Algorithmic Improvements for Fast Concurrent Cuckoo Hashing

Xiaozhou Li (**Princeton**)

David G. Andersen (CMU)

Michael Kaminsky (Intel Labs)

Michael J. Freedman (Princeton)

In this talk

- How to build a fast concurrent hash table
    - algorithm and data structure engineering
- Experience with hardware transactional memory
    - does NOT obviate the need for algorithmic optimizations

Concurrent hash table

- Indexing key-value objects
  - `Lookup(key)`
  - `Insert(key, value)`
  - `Delete(key)`
- Fundamental building block for modern systems
  - System applications (e.g., kernel caches)
  - Concurrent user-level applications
- Targeted workloads: ***small objects, high rate***

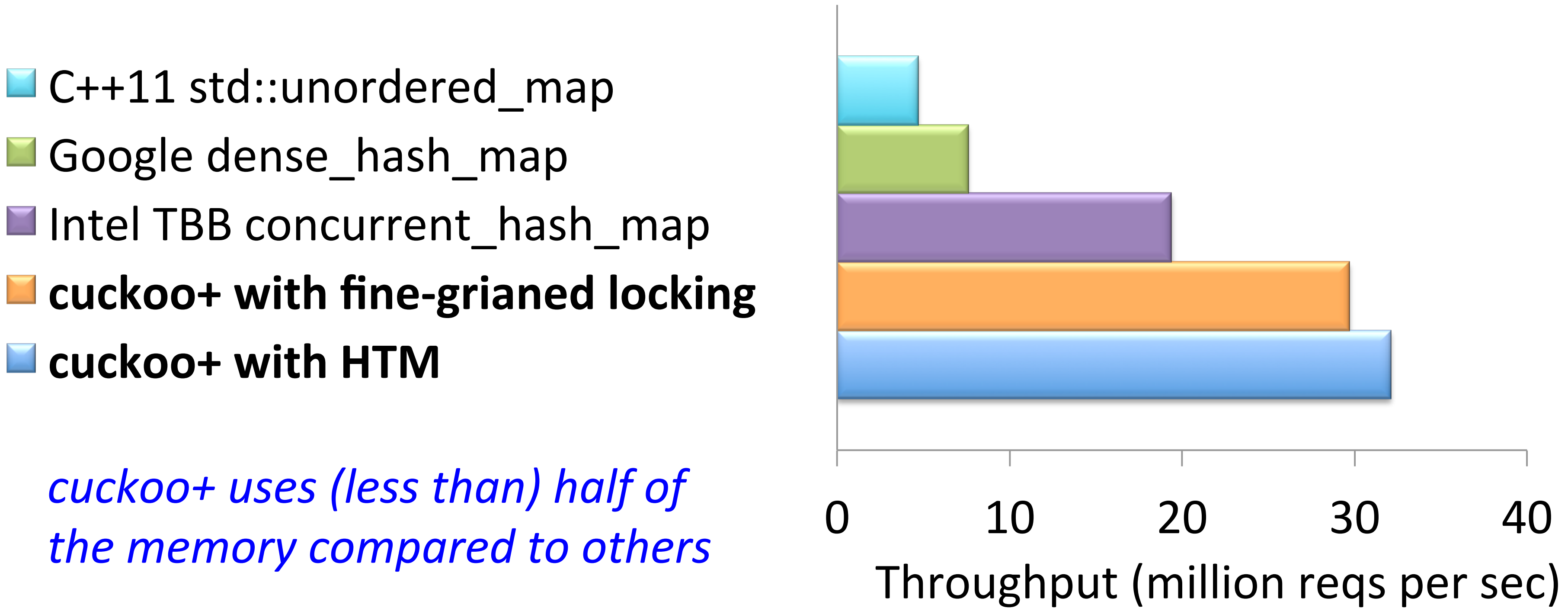
Goal: memory-efficient and high-throughput

- Memory efficient (e.g., > 90% space utilized)
- Fast concurrent reads (scale with # of cores)
- Fast concurrent writes (scale with # of cores)

Preview our results on a quad-core machine

64-bit key and 64-bit value

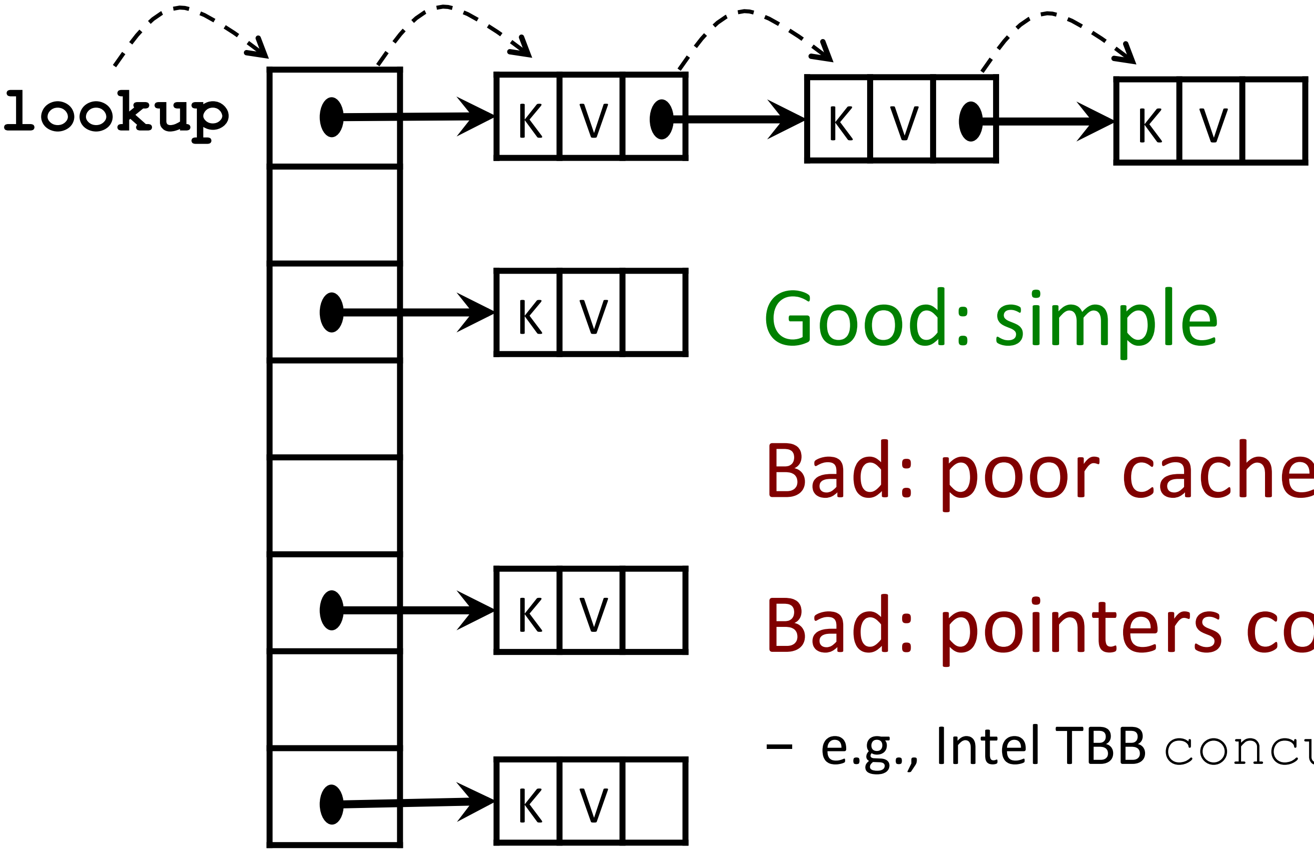
120 million objects, 100% `Insert`



cuckoo+ uses (less than) half of the memory compared to others

Background: separate chaining hash table

Chaining items hashed in same bucket



Good: simple

Bad: poor cache locality

Bad: pointers cost space

- e.g., Intel TBB `concurrent_hash_map`

Background: open addressing hash table

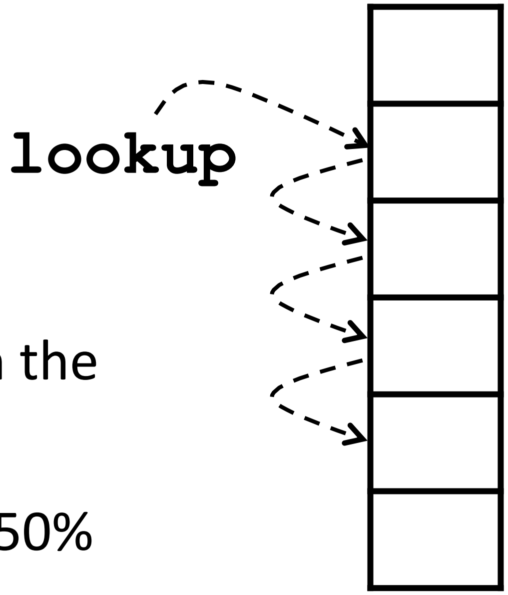
Probing alternate locations for vacancy

e.g., linear/quadratic probing, double hashing

Good: cache friendly

Bad: poor memory efficiency

- performance dramatically degrades when the usage grows beyond 70% capacity or so
- e.g., Google `dense_hash_map` wastes 50% memory by default.



Our starting point

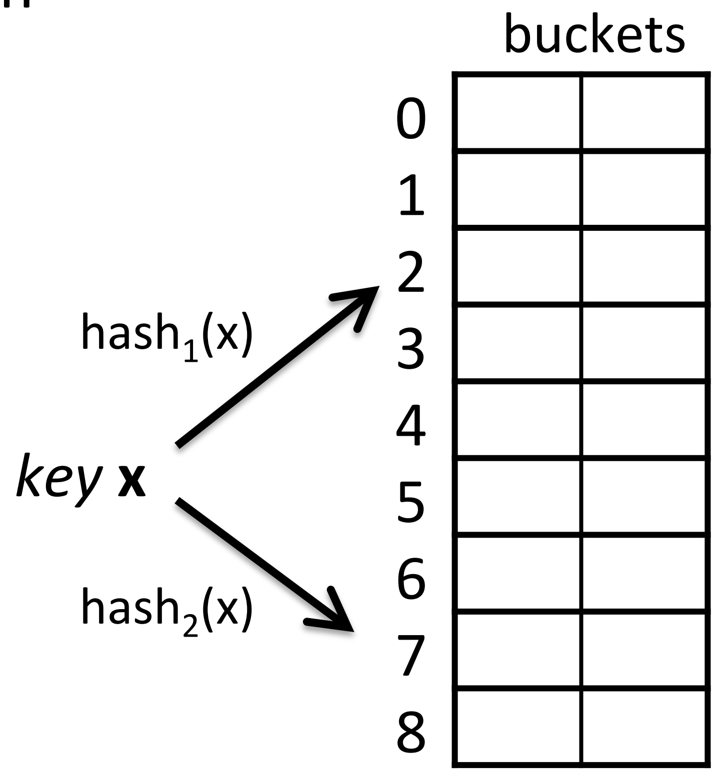
- Multi-reader single-writer cuckoo hashing [Fan, NSDI'13]
  - Open addressing
  - Memory efficient
  - Optimized for read-intensive workloads

Cuckoo hashing

Each bucket has *b* slots for items (*b*-way set-associative)

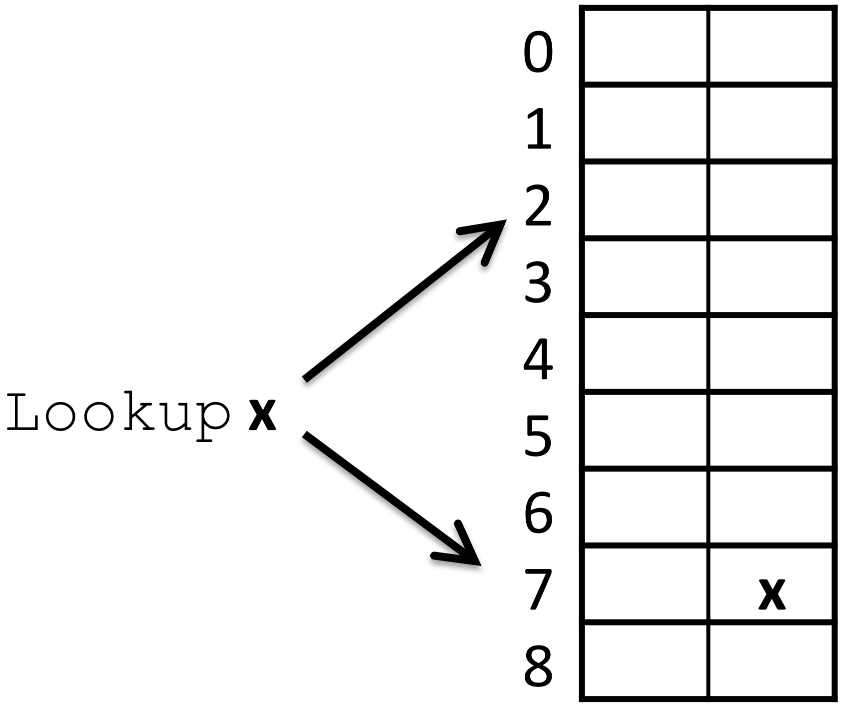
Each key is mapped to two random buckets

- stored in one of them



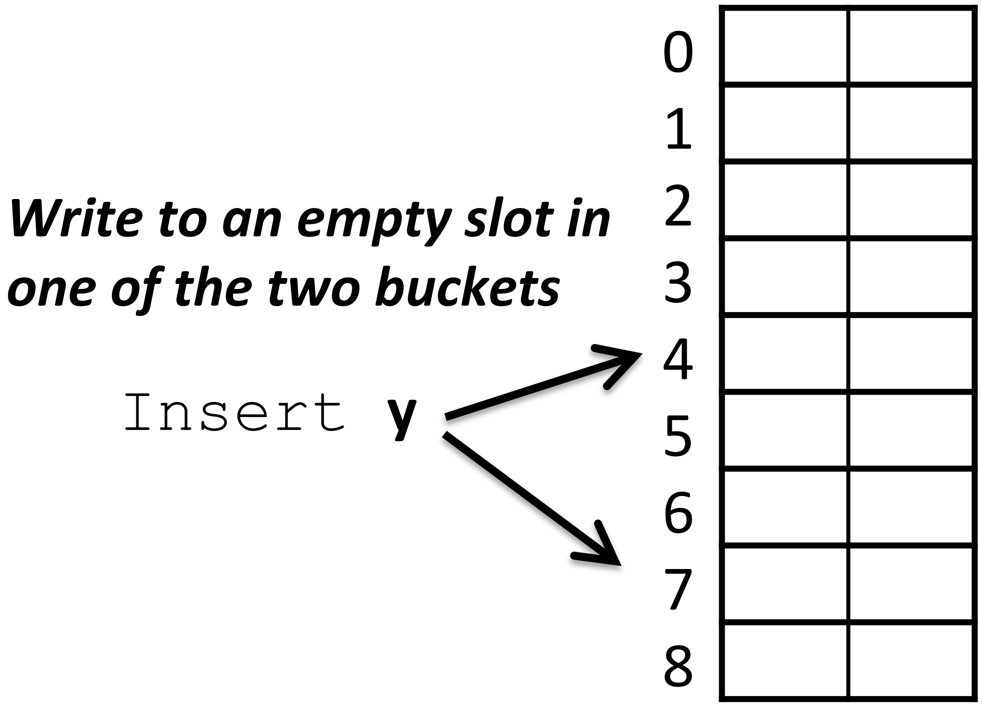
Predictable and fast lookup

- `Lookup`: read 2 buckets in parallel
  - constant time in the worst case



Insert may need "cuckoo move"

- `Insert:`

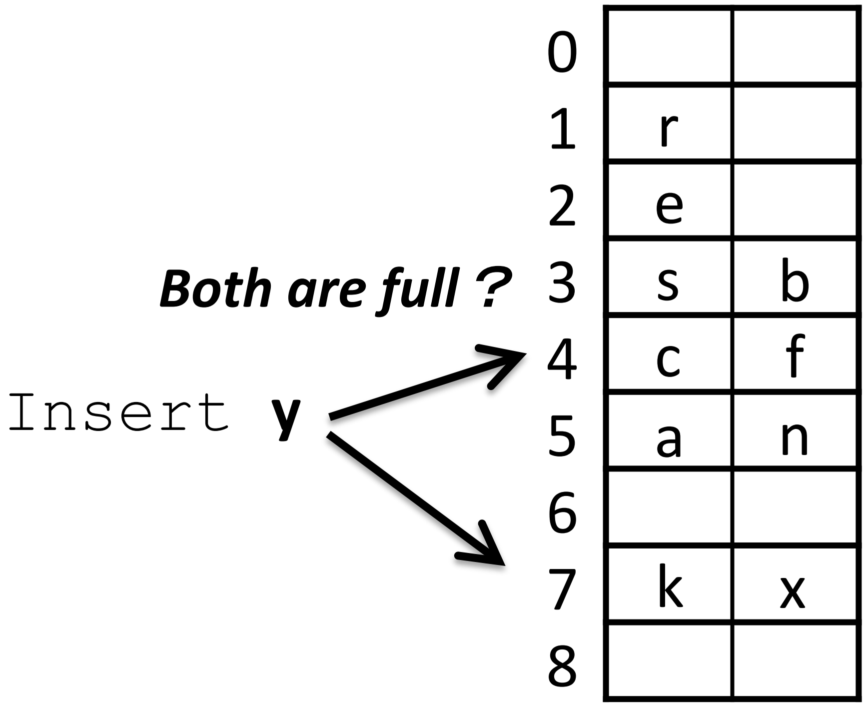


Write to an empty slot in one of the two buckets

`Insert` **y**

Insert may need “cuckoo move”

- `Insert:`



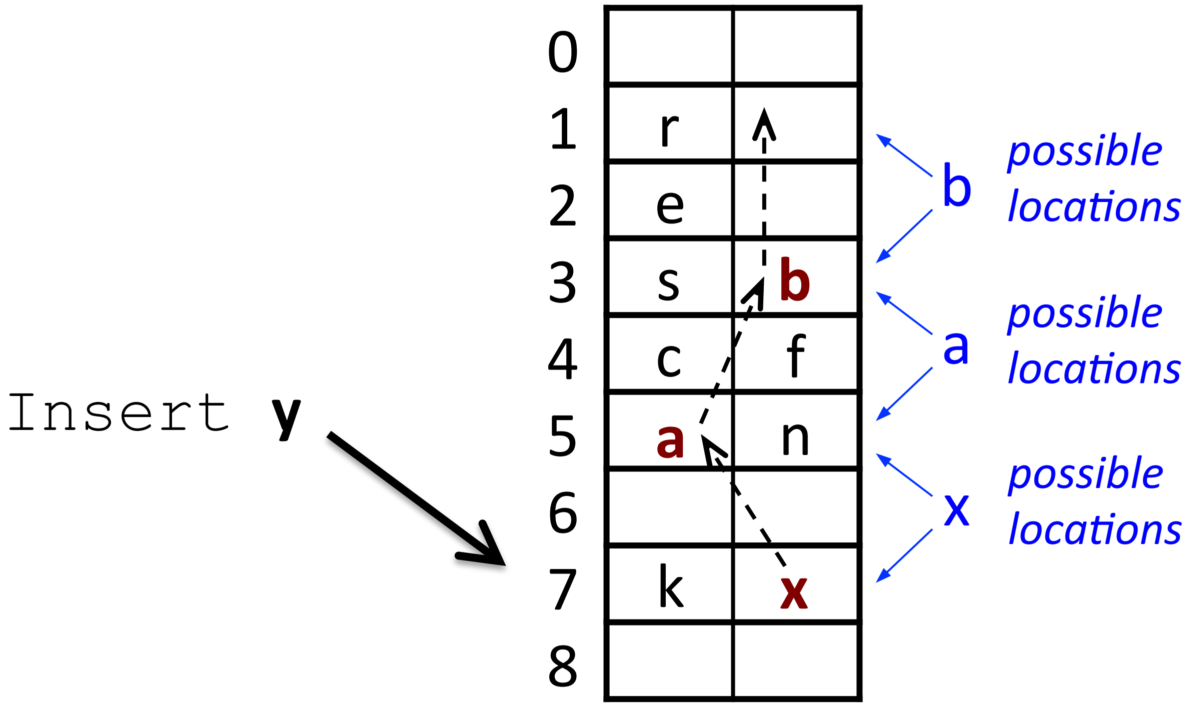
Both are full ?

`Insert` **y**

0		
1	r	
2	e	
3	s	b
4	c	f
5	a	n
6		
7	k	x
8		

Insert may need "cuckoo move"

- `Insert`: move keys to alternate buckets

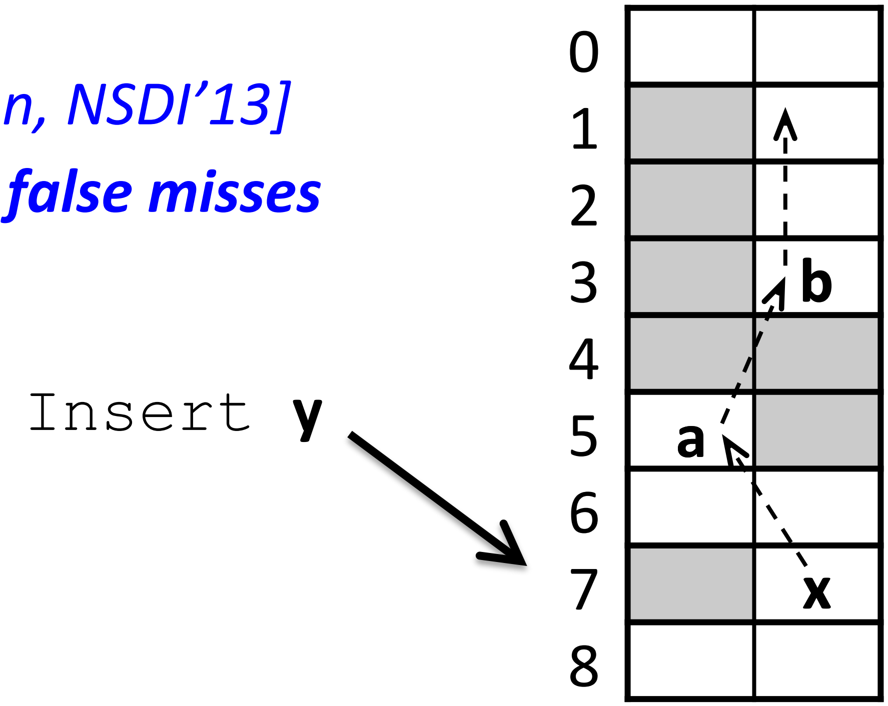


Insert may need "cuckoo move"

- `Insert`: move keys to alternate buckets
  - find a "cuckoo path" to an empty slot
  - move hole backwards

A technique in [Fan, NSDI'13]

No reader/writer false misses



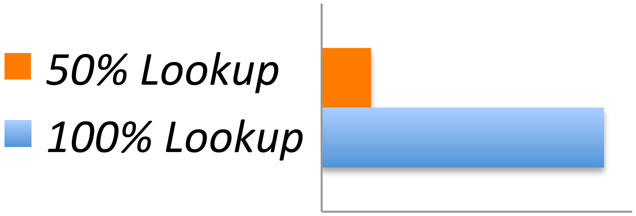
Review our starting point [Fan, NSDI'13]: Multi-reader single-writer cuckoo hashing

- 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



Improve write concurrency

- Algorithmic optimizations
    - Minimize critical sections
    - Exploit data locality
- Explore two concurrency control mechanisms
    - Hardware transactional memory
    - Fine-grained locking

Algorithmic optimizations

- Lock after discovering a cuckoo path
  - minimize critical sections
- Breadth-first search for an empty slot
  - fewer items displaced
  - enable prefetching
- Increase set-associativity (see paper)
  - fewer random memory reads

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

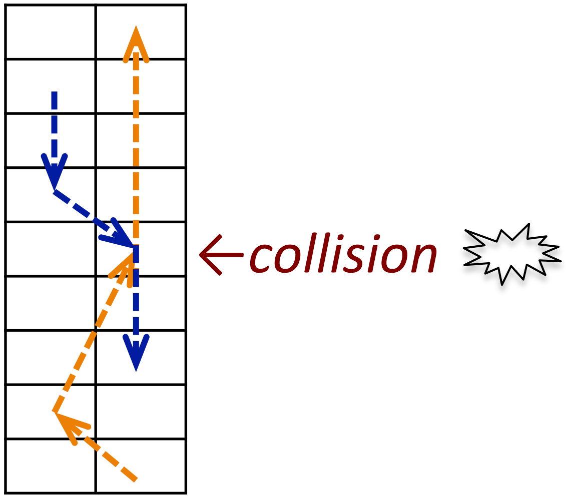
*All `Insert` operations of other threads are **blocked***

```
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();
```



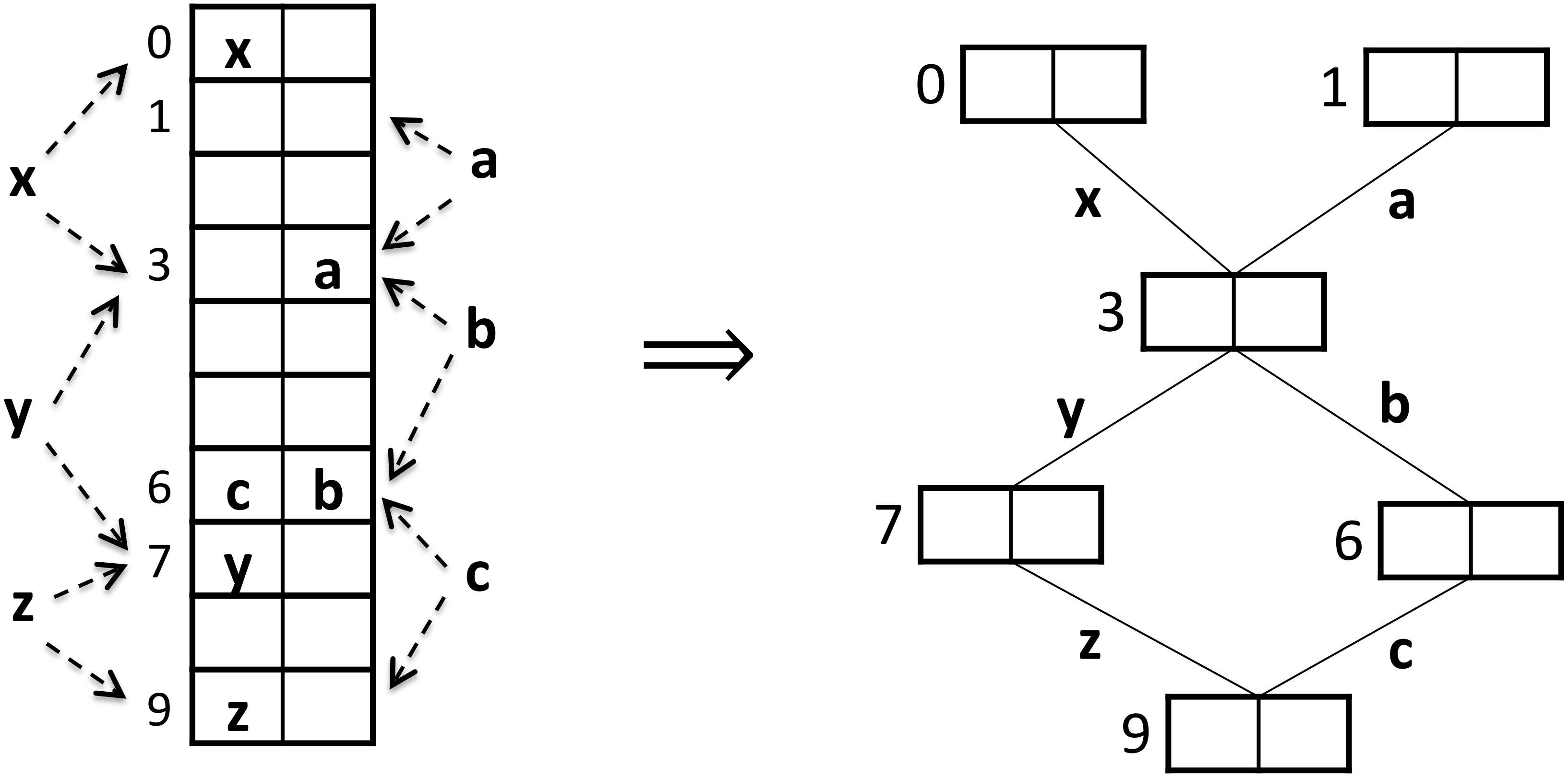
Lock after discovering a cuckoo path

Multiple `Insert` threads can look for cuckoo paths concurrently

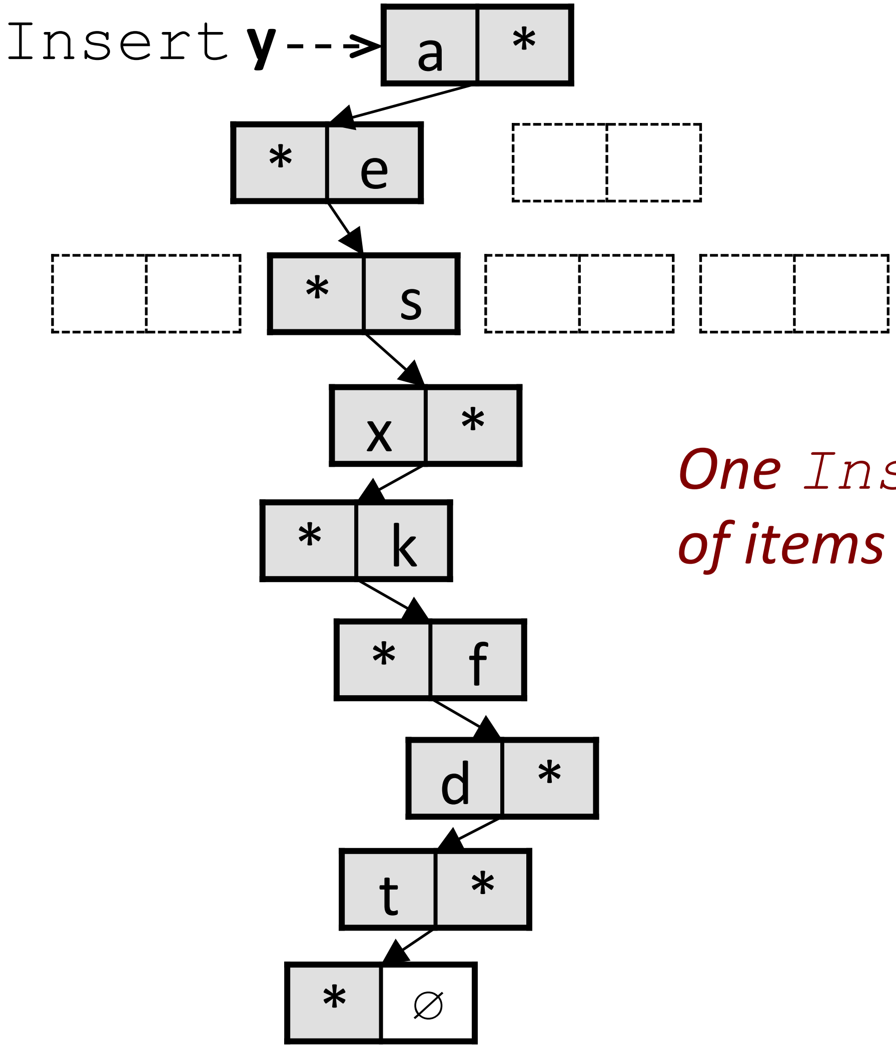
```
while(1) {
    Search for a cuckoo path;  // no locking required
    lock();
    Cuckoo move and insert while the path is valid;
    if(success)
        unlock();
        break;
    unlock();
}
```

Cuckoo hash table ⟹ undirected cuckoo graph

bucket ⟶ vertex
key ⟶ edge



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



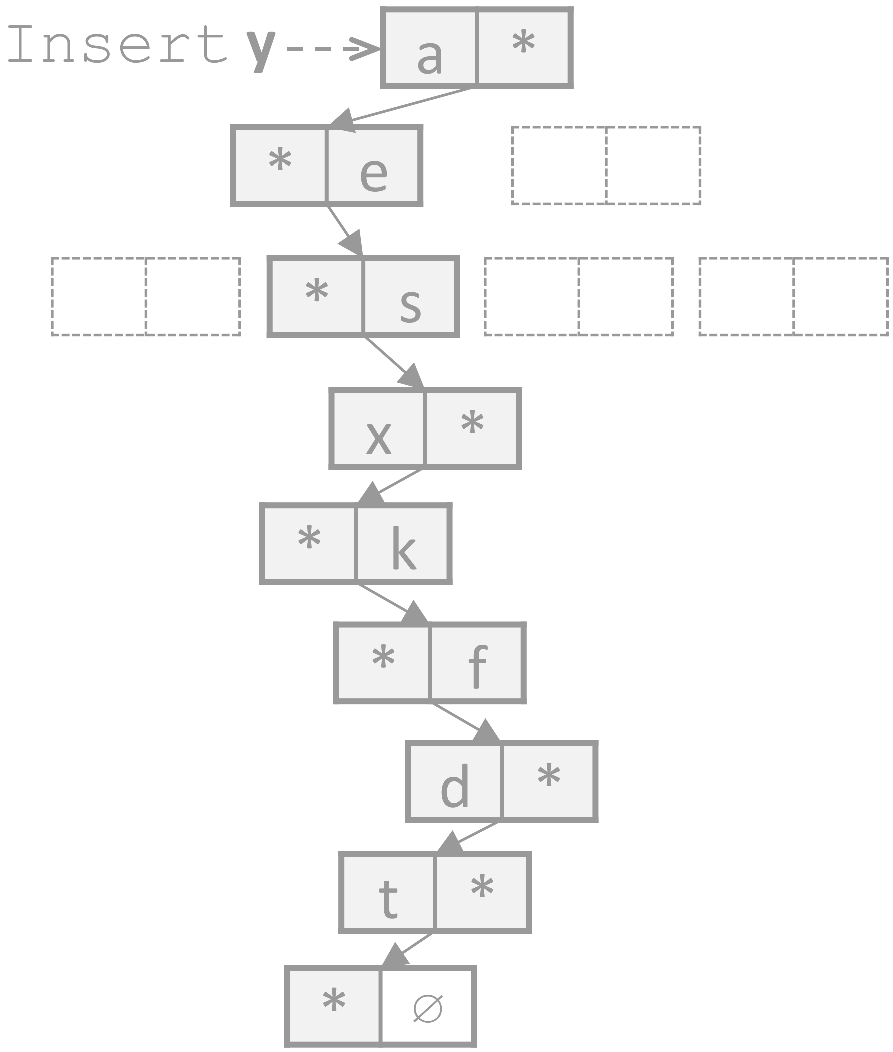
cuckoo path:

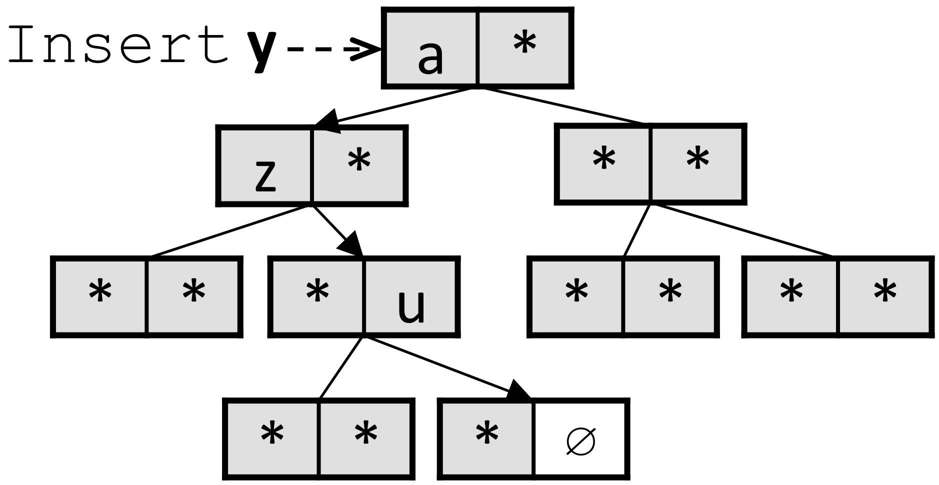
a→e→s→x→k→f→d→t→∅

9 writes

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

Breadth-first search for an empty slot





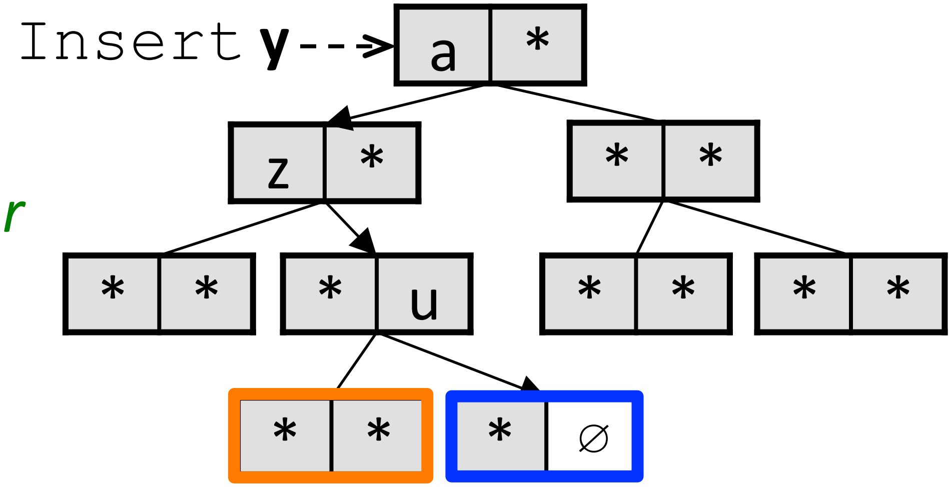
Breadth-first search for an empty slot

cuckoo path:

a→z→u→∅ **4 writes**

Reduced to a ***logarithmic*** *factor*

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

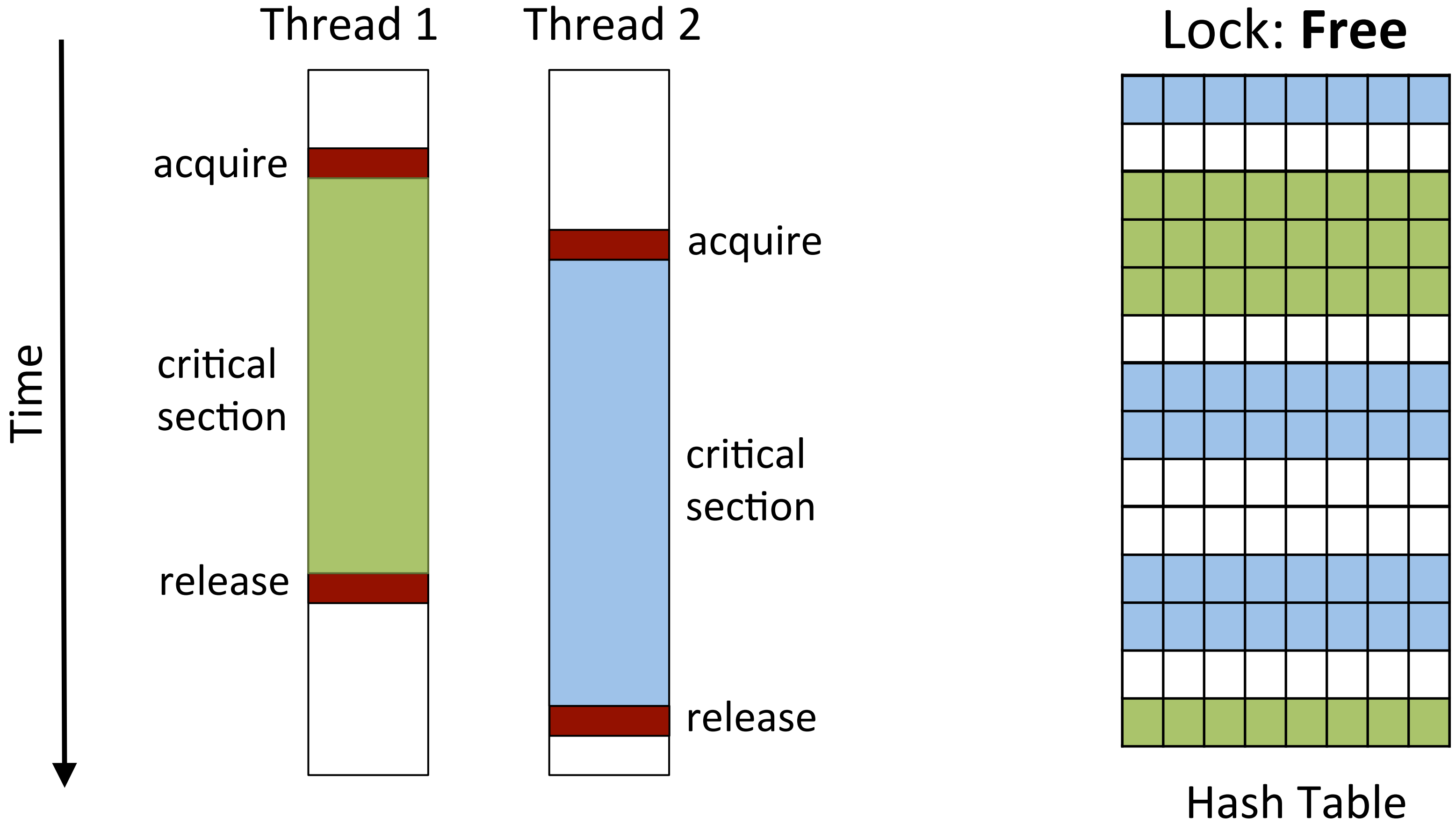


Prefetching: scan one bucket and load next bucket concurrently

Concurrency control

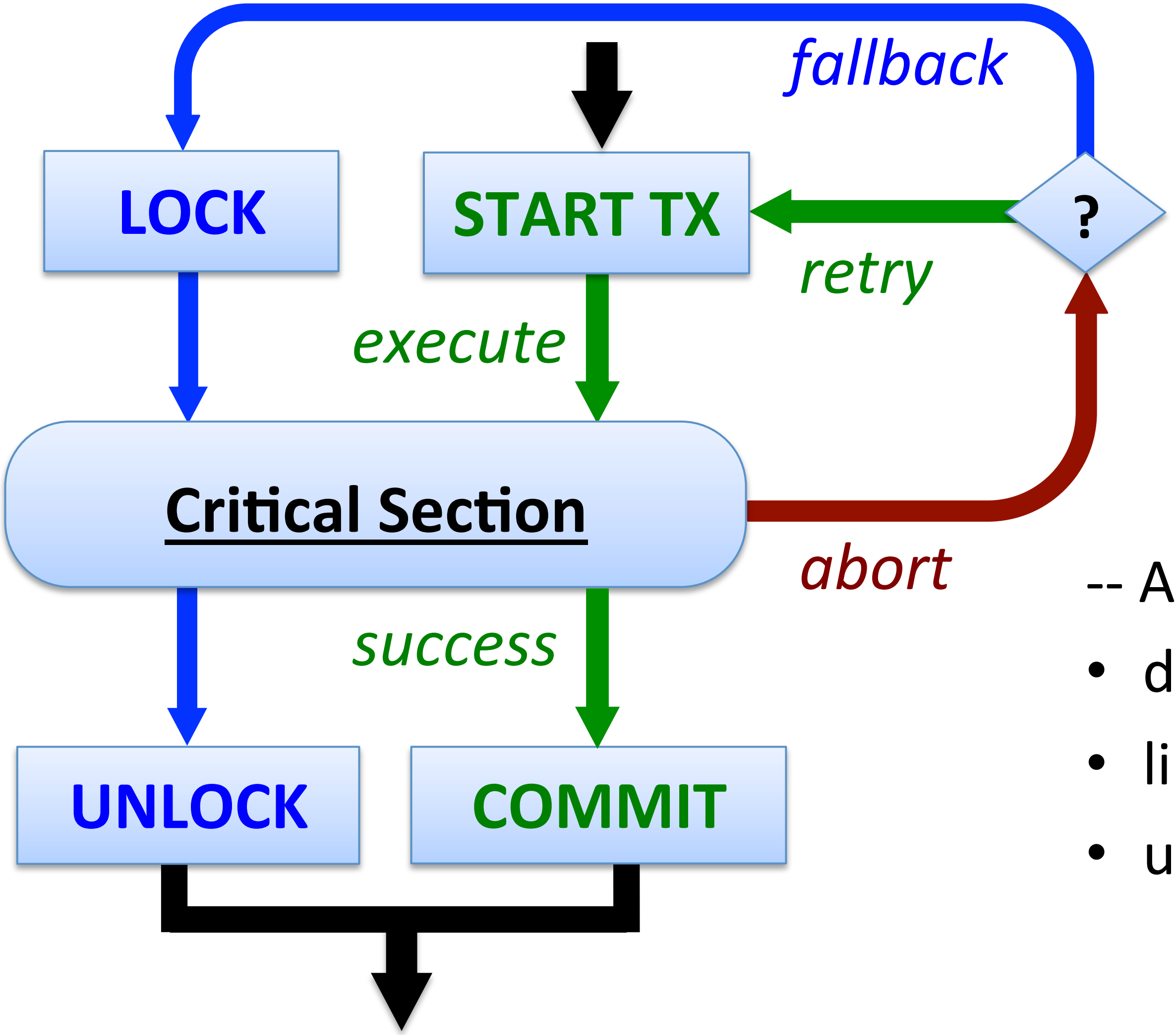
- Fine-grained locking
  - spinlock and lock striping
- Hardware transactional memory
  - Intel Transactional Synchronization Extensions (TSX)
  - Hardware support for lock elision

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

Principles to reduce transactional aborts

1. Minimize the *size* of transactional regions.
    - Algorithmic optimizations
        - lock later, BFS, increase set-associativity

***Maximum size** of transactional regions*

previous cuckoo[Fan, NSDI'13]	optimized cuckoo
cuckoo search: **500 reads**	—
cuckoo move: **250 writes**	cuckoo move: **5 writes/reads**

Principles to reduce transactional aborts

2. Avoid unnecessary access to common data.
   - Make globals thread-local
3. Avoid TSX-unfriendly instructions in transactions
   - e.g., `malloc()` may cause problems
4. Optimize TSX lock elision implementation
   - Elide the lock more aggressively for short transactions

Evaluation

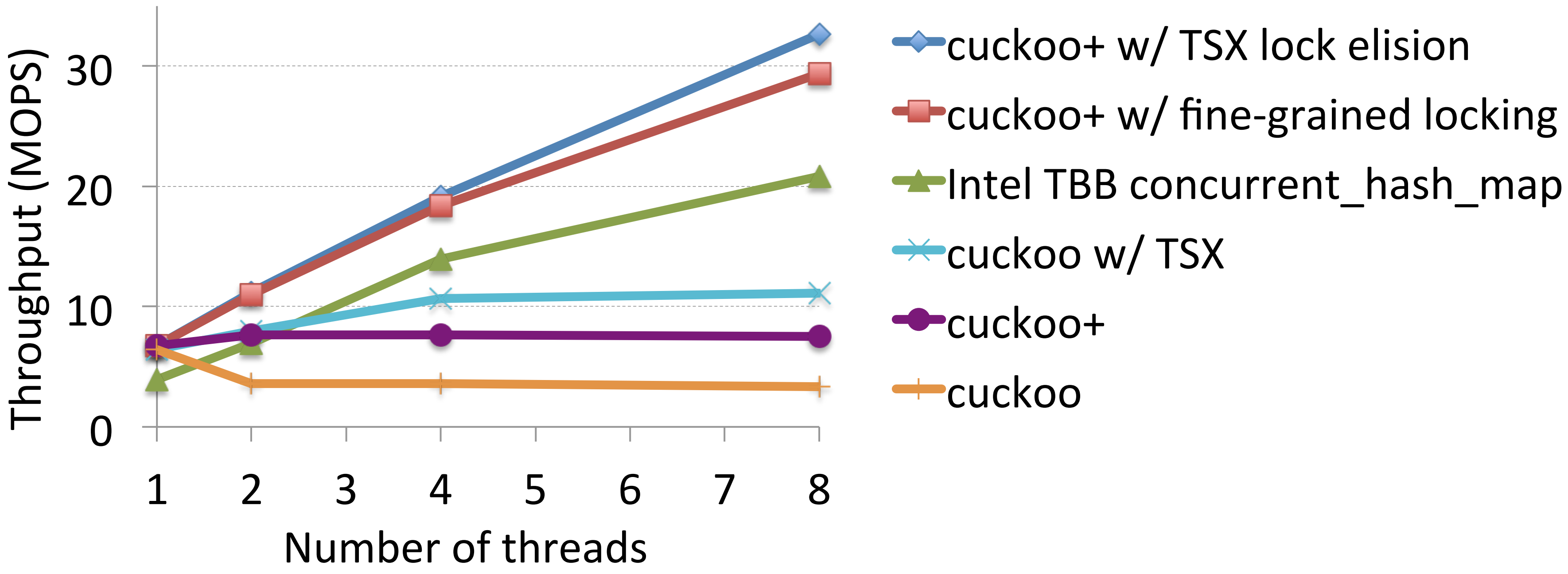
- How does the performance scale?
  - throughput vs. # of cores

- How much each technique improves performance?
  - algorithmic optimizations
  - lock elision with Intel TSX

Experiment settings

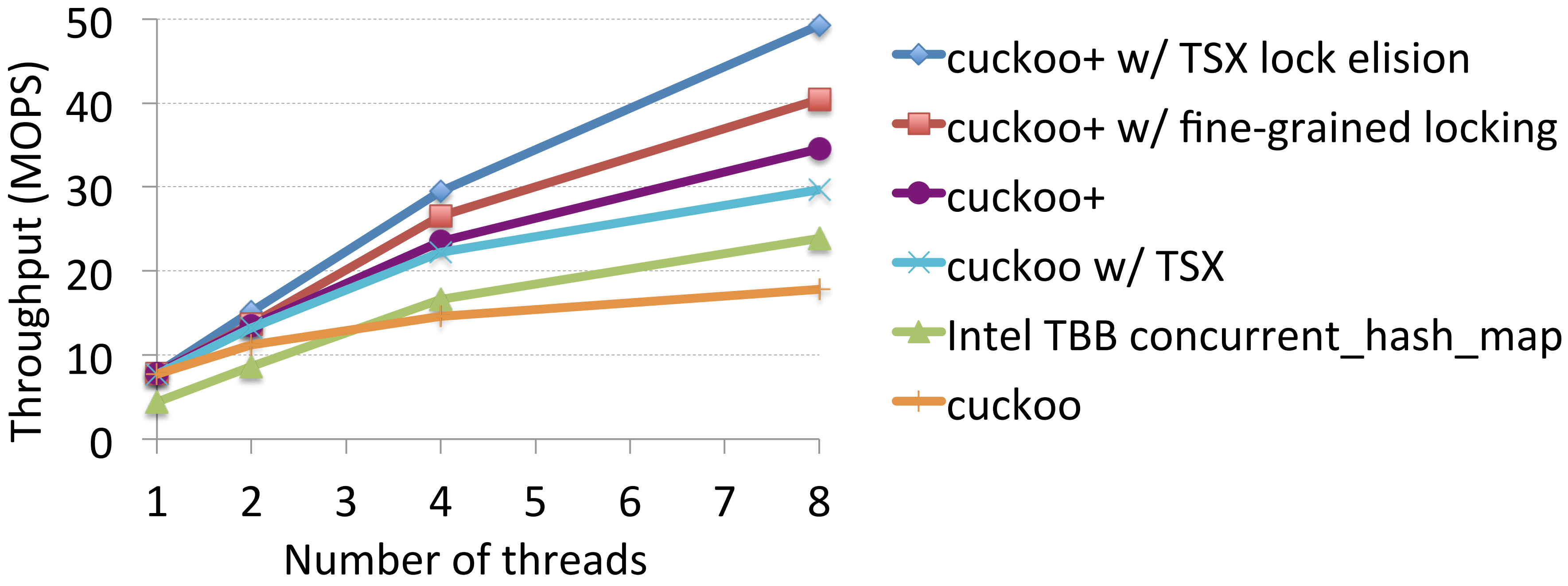
- Platform
  - Intel Haswell i7-4770 @ 3.4GHz (with TSX support)
  - 4 cores (8 hyper-threaded cores)
- Cuckoo hash table
  - 8 byte keys and 8 byte values
  - 2 GB hash table, ~134.2 million entries
  - 8-way set-associative
- Workloads
  - Fill an empty table to 95% capacity
  - Random mixed reads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

Multi-core scaling comparison (50% `Insert`)



cuckoo: single-writer/multi-reader [Fan, NSDI'13]
cuckoo+: cuckoo with our algorithmic optimizations

Multi-core scaling comparison (10% `Insert`)



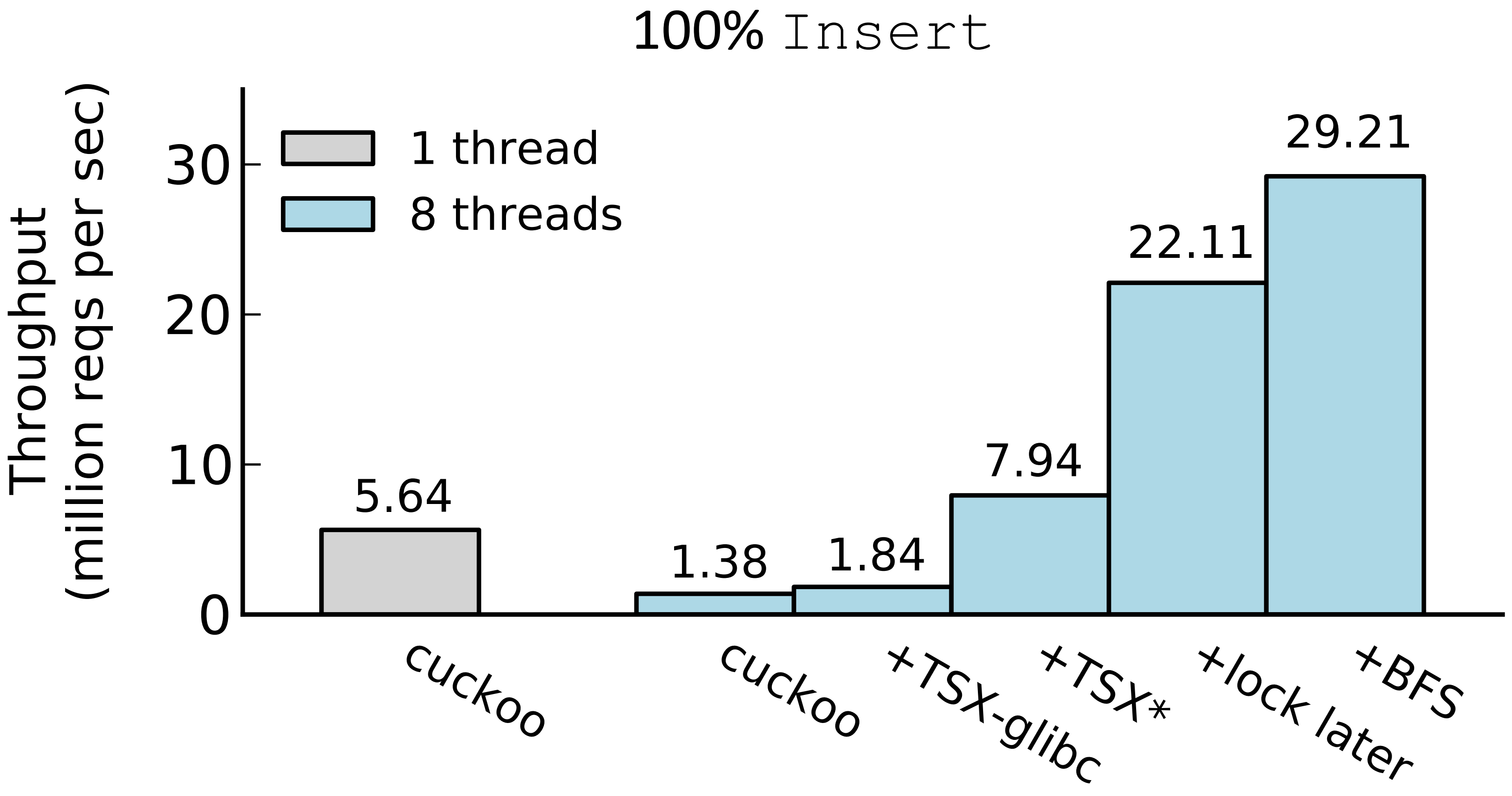
cuckoo: single-writer/multi-reader [Fan, NSDI'13]

cuckoo+: cuckoo with our algorithmic optimizations

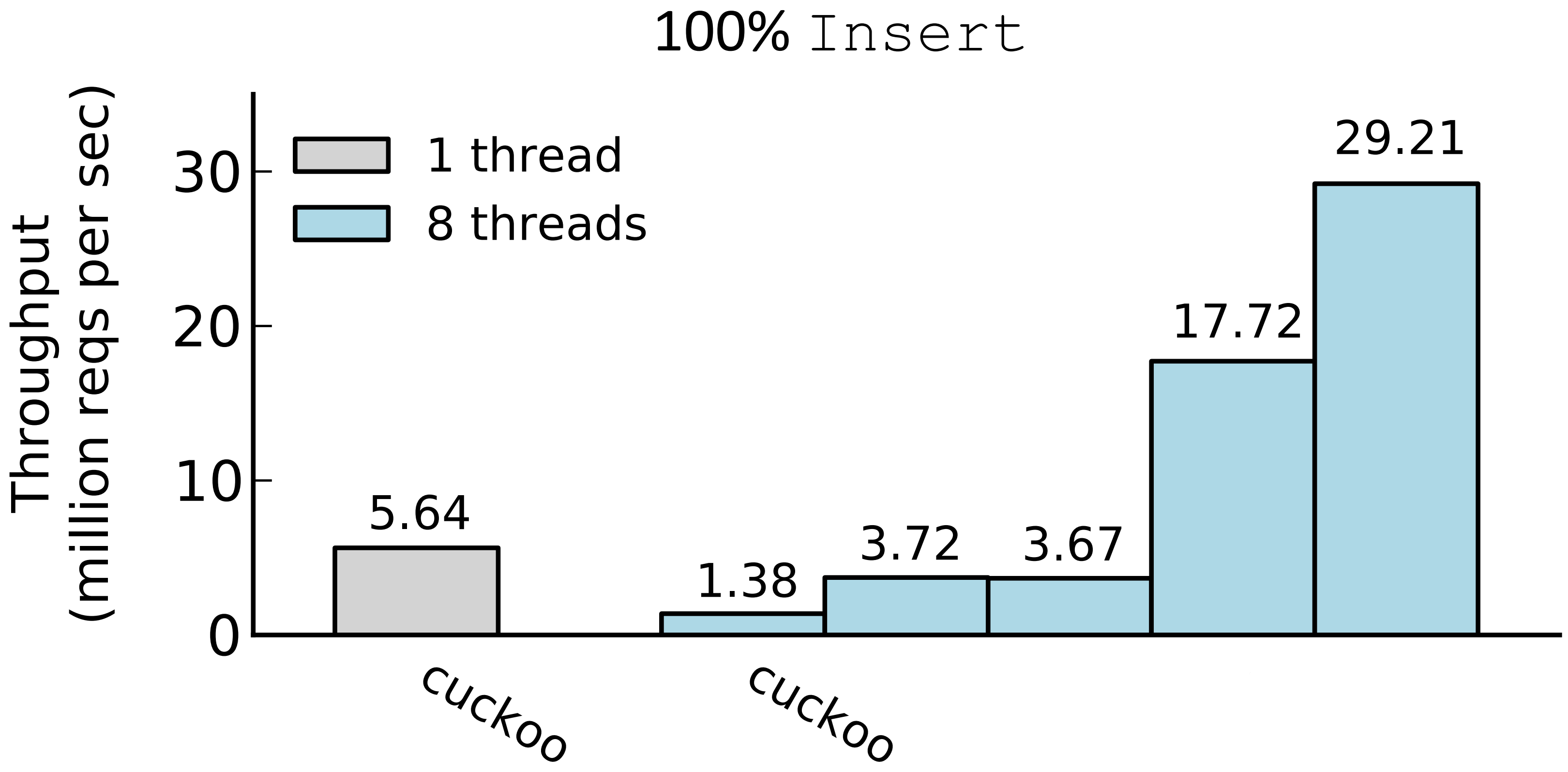
Factor analysis of `Insert` performance

- **cuckoo**: multi-reader single-writer cuckoo hashing [Fan, NSDI'13]
- **+TSX-glibc**: use released Intel glibc TSX lock elision
- **+TSX***: *replace* TSX-glibc with our optimized implementation
- **+lock later**: lock after discovering a cuckoo path
- **+BFS**: breadth first search for an empty slot

Lock elision enabled first and algorithmic optimizations applied later



Algorithmic optimizations applied first and lock elision enabled later



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

Conclusion

- Concurrent cuckoo hash table
  - high memory efficiency
  - fast concurrent writes and reads

- Lessons with hardware transactional memory
  - algorithmic optimizations are necessary

Q & A

Source code available: github.com/efficient/libcuckoo

- fine-grained locking implementation

Thanks!