ONE DOES NOT SIMPLY
WALK INTO A CONCURRENT HASH TABLE
Catching up with Cuckoo

Bin Fan, Xiaozhou Li, Michael Kaminsky, Mike Freedman, David G. Andersen
CMU, Intel, Princeton
Hashing - it’s useful!

CS 101 version

• map[“dog”] = 5

• print map[“dog”] —> 5
Hashing - it’s fun!

CS 201 version

• O(1) insert / lookup / delete
• Linear probing
• Chaining

Standard methods: Either slow, non-concurrent, or waste memory
Hashing - it’s cool again!

Grad school

• Cuckoo Hashing
  • Seriously memory efficient
  • But before our work, slow in practice, non-concurrent (or inefficient)
The Cuckoo Filter

Prior Work
- Basic Cuckoo
- 2,4 associative cuckoo

Building Block #1
- Partial-Key Cuckoo

Building Block #2
- “Move the Hole” Cuckoo

Awesome New Toys
- Optimistic Multi-Reader Cuckoo
- Concurrent Multi-Writer Cuckoo
- New Toys
Cuckoo Hashing

• Hash item to two possible buckets
  H1(key) —> bucket 1
  H2(key) —> bucket 2

What if keys are not stored in table?
Expensive Key Retrieval

- Why not store key-value in table?
  - support variable-len keys
  - to store key-value in external storage

- Lookup requires multiple retrievals for key comparison
Partial-key Cuckoo Hashing

– Definition: a tag
  – a small hash value, 1 Byte in our implementation
  – $\text{tag(“foo”) } = 0x3f$

• Store tags in table to reduce false retrievals
  – Read K-V only on tag match
Cuckoo Move without *Pointer

use current location to compute alternate

\[
\begin{align*}
b_1 &= \text{HASH}(\text{tag}(x)) \quad \text{// 1st bucket} \\
b_2 &= b_1 \oplus \text{HASH}(\text{tag}(x)) \quad \text{// 2nd bucket} \\
al \ &= \text{cur} \oplus \text{HASH}(\text{tag}(x))
\end{align*}
\]
Building Block #1: Partial-key cuckoo hashing

Benefits:

✓ Compact, fixed-sized fields in hash table
✓ Only $1+\varepsilon$ pointer dereferences for lookup
✓ No pointer dereference needed for cuckooing

Hey, Dave - haven’t you heard of multicore?
Prior Work
- Basic Cuckoo
- 2,4 associative cuckoo

Building Block #1
- Partial-Key Cuckoo

Building Block #2
- “Move the Hole” Cuckoo

Awesome New Toys
- The Cuckoo Filter
- Optimistic Multi-Reader Cuckoo
- Concurrent Multi-Writer Cuckoo
Why “Move The Hole”?

- During insertion...
  - one key is always “floating”
  - That key cannot be found by get()

- Prior solutions caused previous concurrent cuckoo tables to waste space [Herlihy]
Move The Hole: Find Path First

Then move hole backwards

✓ Items never disappear
✓ Only individual swaps must be atomic
Full Tables: Lots of Motion

Up to 500 moves needed at 95% occupancy!

Large potential for concurrency conflicts
Optimization Strategy

• Move work outside lock (done: search first)

• Reduce number of moves needed??
Breadth-first search for hole instead of depth-first

Same *search* work
Less *move* work:

~500 bins examined
~5 bins moved

Effective for locking, flash, NVRAM, ...
Prior Work
- Basic Cuckoo
- 2,4 associative cuckoo

Building Block #1
- Partial-Key Cuckoo

Building Block #2
- “Move the Hole” Cuckoo

Awesome New Toys
- The Cuckoo Filter
- Optimistic Multi-Reader Cuckoo
- Concurrent Multi-Writer Cuckoo
Historical Reminder

MemC3 nsdi2013
4.3M ops/sec over the network

Single-writer;
optimized for ~95% reads (Facebook)
Hash Table Microbenchmark

Low Power Xeon CPU (12 cores), 12 MB L3 cache
6 threads reading a ~ 1GB hash table

![Bar chart showing performance comparison of different hash table implementations: Base, Chaining w/ Bkt Lock, and Optimistic Cuckoo. The chart shows Million Lookups/sec with values 1.74, 12.79, and 21.54 respectively.]
Basic Cuckoo

Prior Work

Basic Cuckoo

2,4 associative cuckoo

Partial-Key Cuckoo

Building Block #1

Partial-Key Cuckoo

Building Block #2

“Move the Hole” Cuckoo

Awesome New Toys

The Cuckoo Filter

Optimistic Multi-Reader Cuckoo

Concurrent Multi-Writer Cuckoo
Interlude: Hardware Transactional Memory

- We made two versions of Concurrent Cuckoo:
  - Fine-grained conventional spinlocks
  - One that used Intel's new Hardware Transactional Memory (TSX)
- They perform similarly. I’ll show the TSX results.
Without Concurrency Optimization

Total throughput drops with more threads
Optimizations: 8 thread throughput

Baseline: Optimistic Cuckoo with DFS + TSX-glibc + Our TSX lib + search first, lock later
+ BFS

4 core Haswell Desktop
vs. The Competition
Concurrent beats non-concurrent

- Google dense_hash_map: 64 bit key/value pairs
- C++11 std:unordered_map: read-to-write ratio = 1:1
- Optimistic concurrent cuckoo: 120 million keys inserted
- Intel TBB concurrent_hash_map
- (*) cuckoo+ with fine-grained locking
- (*) cuckoo+ with HTM

Throughput (million reqs per sec)
Concurrent Cuckoo

A really tasty memory-efficient, concurrent hash table
Prior Work

Basic Cuckoo

2,4 associative cuckoo

Building Block #1

Awesome New Toys

The Cuckoo Filter

Optimistic Multi-Reader Cuckoo

Concurrent Multi-Writer Cuckoo

Ongoing work:
Intel DPDK + alternate hash designs =

70 million key/value ops/sec
over the network
Prior Work
- Basic Cuckoo
- 2,4 associative cuckoo
- Intel DPDK

Building Block #1
- Partial-Key Cuckoo

Building Block #2
- "Move the Hole" Cuckoo

Awesome New Toys
- The Cuckoo Filter
- Optimistic Multi-Reader Cuckoo
- Concurrent Multi-Writer Cuckoo