Lock-Free Cuckoo Hashing

By Ross Coker and
Calvin Giroud

Why do we care?

- Hash tables are a prevalent data-structure with widespread use cases
- Specifically, fast general purpose concurrent hash tables can help speed up algorithms:
  - Transposition tables in chess engines
  - Unique tables in generating binary decision diagrams
  - Generalizes to any search problem involving memoization

Cuckoo hashing gives us worst-case guarantees

- A hashing scheme in which two hash functions are used
    - Keys can exist at two possible indices: **hash1(x), hash2(x),** each in a separate table.
- **Search:**
    - Look at the two possible indices
    - Worst case constant time
- **Insert:**
    - Insert in one of the two possible indices
        - If both are taken, evict a key from its slot, thus triggering a chain of relocations
    - Expected constant time

A possible “cuckoo path”

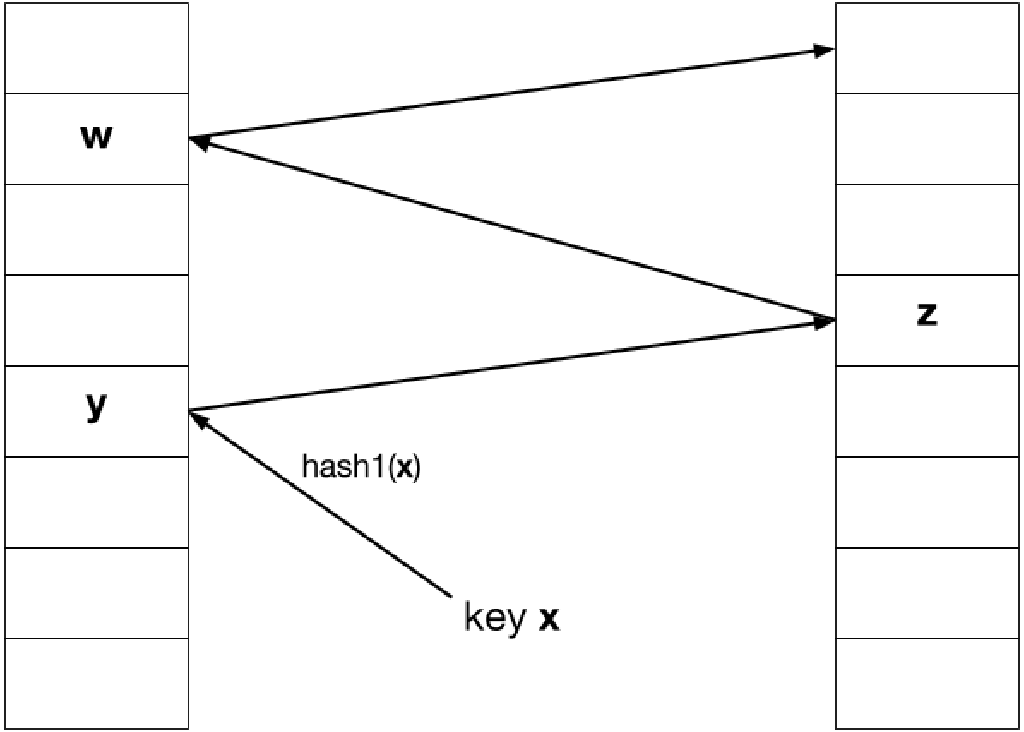


Figure 1. An example “cuckoo path”.

Lock-freedom is optimal for our purposes

- Lock-free solutions better than locking solutions when:
    - Contention is high
    - Machine architecture cannot be optimized against
    - Suitable for general purpose implementations
- However, they are difficult to get right:
    - ABA problem
    - Memory reclamation
    - Tricky to guarantee correctness

Issue: Moving Keys

- Consider a **search(x)** operation, where key **x** is in the table. A possible result of a naive implementation is:
  - Look at index **hash1(x)**. Key is not there
  - Key **x** at index **hash2(x)** is relocated to index **hash1(x)**
  - Look at index **hash2(x)**. Key is not there. Return “key not found”
- Solution: A two-round querying solution with version counters to keep track of number of relocations

Issue: Floating Keys

- When relocating a key, a naive implementation might evict a key, which will a trigger a set of evictions
- Problematic in concurrent environments:
  - From the time the key has been evicted to until it evicts another key, it is inaccessible to other operations
- Solution: Separate cuckoo path discovery and eviction.
  - Step 1: Discover the cuckoo path (i.e. find an empty slot)
  - Step 2: Relocate empty slot backwards along the cuckoo path
  - The “floating key” is now the empty slot, which does not affect correctness

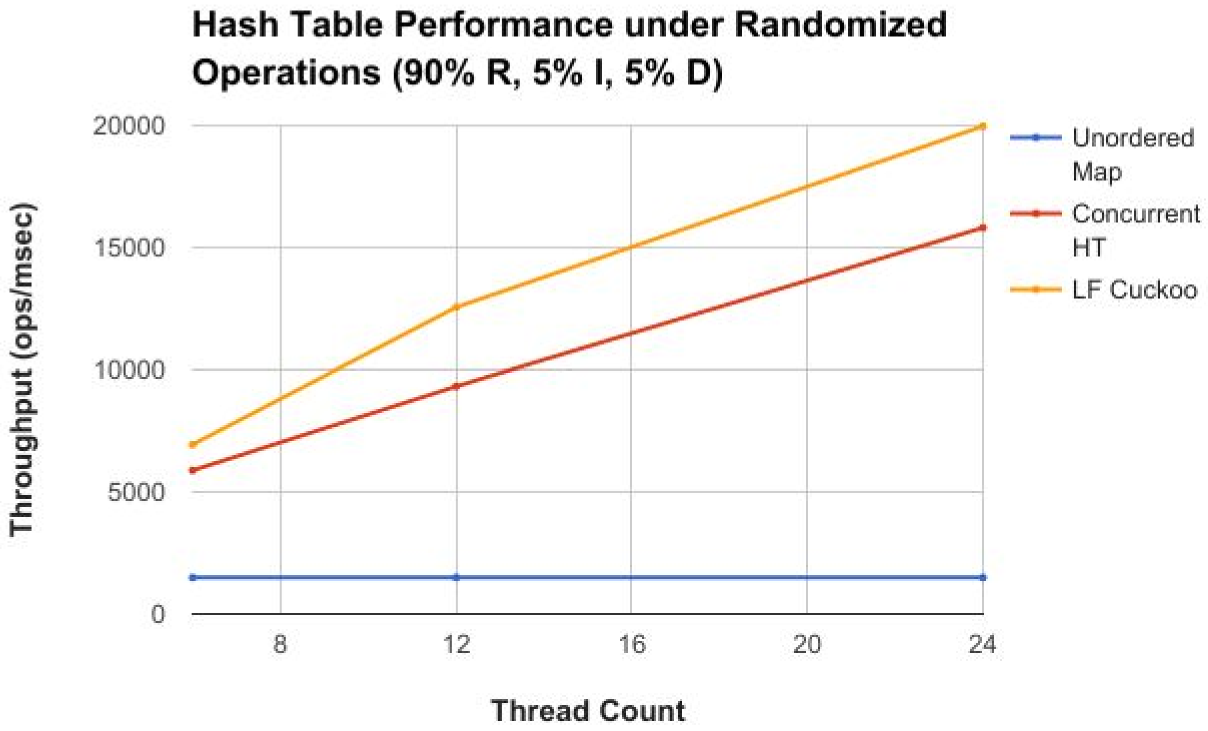
Issue: Memory Reclamation

- Lock-free data structures operate under the assumption that the local copy of data read cannot be invalidated
    - This is not true when dealing with dynamically allocated memory
- Example:
    - Thread 1 looks up a key in the table, getting back a pointer
    - Thread 2 removes the same key from the table, and frees the pointer
    - Thread 1 could make an illegal memory access
- Solution: Hazard pointers
    - Safe memory reclamation technique
        - Threads will announce their intention of holding hazardous memory references
        - Memory references are only freed if no one else is holding it

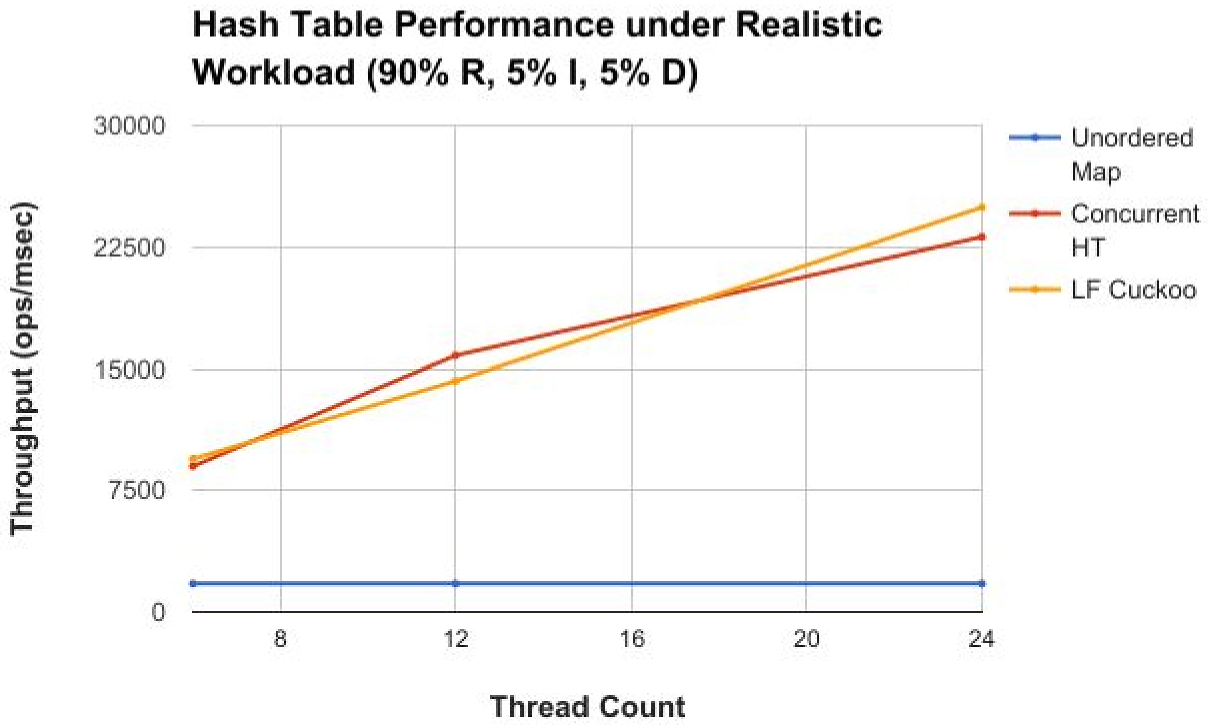
Test Scenario

- Latedays Cluster
  - Two six-core Intel Xeon E5-2620 processors
- 10,000,000 operations
- Benchmarking against:
  - C++ unordered_map
  - Intel's concurrent_hash_map
- Measuring throughput (ops/msec)

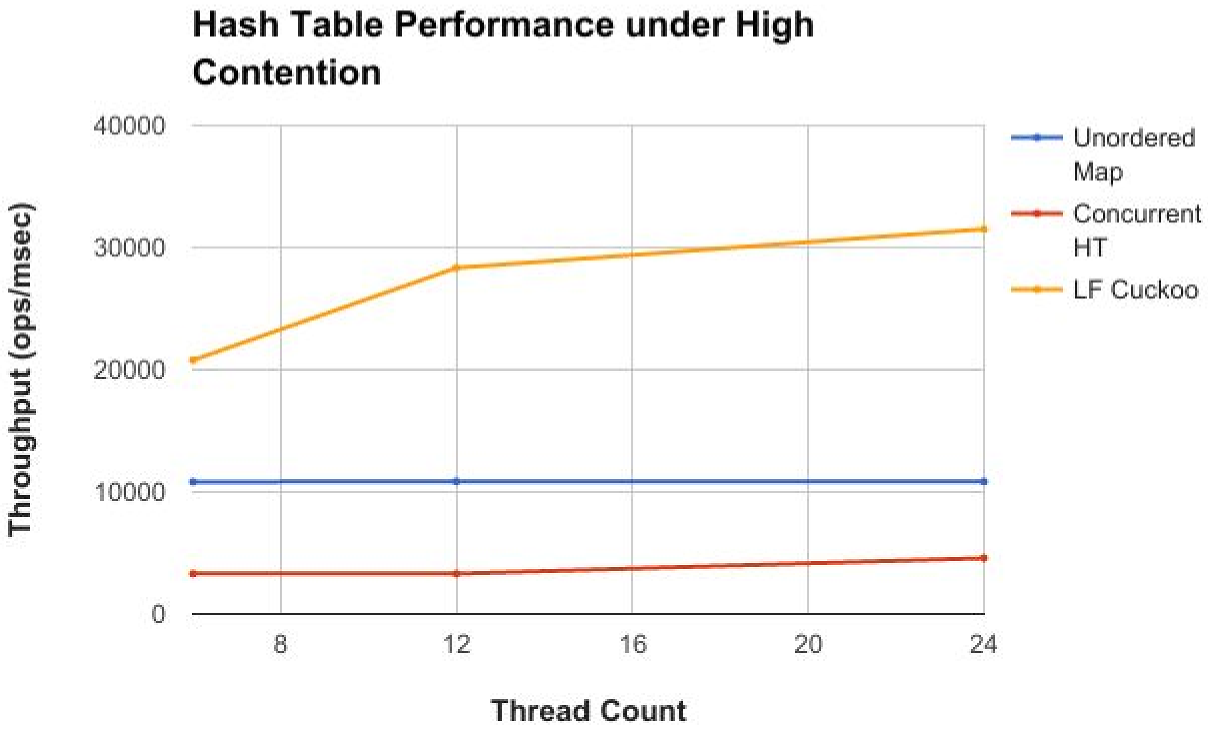
13x speedup over an unordered_map



Still 13x speedup under more realistic test



LF Cuckoo performs well under high contention



Questions?