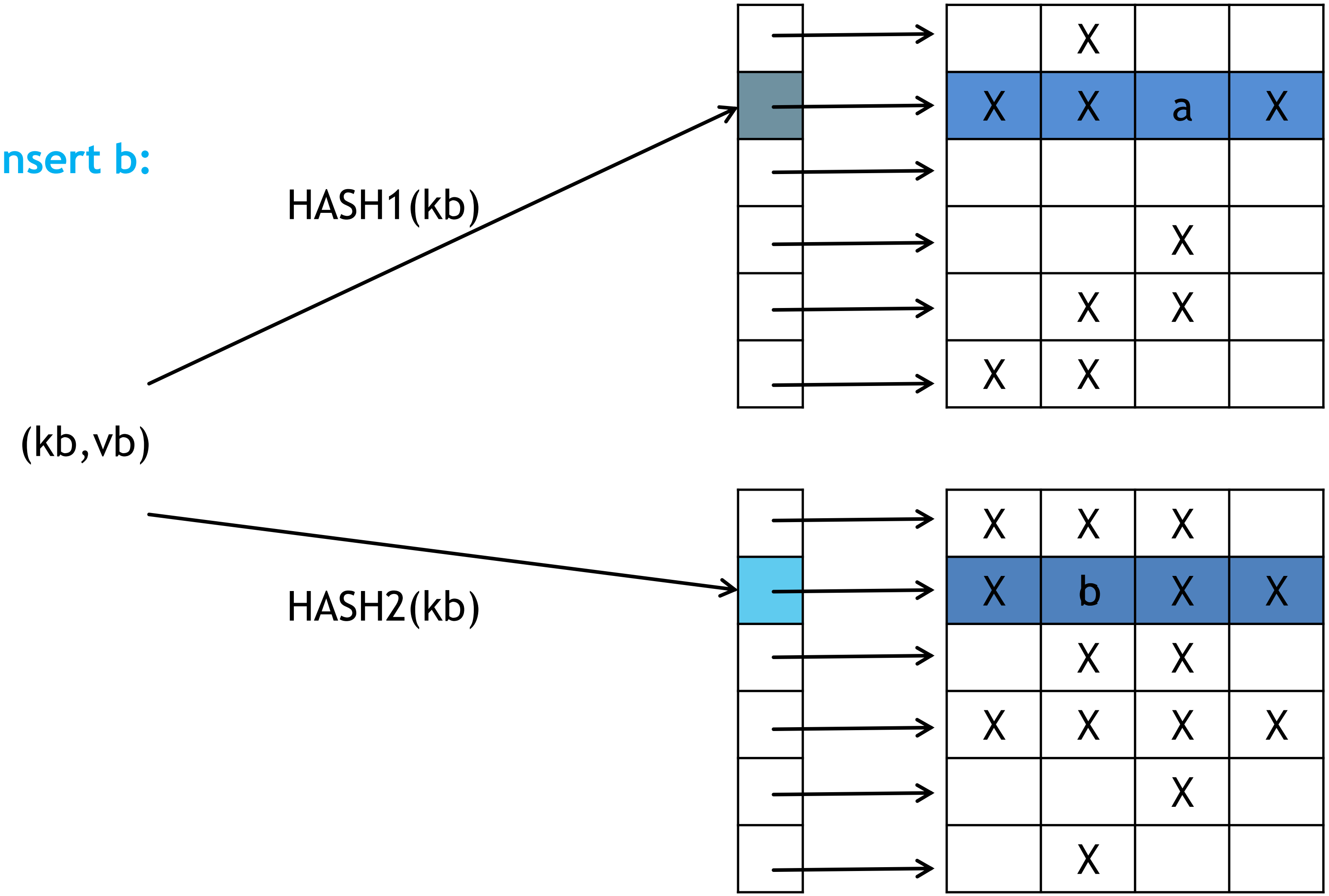
Insert a:



Problem: after (kb, vb) is kicked out, a reader might attempt to read (kb, vb) and get a false cache miss

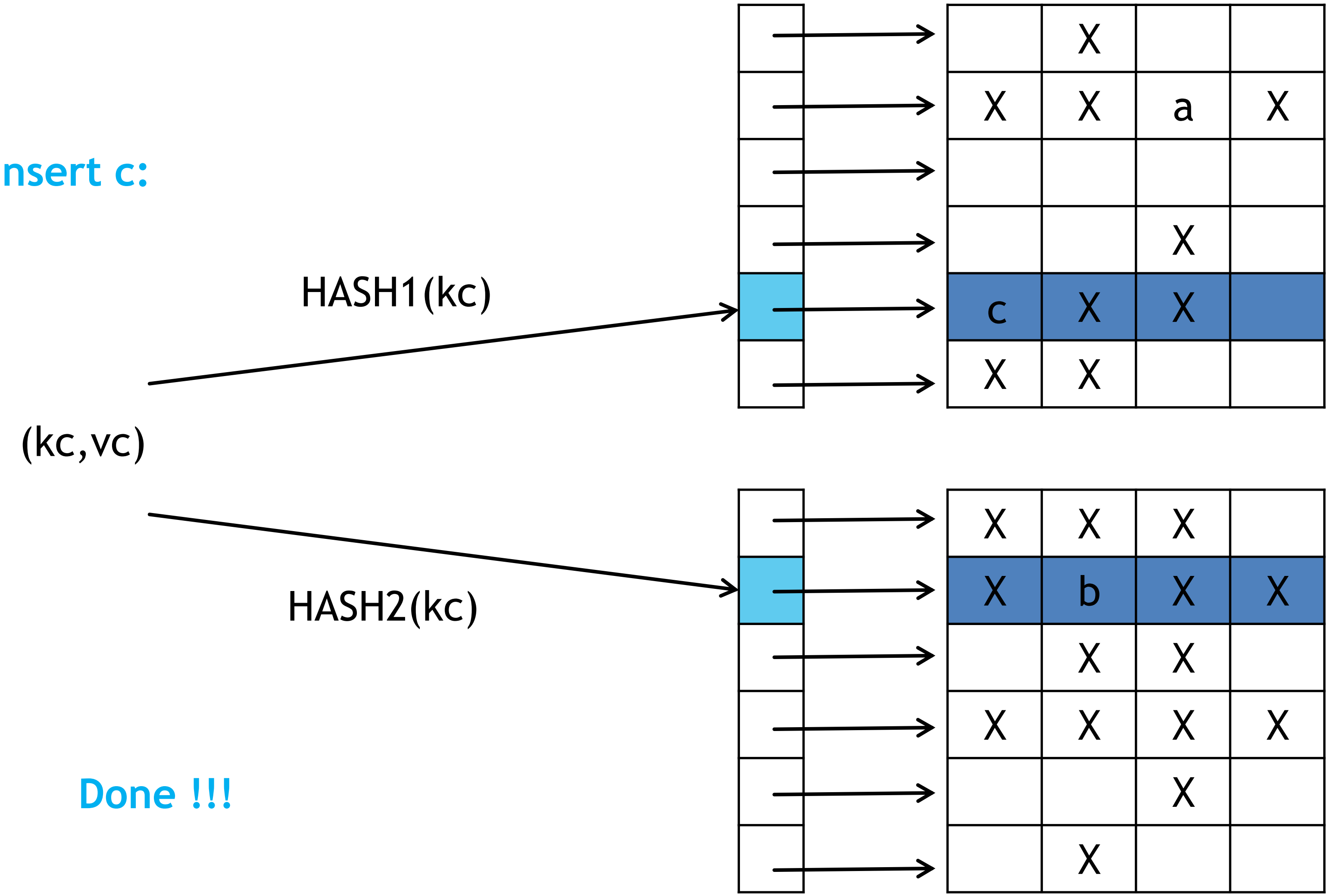
Cuckoo Hashing

Insert b:



Cuckoo Hashing

Insert c:



Done !!!

Prior Work in Concurrent Cuckoo

Starting point: **MemC3's table**

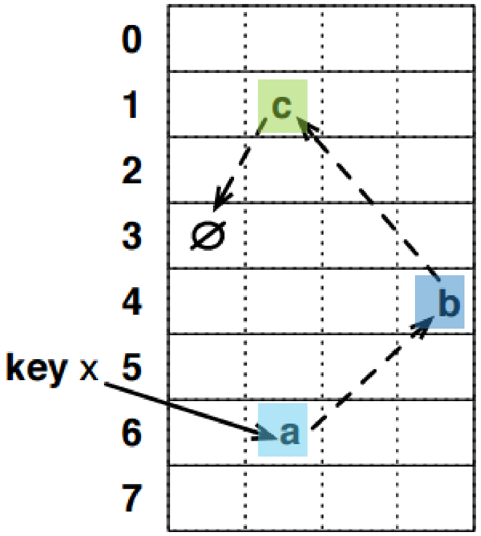
To eliminate **false misses**, change the order of the basic cuckoo hashing insertions:

- Allow concurrent reads and cuckoo movement by moving "holes" backwards along the cuckoo path instead of moving "items" forward along the cuckoo path
- This ensures that an item can always be found by a reader thread!
- If it's undergoing concurrent cuckoo movement, it may be present **twice** in the table, but will never be missing
- Let's see an example

Prior Work in Concurrent Cuckoo

It requires separating the process of searching for a cuckoo path from using it:

- First find a valid cuckoo path “a→b→c→∅” for key x without editing any buckets
- After the path is known, c is swapped to the empty slot in bucket 3, followed by relocating b to the original slot of c in bucket 1 and so on. Finally, the original slot of a will be available and x can be directly inserted into that slot.

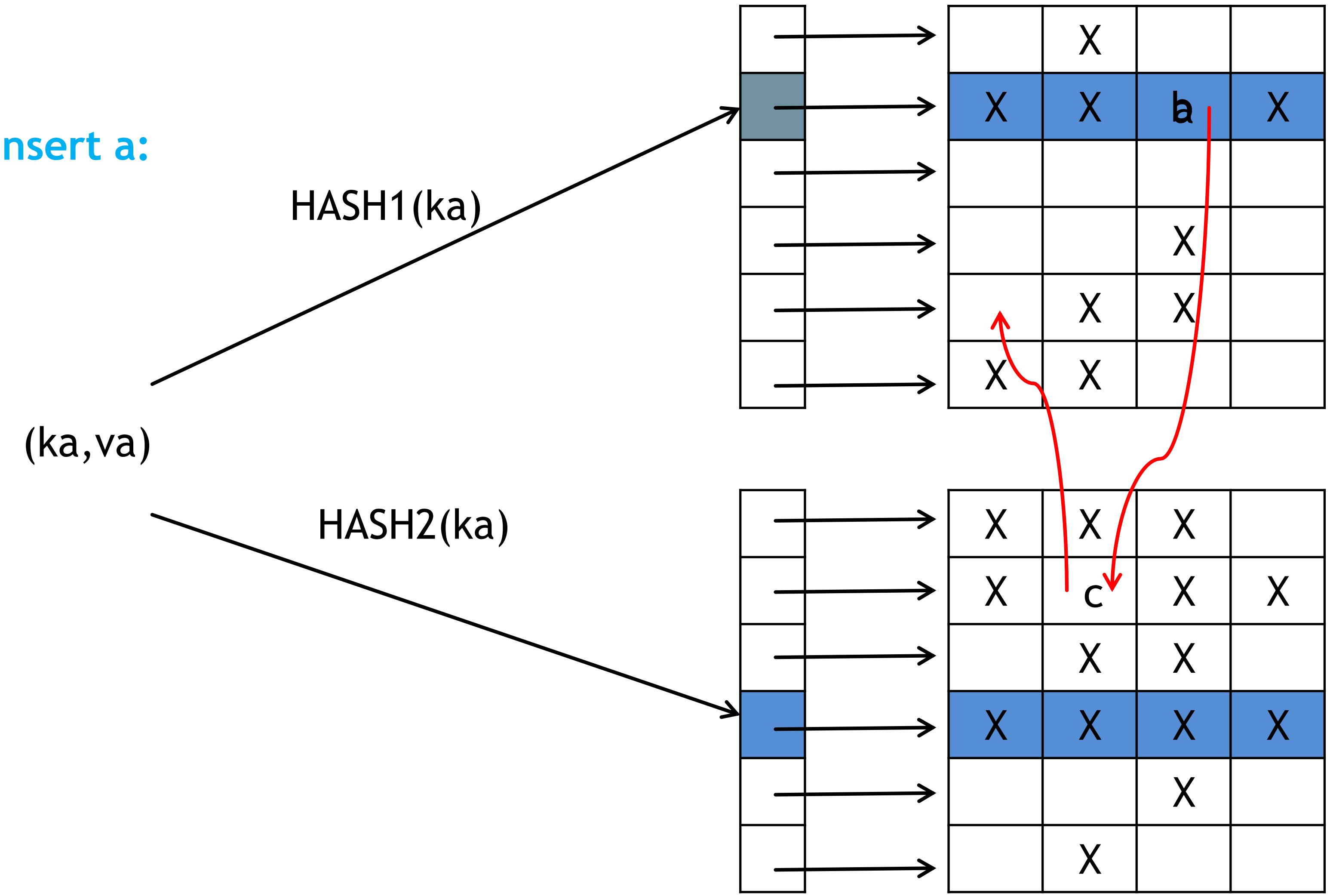


Cuckoo path

No kick-outs

Cuckoo path backward insert

Insert a:



Why a key will never be missing?

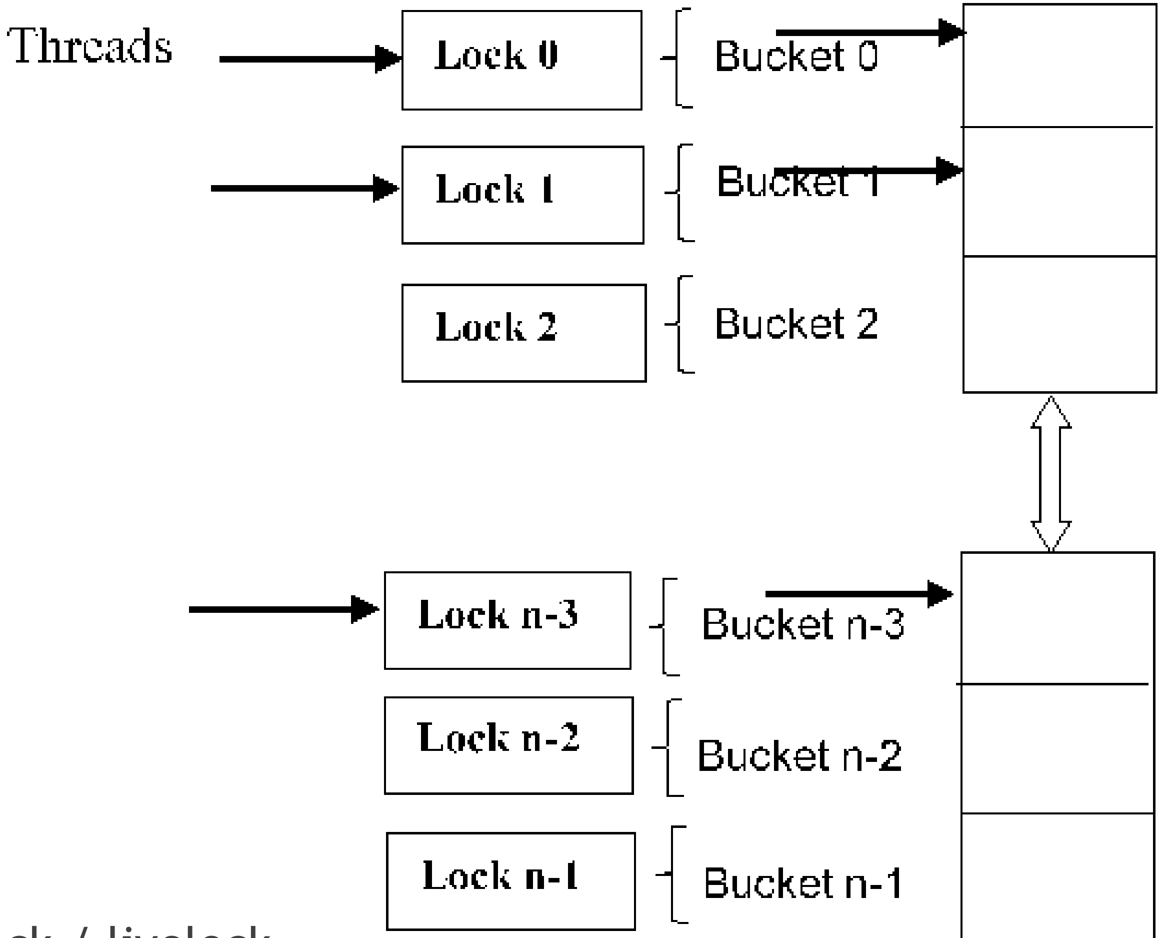
- Before: (b,c,Null)->(a,c,Null)->(a,b,Null)->(a,b,c)
- Fixed: (b,c,Null)->(b,c,c)->(b,b,c)->(a,b,c)

- What if reader looks for c, see null in bucket 3, then go to sleep.
- Then c is copied to bucket 3, and also b is copied (overriding c in bucket 2).
- Reader wakes up, and search c in bucket 2. Then returns mistakenly false.

- In next slides I will explain why this scenario can not happen.

Prior Work in Concurrent Cuckoo

Starting point: **MemC3's table used Lock Striping**



- **Risks:** deadlock / livelock

Prior Work in Concurrent Cuckoo

Starting point: **MemC3's table - Optimistic Locks**

- The most straightforward scheme is to lock two relevant buckets before each displacement and each Lookup

```
//! find searches through the table for \p key, and stores the associated
//! value it finds in \p val. Must be copy assignable.
template <typename K>
bool find(const K& key, mapped_type& val) const {
    const size_t hv = hashed_key(key);
    const auto b = snapshot_and_lock_two(hv);
    const cuckoo_status st = cuckoo_find(key, val, hv, b.i[0], b.i[1]);
    return (st == ok);
}
```

Prior Work in Concurrent Cuckoo
Starting point: **MemC3's table - Optimistic Locks**

Using optimistic locks for Lookup

- The most straightforward scheme is to lock two relevant buckets before each displacement and each Lookup

- Though simple, this scheme requires locking twice for every Lookup. *Deadlock?*

- Instead of locking for reads, the hash table uses a lock-striped version counter associated with the buckets, updates it upon insertion or displacement, and looks for a version change during lookup

Prior Work in Concurrent Cuckoo

Starting point: **MemC3's table - Optimistic Locks**

- Before displacement, $counter_{k_i}$++, indicating to the other Lookups an on-going update for k_i
- After moved to new location, $counter_{k_i}$++ again to indicate completion
- So $counter_{k_i}$ is increased by 2 after each displacement
- Before the Lookup process starts, it checks $counter_{k_i}$
- If it's odd, there must be a concurrent writer working on the same key, and the reader should **wait and retry**
- Otherwise Lookup proceeds
- After it finishes, it checks $counter_{k_i}$ again and compares this new version with the old version. If differ, it indicates that the writer has modified this key, and the reader should **retry**
- <u>Exercise</u>: prove correctness ;) Email Adam the proof to get a bonus

Concurrent Cuckoo Hashing – Algorithmic Optimizations

- Lock after discovering a cuckoo path
  - Minimize critical sections
- Breadth-first search for an empty slot
  - fewer items displaced
- Increase set-associativity
  - fewer random memory reads

Previous approach: writer locks the table during the whole insert process

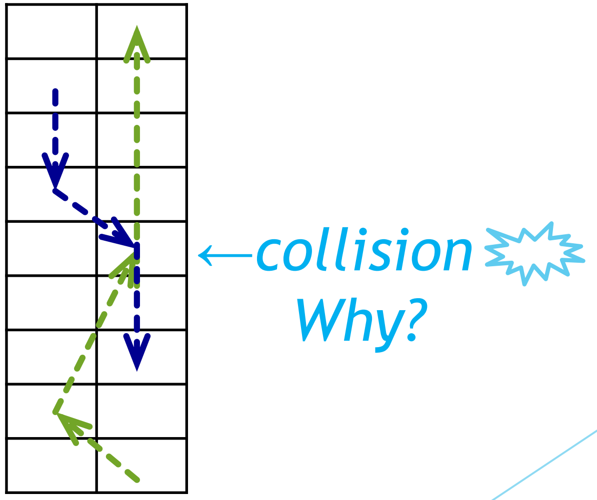
All Insert operations of other threads are blocked

- **Lock();**
- Search for a cuckoo path; //at most hundreds of bucket reads
- Cuckoo move and insert; //at most hundreds of writes
- **Unlock();**

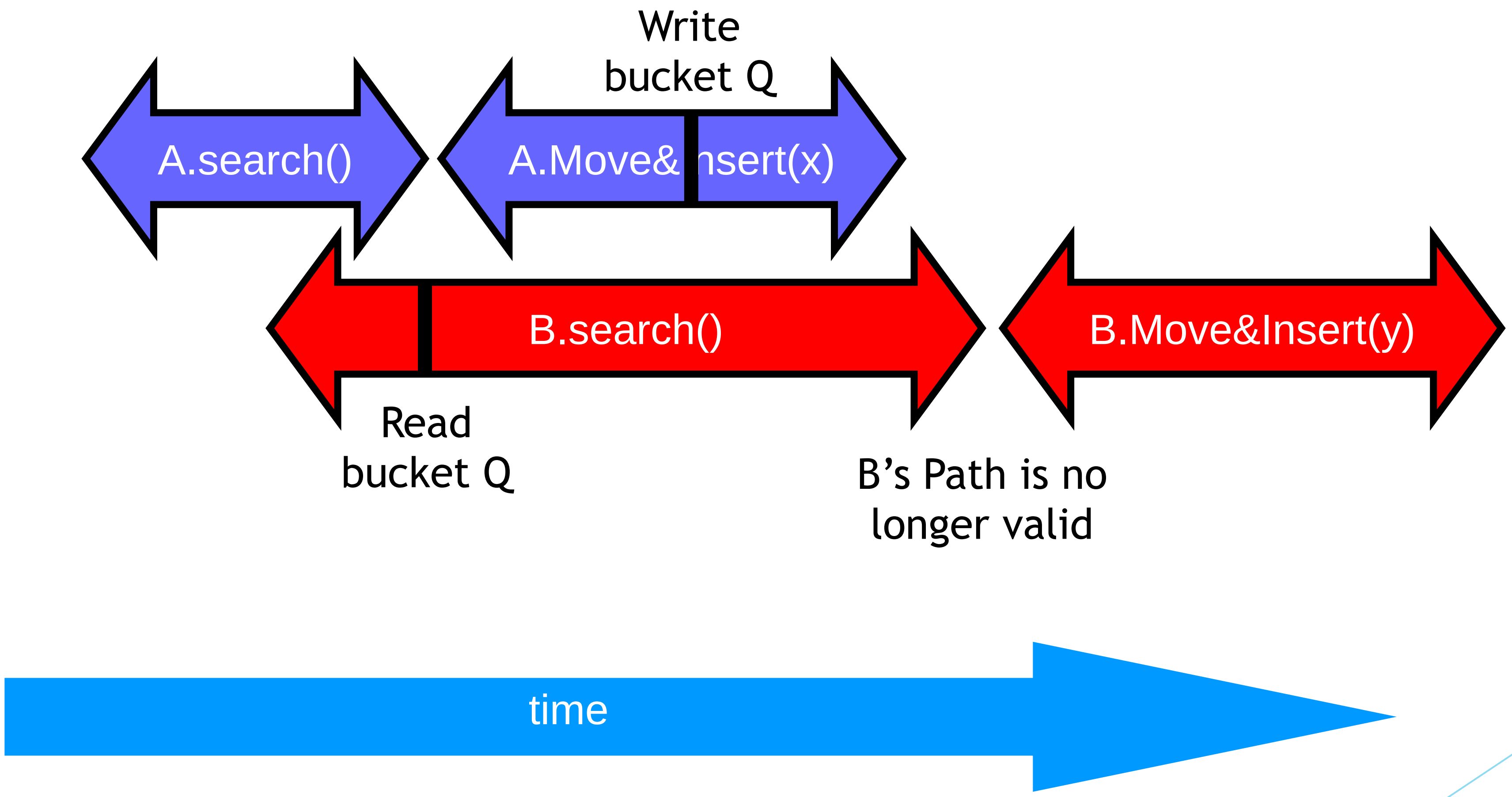
Lock after discovering a cuckoo path

Multiple Insert threads can look for cuckoo paths concurrently

- Search for a cuckoo path; //no locking required
- **Lock();**
- Cuckoo move and insert;
- **Unlock();**



Lock after discovering a cuckoo path



Lock after discovering a cuckoo path

Multiple Insert threads can look for cuckoo paths concurrently

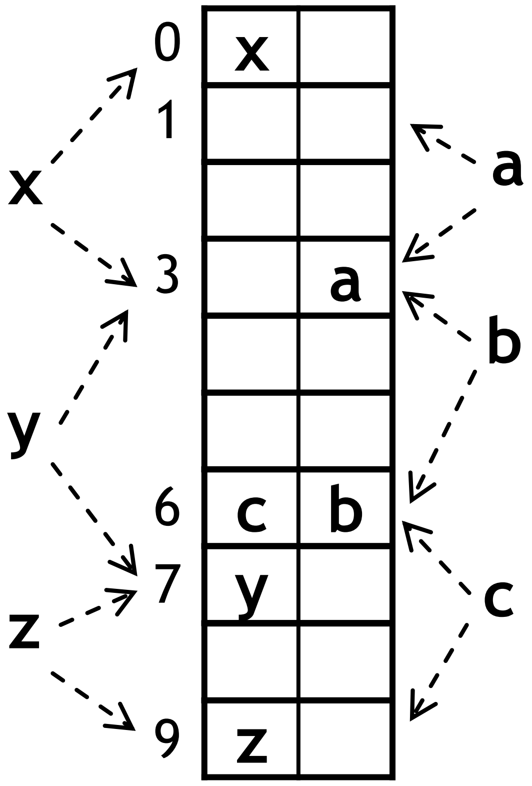
```
While(1) {
    Search for a cuckoo path;     //no locking required
    Lock();
    Cuckoo move and insert while the path Is valid
    If (success) {
        Unlock();
        Break; }
    Unlock();
}
```

How often does a path become invalid after being discovered?

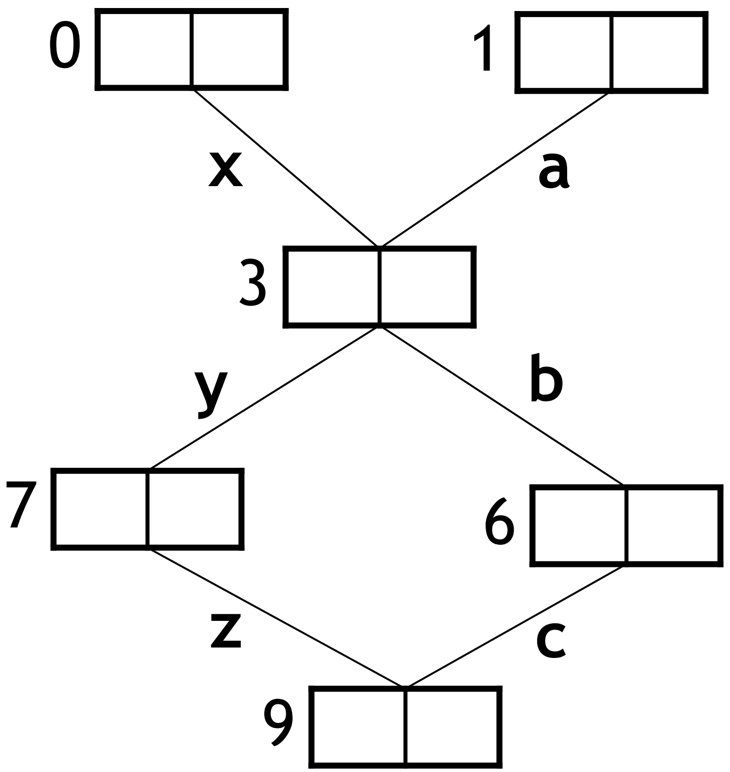
- We can estimate the probability that a cuckoo path of one writer overlaps with paths of other writers
- **N** num of entries in the hash table
- **L(<<N)** max length of a cuckoo path
- **T** num of threads
- $P_{invalid\_max} \approx 1 - \left(\frac{(N-L)}{N}\right)^{L(T-1)}$
- For N = 10M, L = 250, T = 8, we get: $\boldsymbol{P_{invalid} < 4.281\%}$

Cuckoo hash table > cuckoo graph

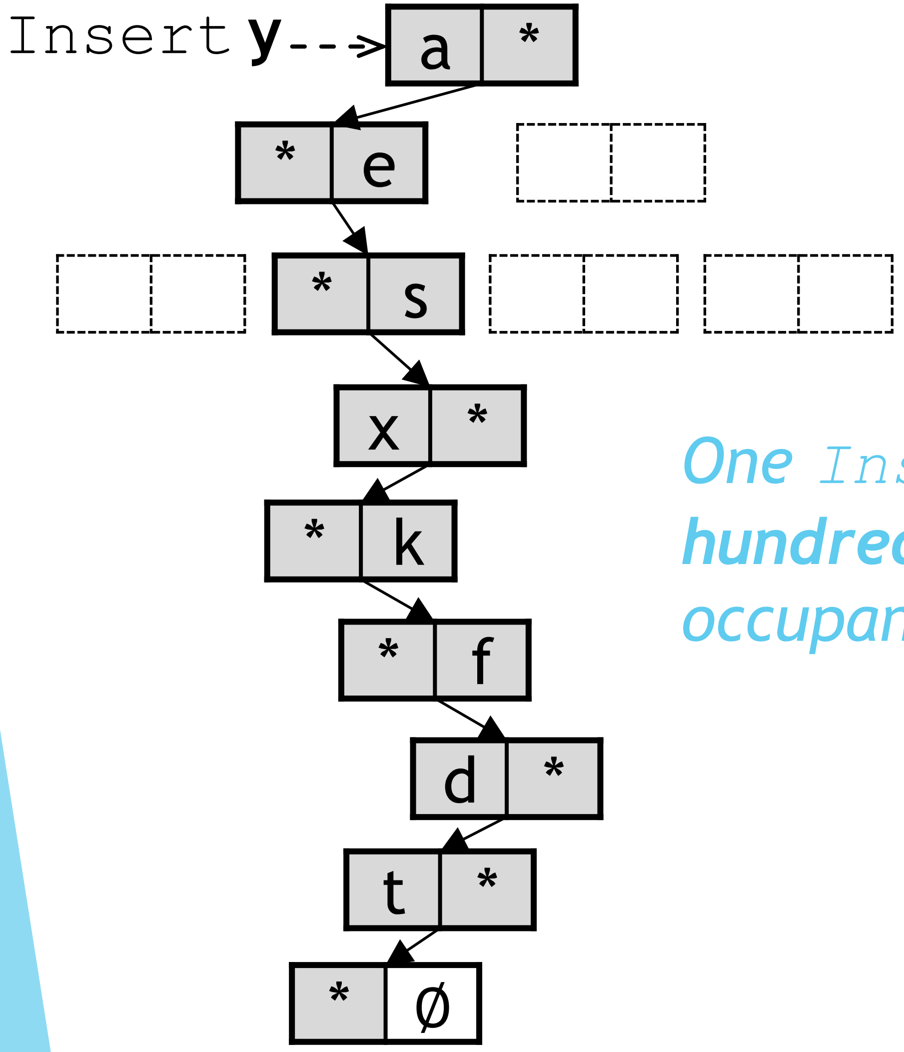
bucket ⟶ vertex
key ⟶ edge



⟹



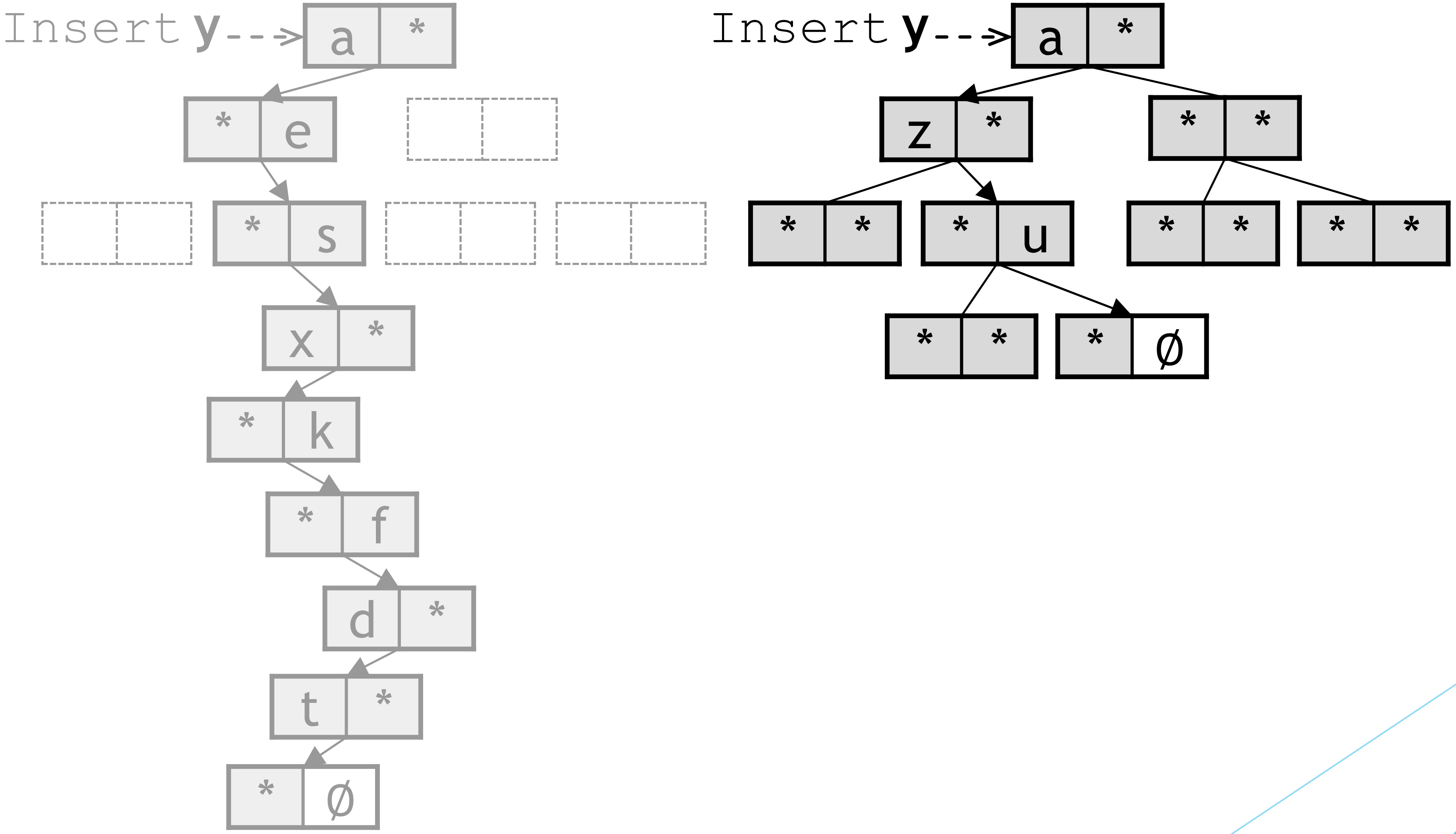
Previous approach to search for an empty slot: **random DFS** on the cuckoo graph



cuckoo path:
a→e→s→x→k→f→d→t→Ø
9 writes

*One `Insert` may move at most **hundreds** of items when table occupancy > 90%*

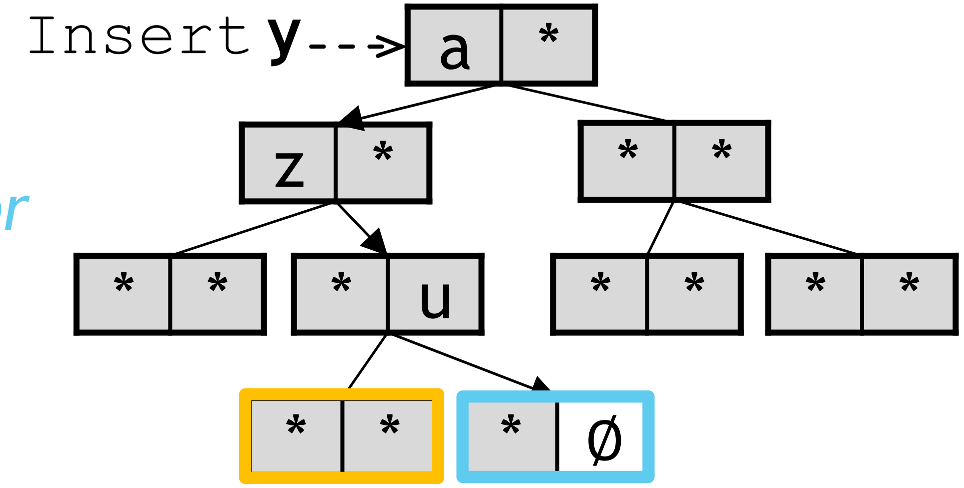
Breadth-first Search for an Empty Slot



Breadth-first Search for an Empty Slot

cuckoo path:
a→z→u→∅ **4 writes**
Reduced to a logarithmic factor

- Same # of reads → unlocked
- Far fewer writes → **locked**



Prefetching: scan one bucket and load next bucket concurrently

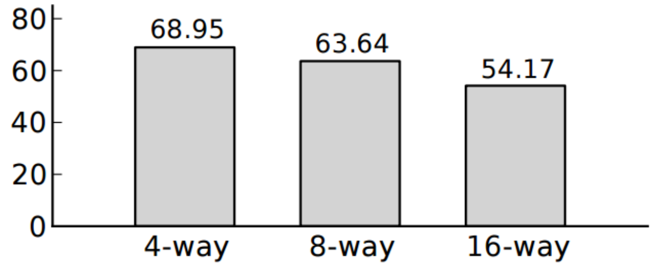
Further Explanation for BFS

- The maximum lengths of cuckoo paths from BFS is:

$$L_{BFS} = \left\lceil log_B\left(\frac{M}{2} - \frac{M}{2B} + 1\right)\right\rceil$$

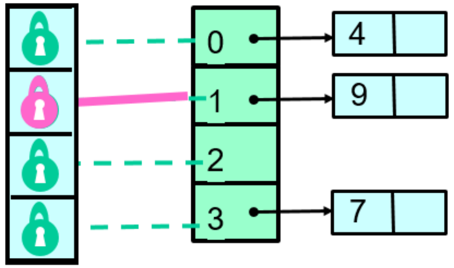
- M – maximum number of slots to be checked to look for an available bucket before declaring the table is too full
- As used in MemC3, B=4 , M=2000
- With DFS, the maximum number of displacements for a single Insert is 250, whereas with L_{BFS} = 5
- Indeed, $\lceil \log_4(250) \rceil \approx 5$

Increase Set-associativity

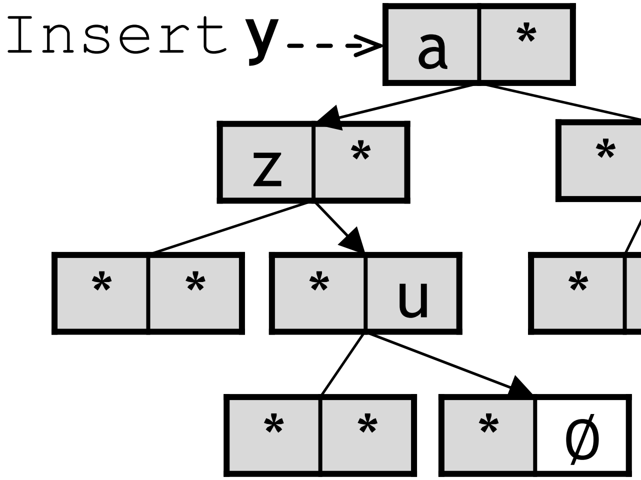


- higher set-associativity improves space utilization
- It leads to **lower** read throughput, since each *Lookup* must scan up to 2B slots from two buckets in an B-way set-associative hash table. But what if a bucket fits in a cache line?
- It may **improve** write throughput, because each Insert can read fewer random buckets (with fewer cache misses) to find an empty slot, and needs fewer item displacements to insert a new item *
- To achieve a good balance, they use a **8-way set-associative** hash table

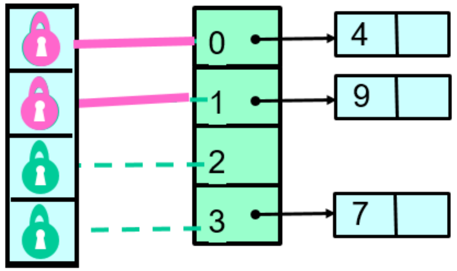
Fine-grained Locking



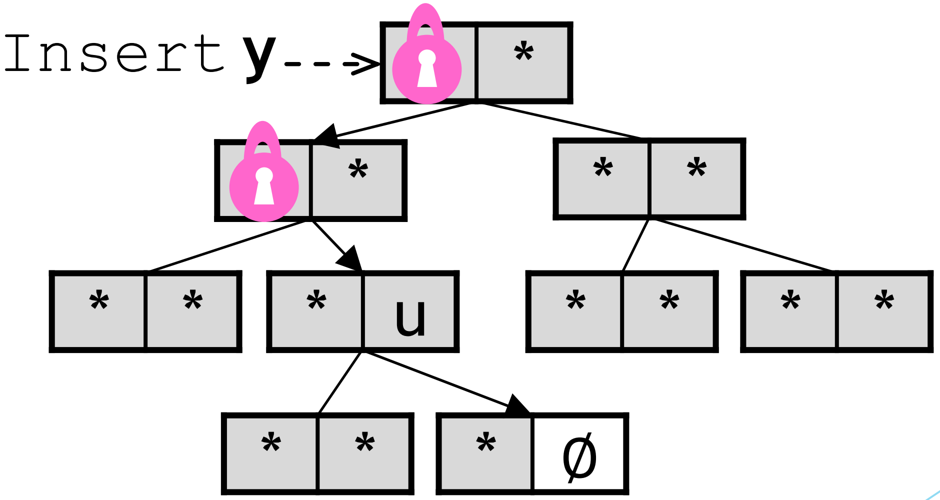
- There are high deadlock and livelock risks
- Maintain an actual lock in the stripe in addition to counter
- To Insert each new key-value pair, there is at most one new item inserted and four item displacements



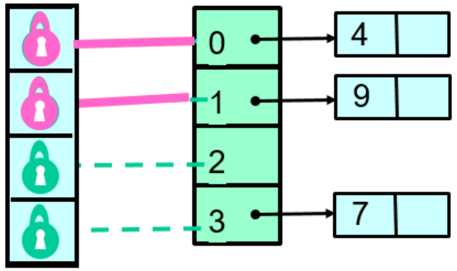
Fine-grained Locking



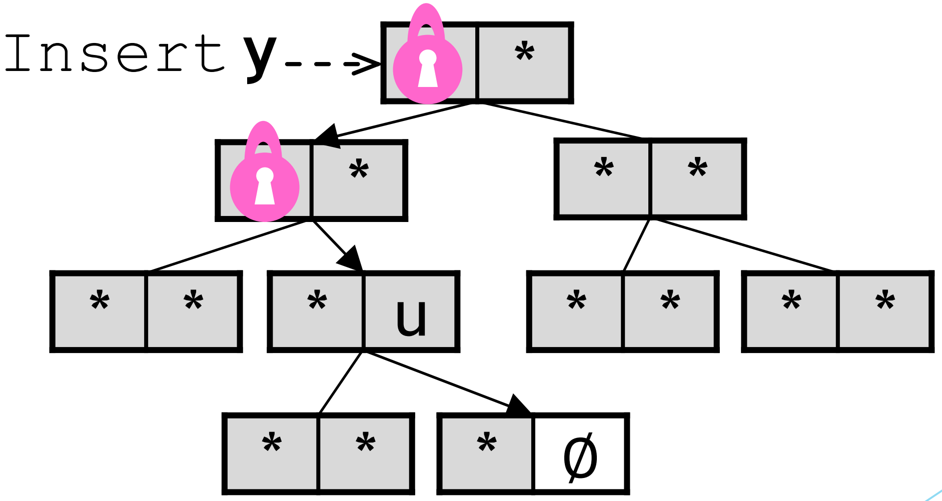
- Each insert or displacement involves **exactly two buckets**.
- The Insert operation only locks the pair of buckets associated with ongoing insertion or displacement.
- Release the lock immediately after complete, before locking the next pair.
- Locks of the pair of buckets are ordered by the bucket id to avoid deadlock.



Fine-grained Locking



- If two buckets share the same lock, then only one lock is acquired and released during the process.
- In summary, a writer must only lock at most five (usually fewer than three) pairs of buckets sequentially for an Insert operation.



Optimizing for Intel TSX

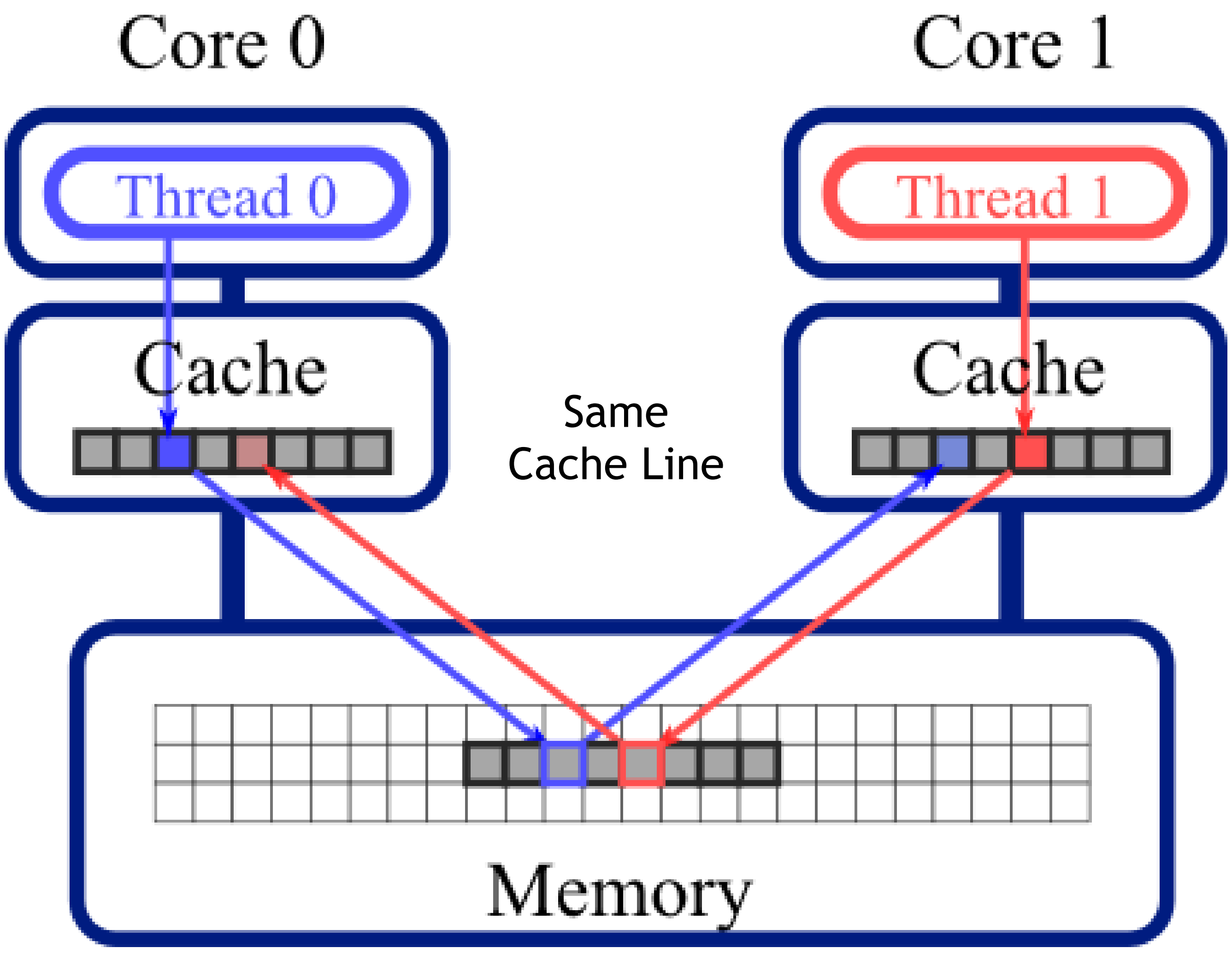
- As mentioned, naive use of TSX lock elision with a global lock **does not** provide high multi-threaded throughput
- So the key is to reduce the "transactional abort rate"
- Transactions abort for 3 common reasons:

1. Data conflicts
   - a cache line in its readset is written by another thread
2. Limited resources for transactional stores
   - There is not enough space to buffer its reads and writes in L1 cache
3. TSX-unfriendly instructions
   - MALLOC, XABORT, PAUSE etc.

Optimizing for Intel TSX - Conclusions

- Transactions that touch more memory are more likely to conflict with others, as well as to exceed the L1-cache limited capacity
- Transactions that take longer to execute are more likely to conflict with others
- Sharing of commonly-accessed data, such as global statistics counters, can greatly increase conflicts
- False sharing (possible solution: padding)

False sharing



Optimizing for Intel TSX - Conclusions

- The algorithmic optimizations reduce the size of the transactional region in a cuckoo *Insert* process from hundreds of bucket reads/writes to only a few bucket writes
- So it reduces the transactional **abort rate** caused by data conflicts or limited transactional stores

- Regarding unfriendly instruction, It is useful to pre-allocate structures that may be needed inside the transactional region

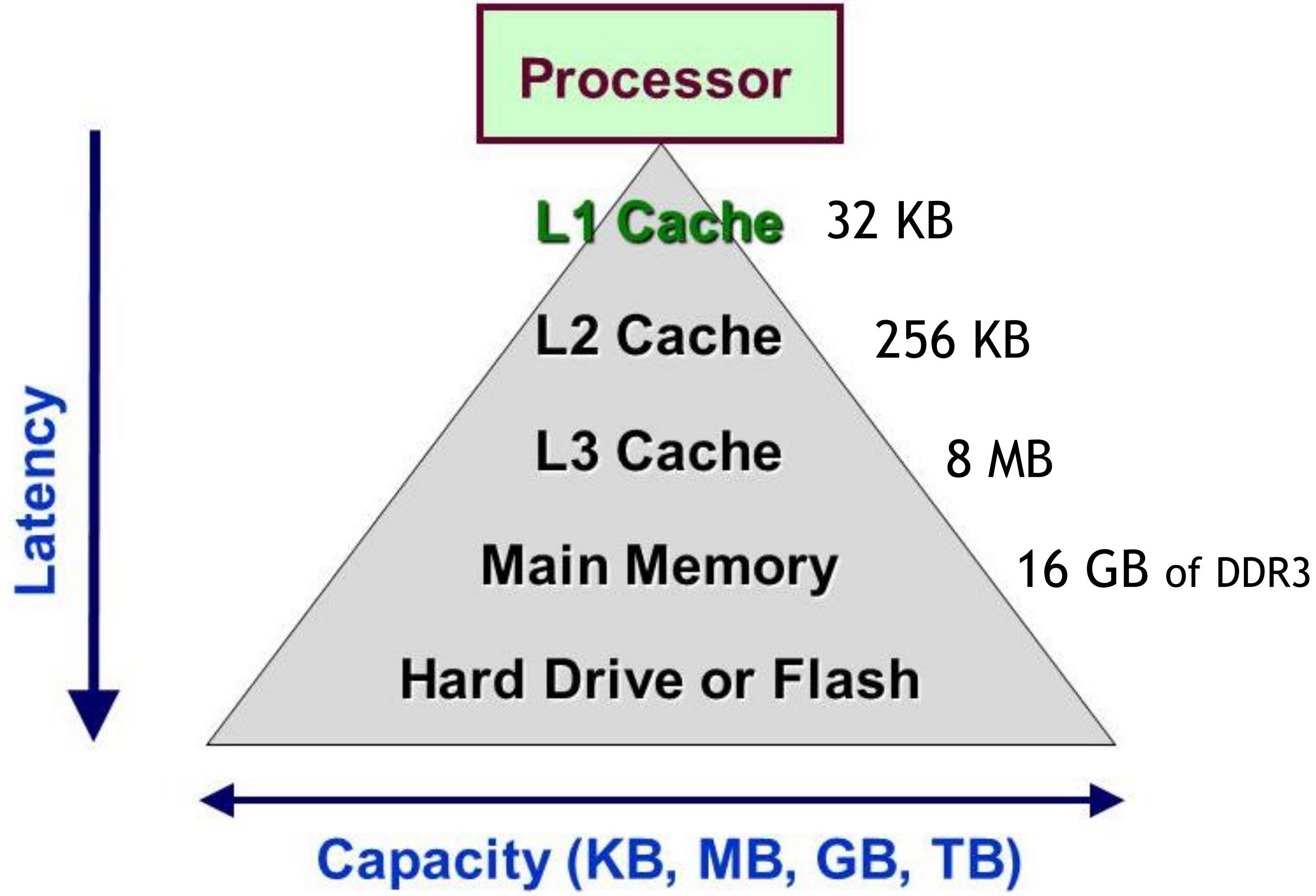
Evaluation

- How does the performance scale?
  - throughput vs. # of cores
- How much each technique improves performance?
  - algorithmic optimizations
  - lock elision with Intel TSX

Evaluation

- Platform
  - Intel Haswell i7-4770 @ 3.4GHz (with TSX support)
  - 4 cores (8 hyper-threaded cores)
- Cuckoo hash table
  - 8 byte keys and 8 byte values
  - 2 GB hash table, ~134.2 million entries
  - 8-way set-associative
- Workloads
  - Fill an empty table to 95% capacity
  - Random mixed reads and writes

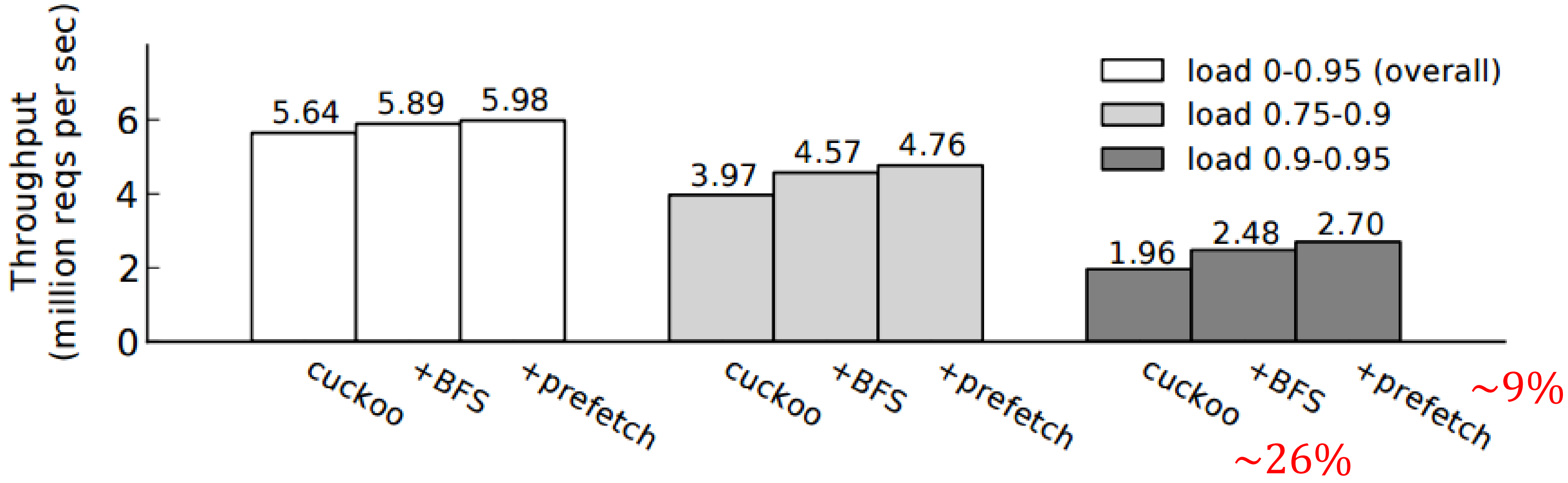
Evaluation – Memory Hierarchy



Evaluation

- **cuckoo:** The optimistic concurrent multi-reader/single-writer cuckoo hashing. Each Insert locks the whole hash table.
- **+lock later:** Lock after discovering a cuckoo path.
- **+BFS:** Look for an empty slot by breadth-first search.
- **+prefetch:** Prefetch the next bucket into cache.
- **+TSX-glibc:** Use the released glibc TSX lock elision to support concurrent writers.
- **+TSX*:** Use their TSX lock elision implementation that is optimized for short transactions instead of TSX-glibc

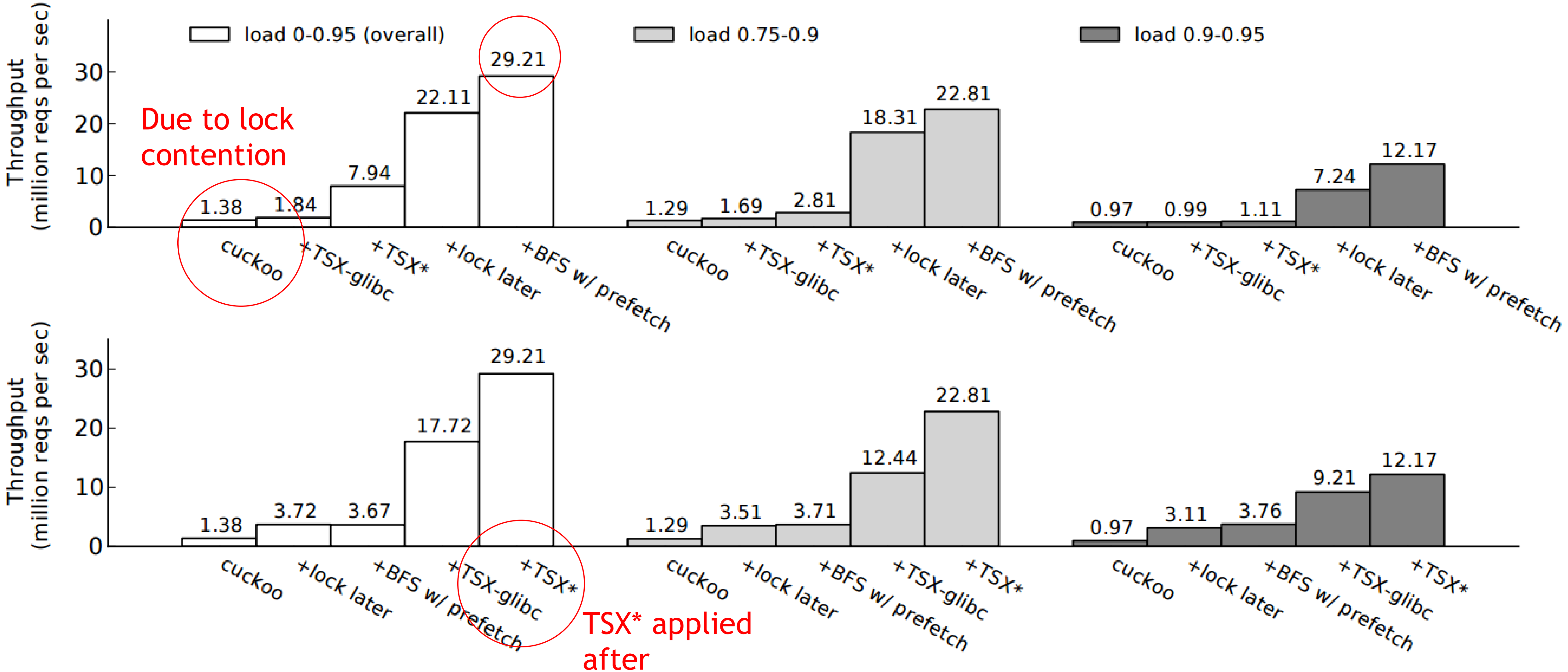
Single-thread Insert performance



(a) Single thread Insert performance (all locks disabled)

Multi-thread Insert performance

Both data structure and concurrency control optimizations are needed to achieve high performance



(b) Aggregate `Insert` performance of 8 threads, with locking

Multi-core Scaling Comparison

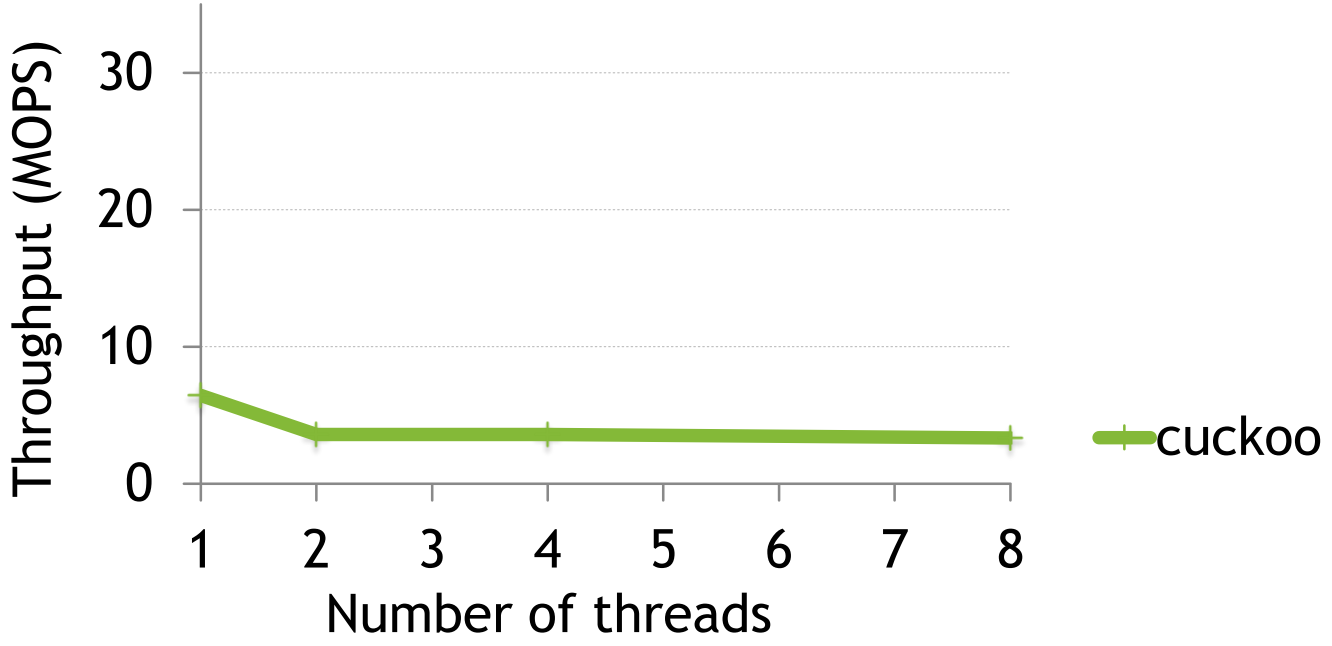
Let:

- "cuckoo" - optimistic cuckoo hashing used in MemC3
- "cuckoo+" - cuckoo with optimizations we have seen:
  - Lock After Discovering a Cuckoo Path
  - BFS and Prefetching
  - 8-way set-associative hash table

As we will see:

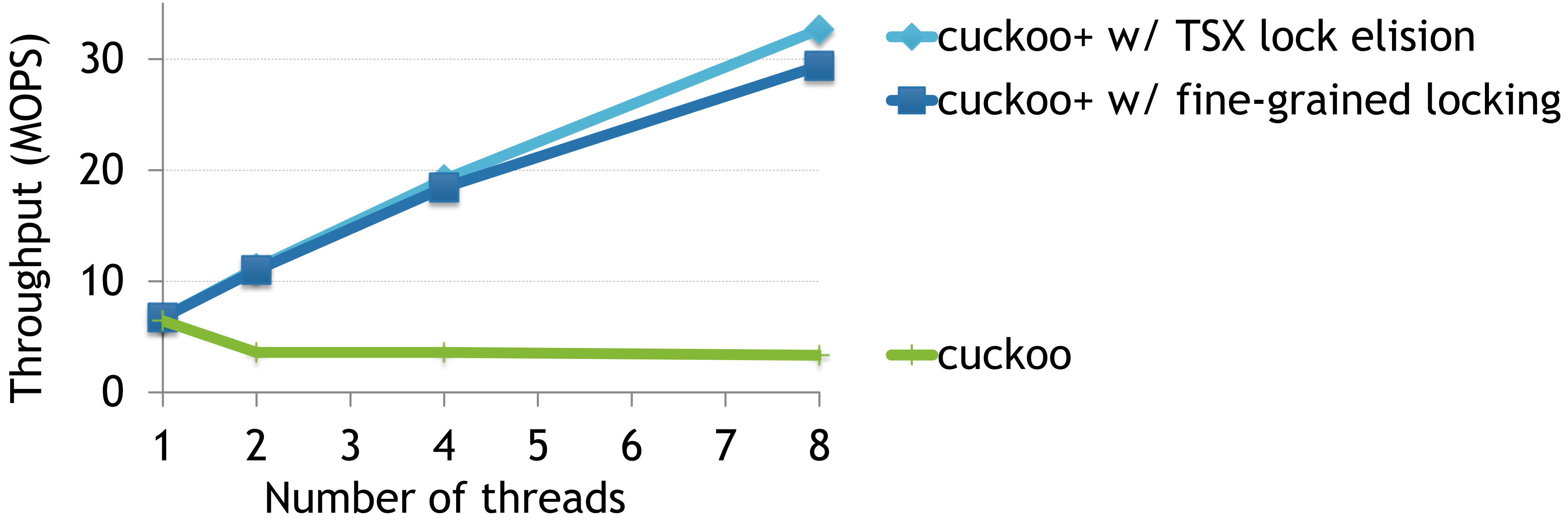
- Cuckoo+ scales well as the number of cores increases, on 4-core Haswell

Multi-core scaling comparison (50% Insert)



cuckoo: single-writer/multi-reader [MemC3, Fan, NSDI’13]
cuckoo+: cuckoo with their algorithmic optimizations

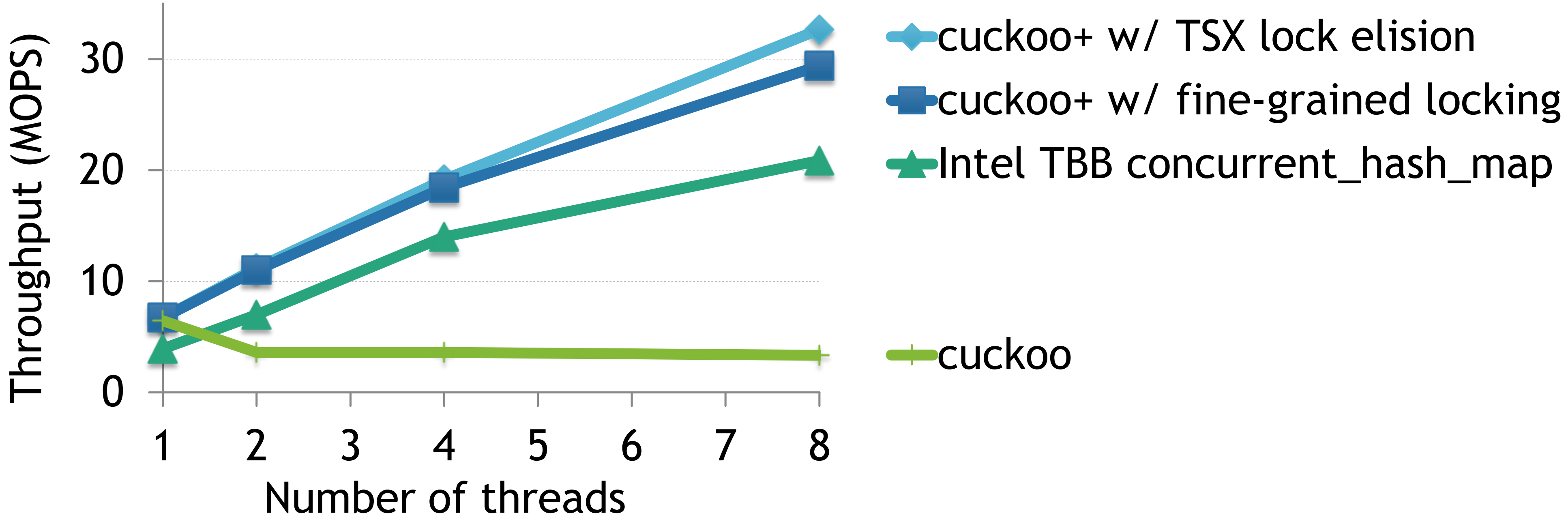
Multi-core scaling comparison (50% Insert)



cuckoo: single-writer/multi-reader

cuckoo+: cuckoo with their algorithmic optimizations

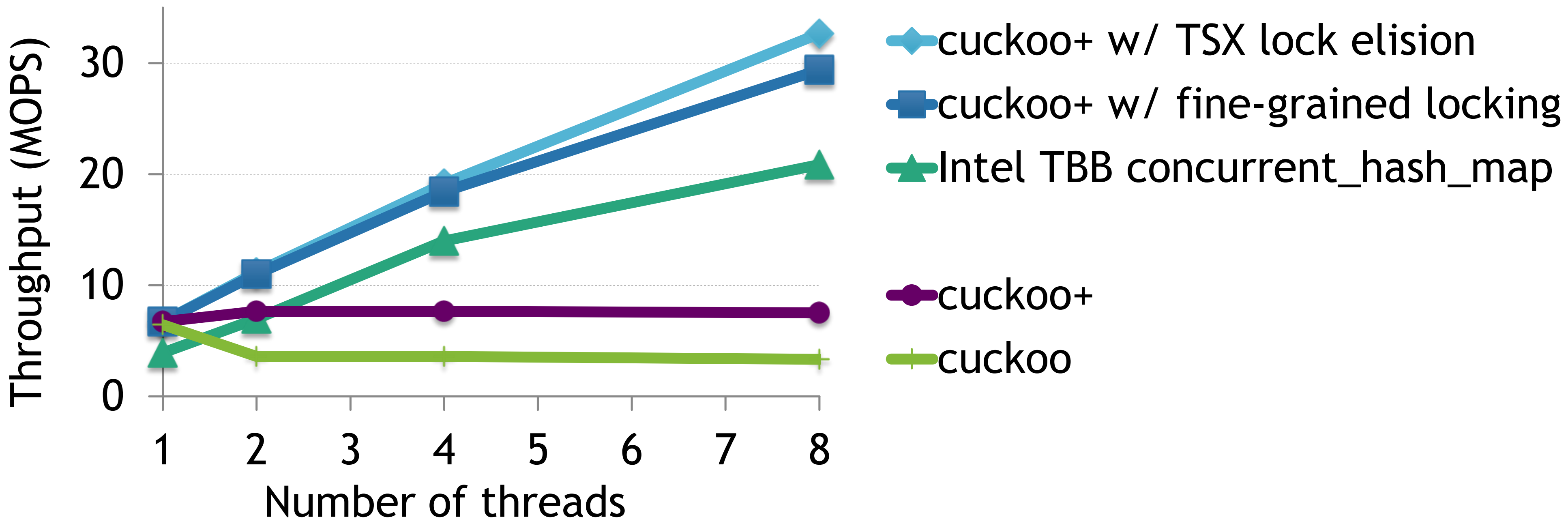
Multi-core scaling comparison (50% Insert)



cuckoo: single-writer/multi-reader

cuckoo+: cuckoo with their algorithmic optimizations

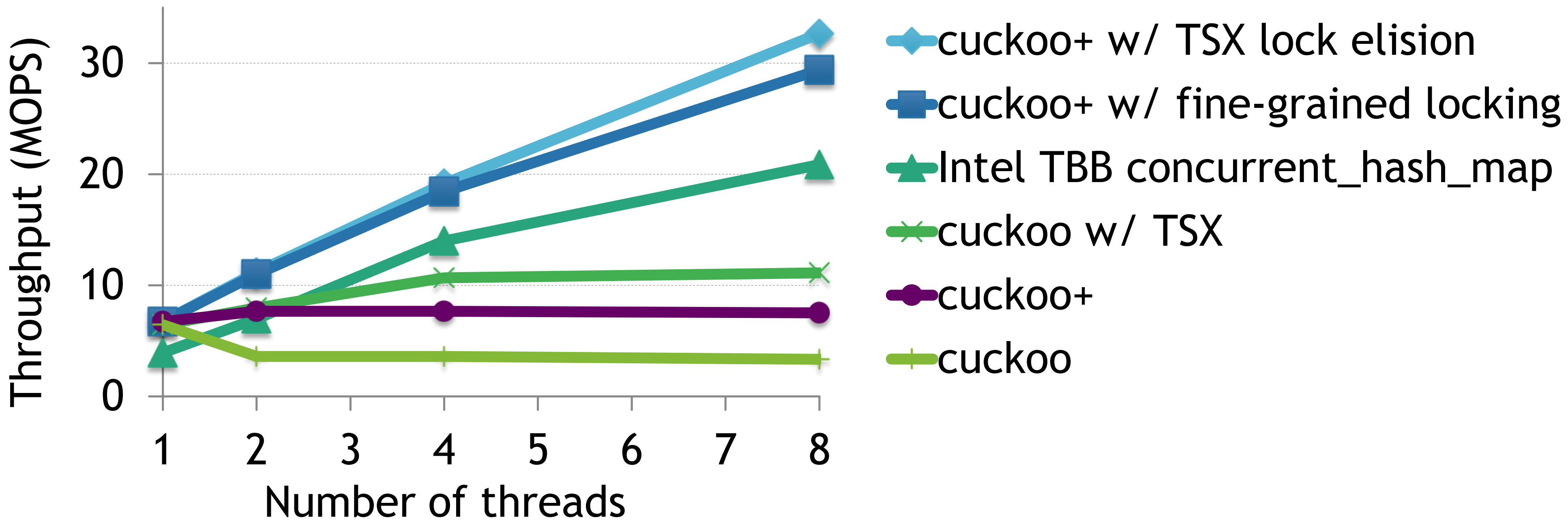
Multi-core scaling comparison (50% Insert)



cuckoo: single-writer/multi-reader

cuckoo+: cuckoo with their algorithmic optimizations

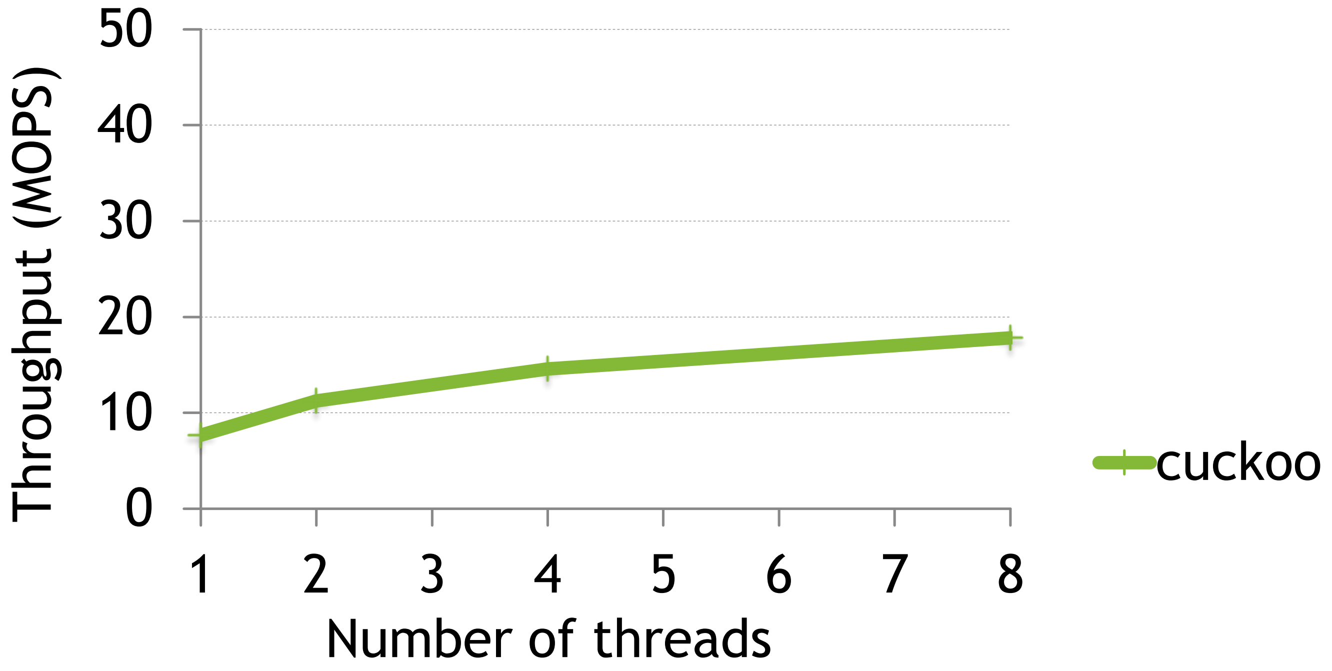
Multi-core scaling comparison (50% Insert)



cuckoo: single-writer/multi-reader

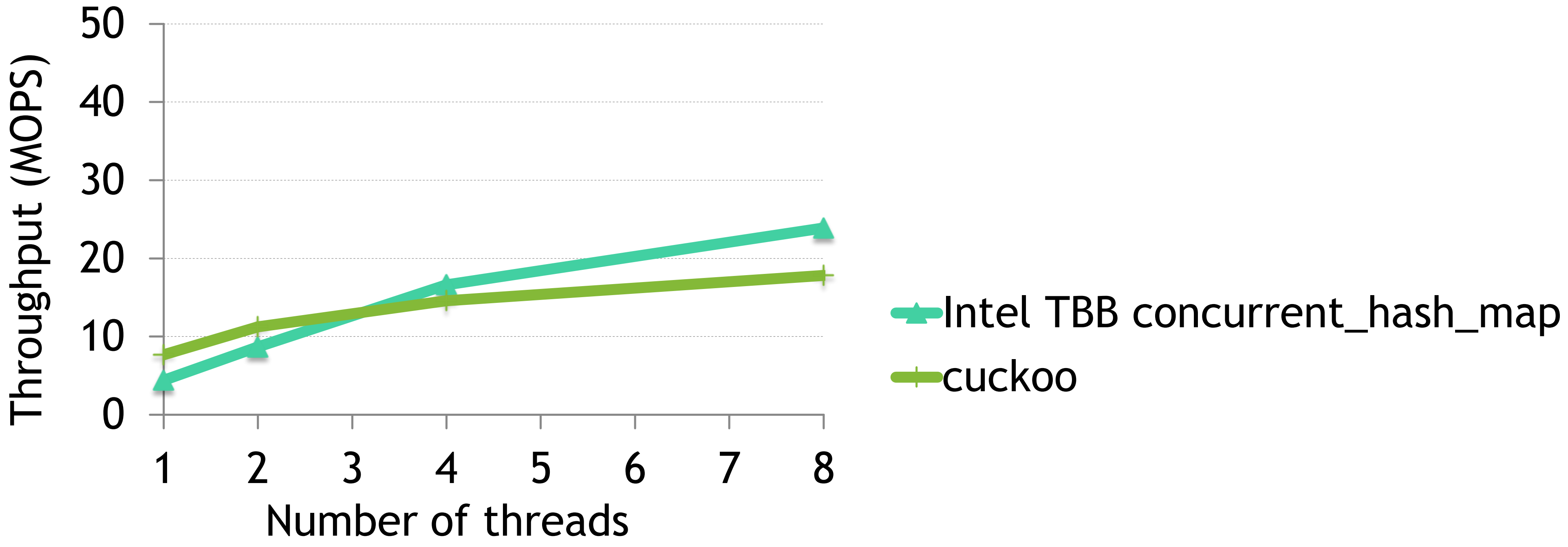
cuckoo+: cuckoo with their algorithmic optimizations

Multi-core scaling comparison (10% Insert)



cuckoo: single-writer/multi-reader
cuckoo+: cuckoo with their algorithmic optimizations

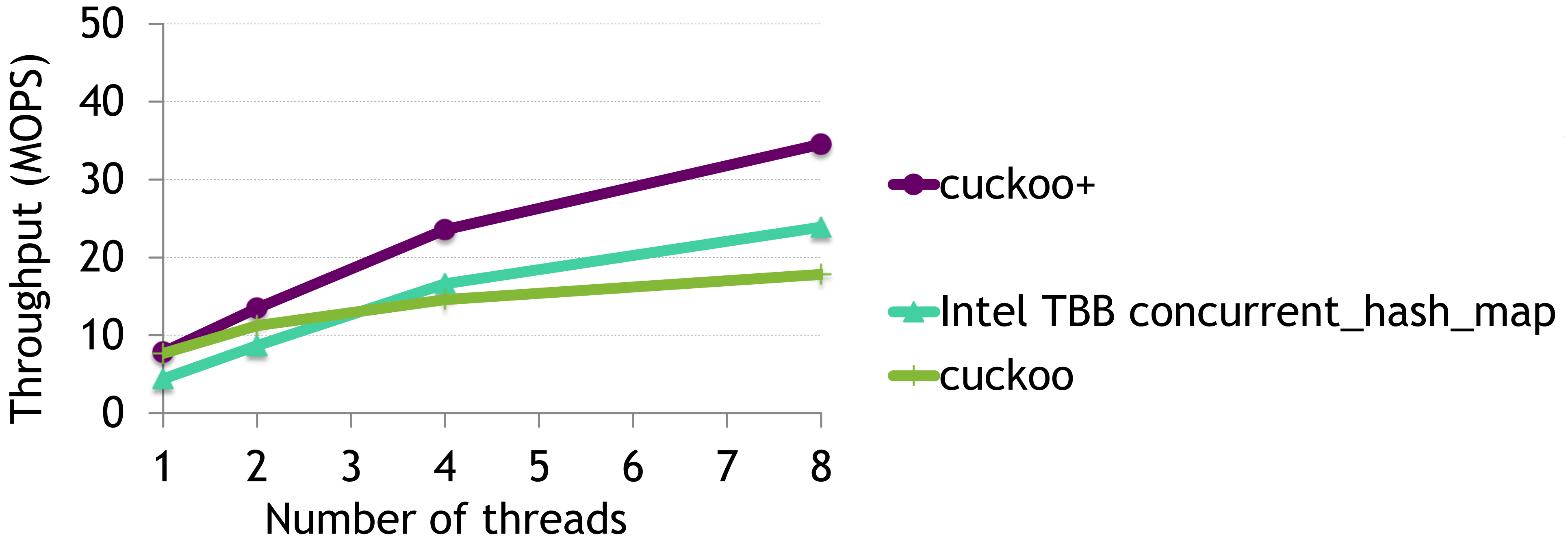
Multi-core scaling comparison (10% Insert)



cuckoo: single-writer/multi-reader

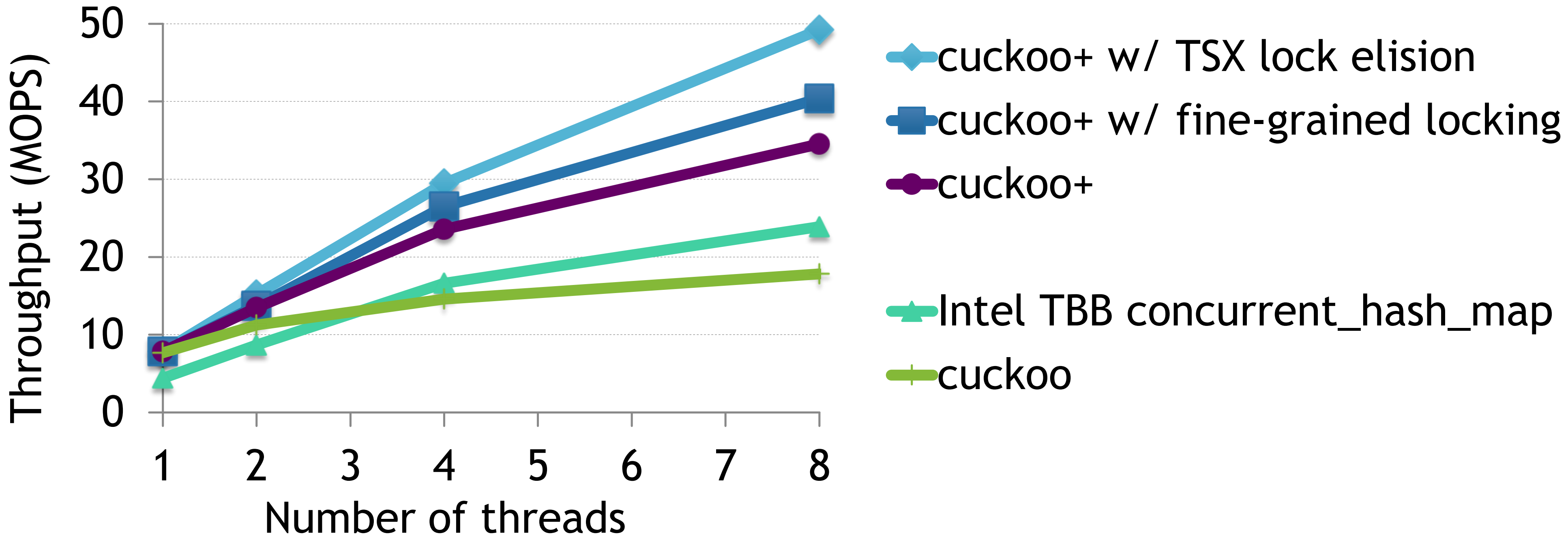
cuckoo+: cuckoo with their algorithmic optimizations

Multi-core scaling comparison (10% Insert)



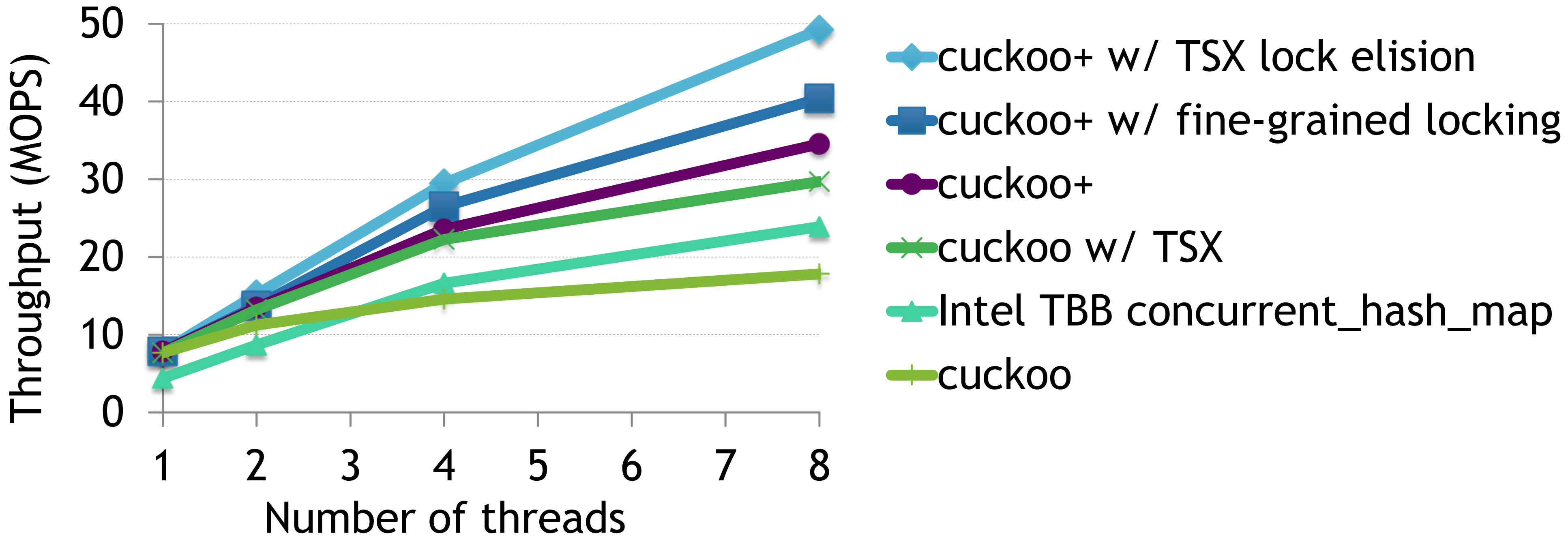
cuckoo: single-writer/multi-reader
cuckoo+: cuckoo with their algorithmic optimizations

Multi-core scaling comparison (10% Insert)



cuckoo: single-writer/multi-reader
cuckoo+: cuckoo with their algorithmic optimizations

Multi-core scaling comparison (10% Insert)



cuckoo: single-writer/multi-reader
cuckoo+: cuckoo with their algorithmic optimizations

Set-associativity and Load Factor - Lookup

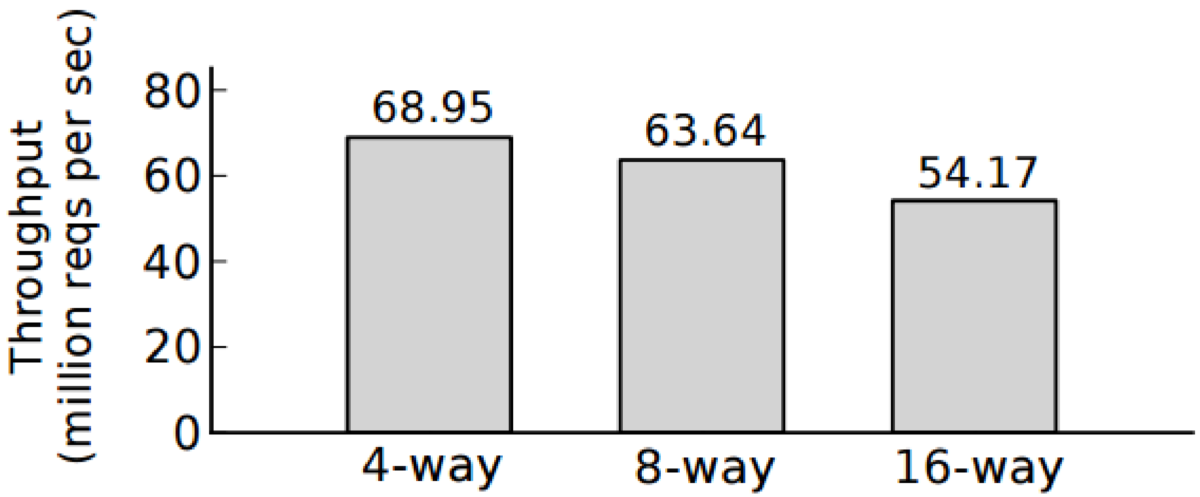


Figure 8: 8-thread aggregate `Lookup` throughput of hash tables with different set-associativities at 95% occupancy. Use optimized cuckoo hashing with TSX lock elision.

Lower associativity improves throughput, because each reader needs to check fewer slots in order to find the key

Set-associativity and Load Factor - Insert

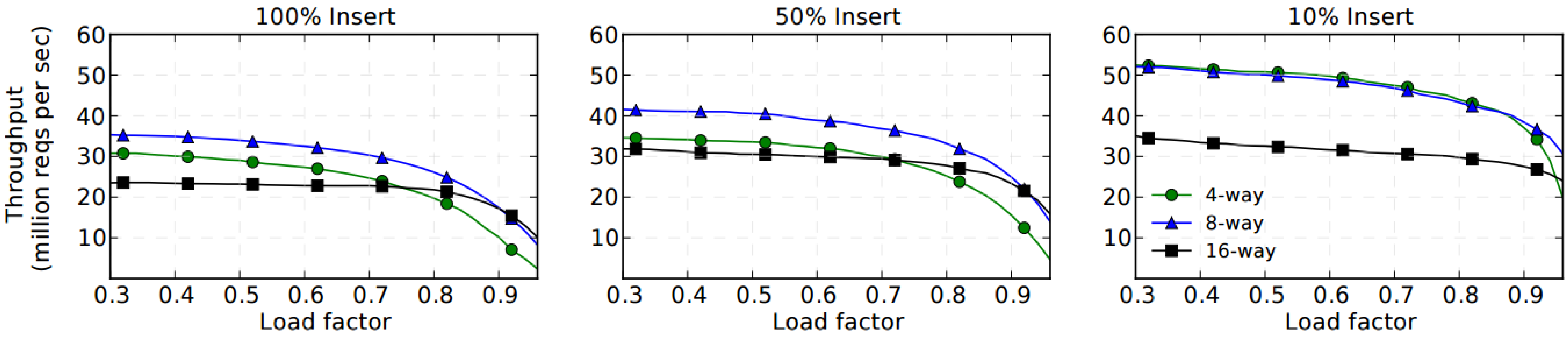
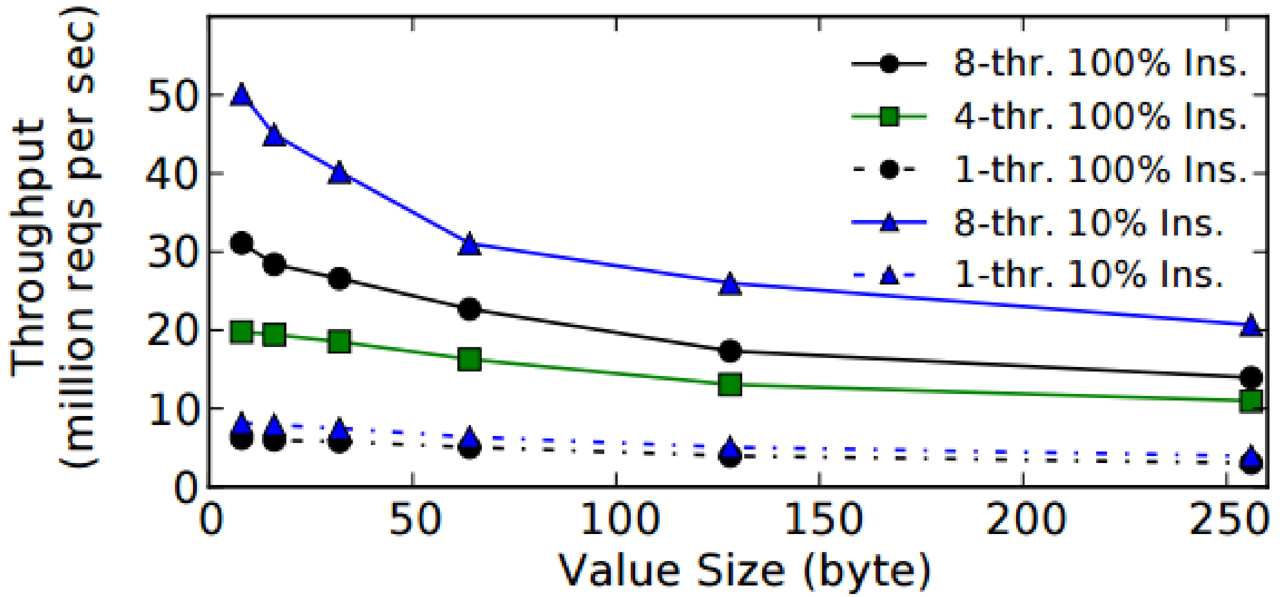


Figure 9: 8-thread aggregate throughput of hash tables with different set-associativities at different table occupancy. Use optimized cuckoo hashing with TSX lock elision.

8-way has the best overall performance.
It always outperforms 4-way for 100% and 50% Insert workloads,
and for 10% Insert workloads when the load factor is above 0.85.

Different Key-Value Sizes

- All previous experiments used workloads with 8 byte keys and 8 byte values.
- Evaluation of the cuckoo hash table performance with different value sizes:



(a) Hash table with fixed number ($\sim$ 33.4 million) of entries, using optimized cuckoo hashing with TSX lock elision.

Different Key-Value Sizes

- As expected, the throughput decreases as the value size increases because of the increased memory bandwidth needed

- Large values increase the amount of memory touched during the transaction and therefore increase the odds of a transactional abort

Conclusion

- Concurrent cuckoo hash table
  - high memory efficiency
  - fast concurrent writes and reads
- Lessons with hardware transactional memory
  - algorithmic optimizations are necessary