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

- C++11 std::unordered_map
- Google dense_hash_map
- Intel TBB concurrent_hash_map
- cuckoo+ with fine-grained locking
- cuckoo+ with HTM

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:

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>r</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>s</td>
<td>b</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td>n</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>k</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Both are full?

Insert $y$
Insert may need “cuckoo move”

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

Insert y

```
  0       
  1       r
  2       e
  3       b       s
  4       c       f
  5       a       n
  6                       
  7       k       x
  8                       
  
possible locations
```
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 $\Rightarrow$ undirected cuckoo graph

bucket $\rightarrow$ vertex
key $\rightarrow$ edge
Previous approach to search for an empty slot: **random walk** on the cuckoo graph

**Insert** $y \rightarrow a *$

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>e</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>s</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>k</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>f</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>t</td>
<td>*</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>*</td>
<td>$\emptyset$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**cuckoo path:**

$a \rightarrow e \rightarrow s \rightarrow x \rightarrow k \rightarrow f \rightarrow d \rightarrow t \rightarrow \emptyset$

**9 writes**

*One Insert may move at most hundreds of items when table occupancy > 90%*
Breadth-first search for an empty slot

Insert $y \to a *$

```
* e
* s
x *
* k
* f
* d
* t
* Ø
```

Insert $y \to a *$

```
z *
* *
* * u
* *
* * Ø
```
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

---

- abort
- success
- execute
- retry
- fallback

---

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

---

optimized to make better decisions
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**

<table>
<thead>
<tr>
<th>previous cuckoo[^Fan, NSDI’13]</th>
<th>optimized cuckoo</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuckoo search: 500 reads</td>
<td>—</td>
</tr>
<tr>
<td>cuckoo move: 250 writes</td>
<td>cuckoo move: 5 writes/reads</td>
</tr>
</tbody>
</table>
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)

Throughput (MOPS) vs Number of threads

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

Throughput (million reqs per sec)

- cuckoo
- cuckoo + TSX-glibc
- +TSX*
- +lock later
- +BFS
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!