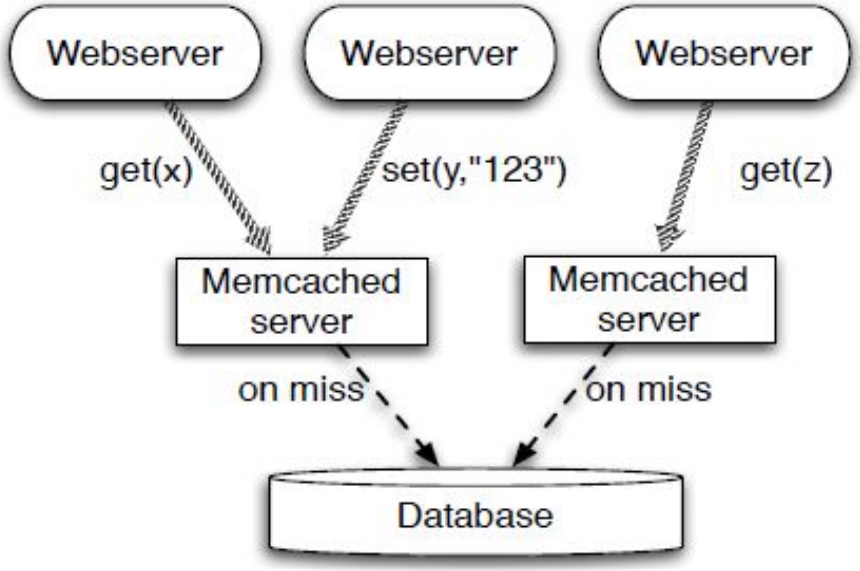
MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing

NSDI2013

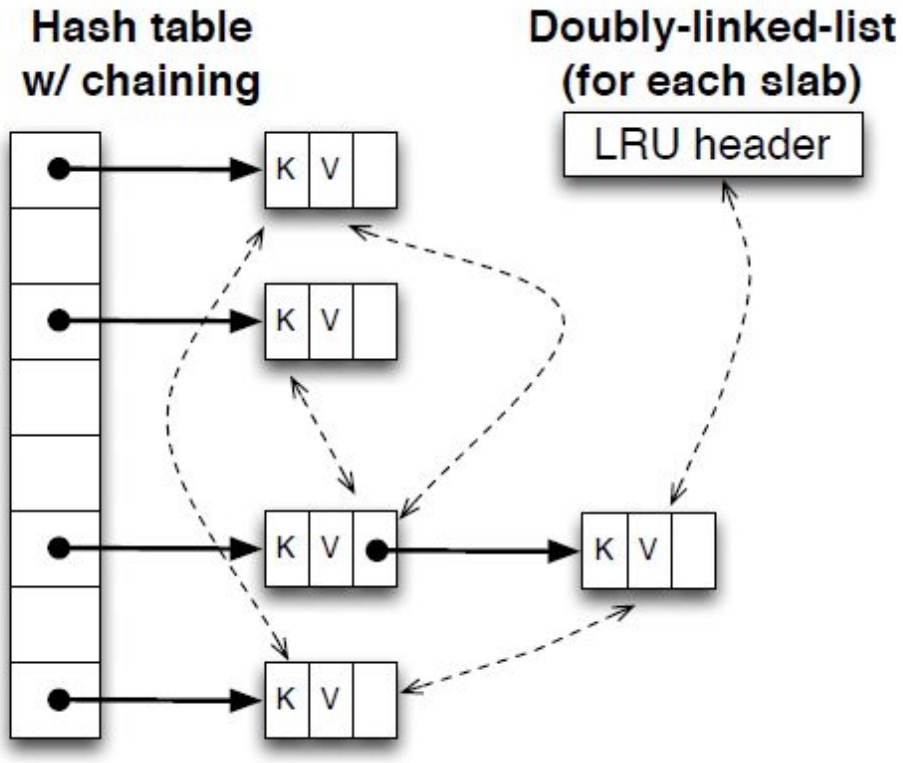
About MemCached

- MemCached is an open-source in-memory key-value store for small chunks of data which may be produced by database call, API call or page rendering call.
- Client uses simple API to retrive data object from Memcached server:
  - SET(key, value): add a (key, value) object to cache.
  - GET(key): retrieve the value associated with a key.
  - DELETE(key): delete a key.
- Memcached server returns the object to the client on a cache hit and queries the database on a cache miss.



About MemCached

- Chain based hashing.
- Use Slab Allocator to manage allocated chunks to avoid memory fragmentation.
- Use Least Recently Used (LRU) based cache swapping policy.



About MemCached

- Core data structures are protected by gloabal locks.
- This poor design prevents MemCached from scaling up on multi-core machine.
- Previous work focuses on sharding in-memory data to different cores to eliminate thread synchronization.
- This kind of approach can not deal with skewed workloads.

Real World Worload

- The key-value workload of facebook indicates that:

- Queries for small objects dominate.
  - A common type of request uses 16 or 21 byte keys and 2bytes values. But a 56-byte header for each key-value pair.
  - High memory overhead.

- Queries are read heavy.
  - GET/SET ratio is 30:1 reported by facebook.
  - A GET operation acquires too many locks in MemCached.
    - a global lock for exclusively access the key.
    - a global lock for exclusively access the hash table and updating the LRU list.
  - Need a multiple-reader/single writter locking structure.

MemC3

- MemC3 is designed to deal with the mentioned problem of MemCached.
- Concurrent Cuckoo Hashing.
- CLOCK based cache management.

Cuckoo Hashing

- Use two hash functions T_1 and T_2 ,each associated with a hash table.
- Every key is hashed by either of the two functions, and inserted in the associated hash table.
- Lookup function is simply:

```
function lookup(x)
    return T1[h1(x)] = x ∨ T2[h2(x)] = x
end
```

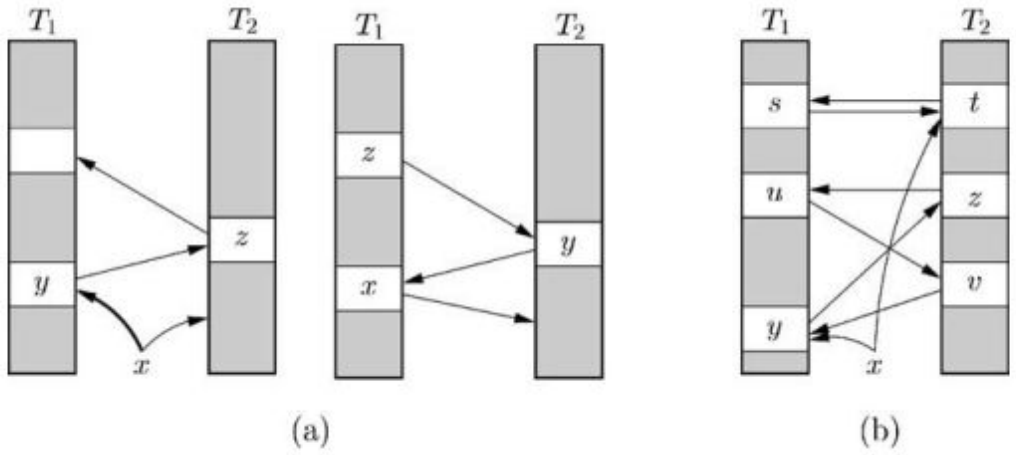
- Deletion function looks up the key in the two hash table and deletes that key.

Cuckoo Hashing

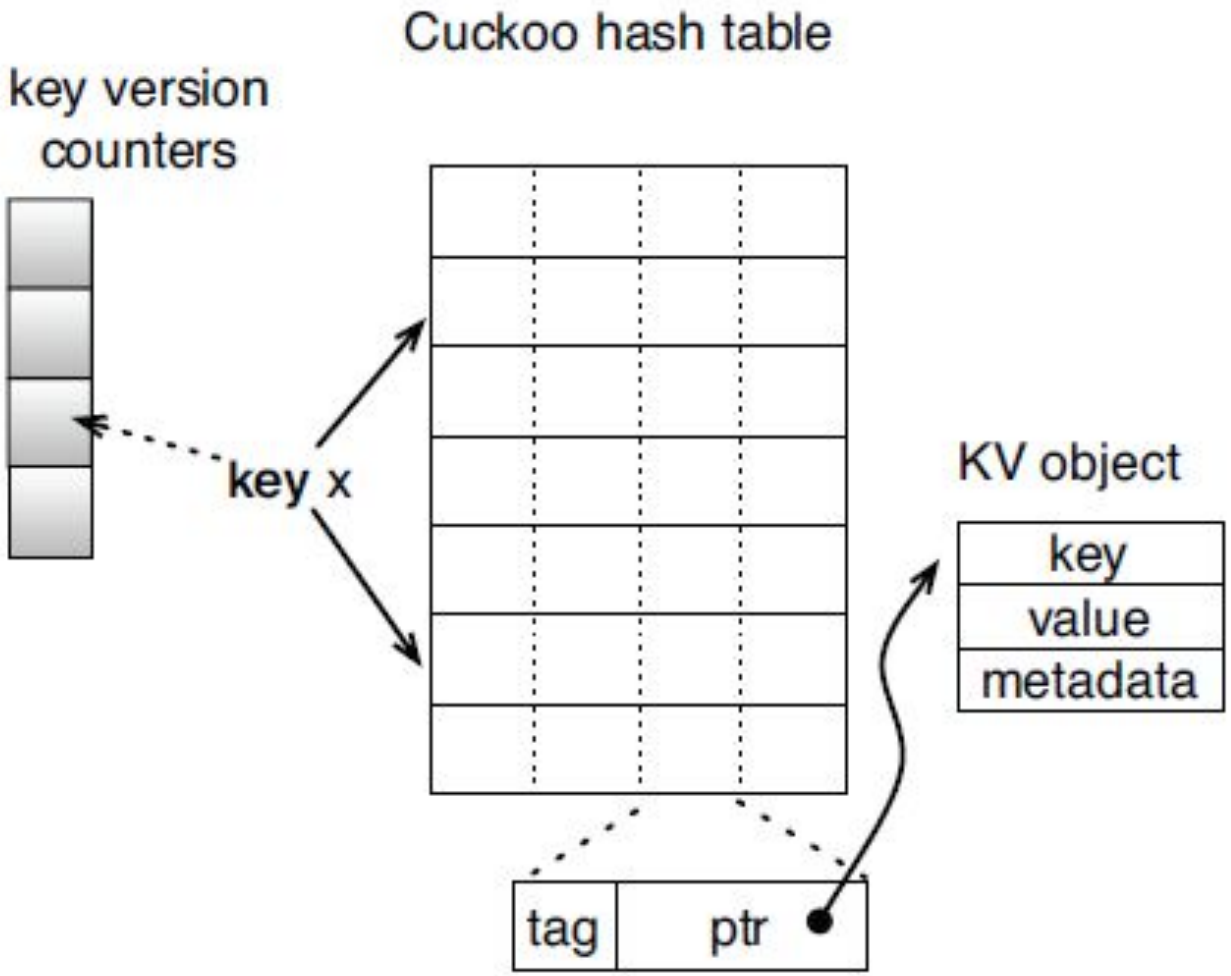
- Insertion:
  - If the slot in hash table T_1 or T_2 is empty, insert the current object a.
  - If the slot is occupied by object b in T_1, kick the object to T_2, and insert object a in the unoccupied slot in T_1.
  - Repeat the above process for object b.

- Insertion may encounter infinite loop because of the hash collision.
- Set a maixmum loop number, if reached then use new hash functions and rehash all the existing keys.

```
procedure insert(x)
    if lookup(x) then return
    loop MaxLoop times
        x ↔ T1[h1(x)]
        if x = ⊥ then return
        x ↔ T2[h2(x)]
        if x = ⊥ then return
    end loop
    rehash(); insert(x)
end
```



Cache-friendly and Concurrent Cuckoo Hashing

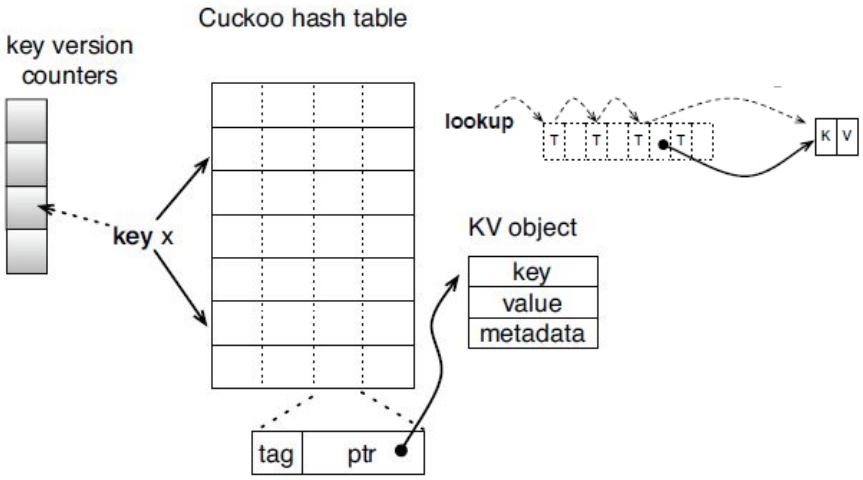


Cache Friendly Lookup/Insert

- Naive approach is not cache-friendly
  - up to 8 pointer dereference for each lookup.

- Tag is another one byte hashing of the key, acting as a summary for the original key. $tag = HASH_2(x)$

- Each bucket fits into the cache line.



Cache Friendly Lookup/Insert

- Naive approach is not cache-friendly
  - multiple pointer dereference for each insert.

- How the key is mapped to two buckets indirectly:

$$b_1 = HASH_1(x)$$
$$b_2 = b_1 \oplus HASH_1(tag) = b_1 \oplus HASH_1(HASH_2(x))$$

- To relocate an occupied key in the bucket, the alternate bucket for that key can be directly calculated as:

$$b' = b \oplus HASH_1(tag)$$

- Insert operations can operate using only information in the table and never have to retrieve keys.

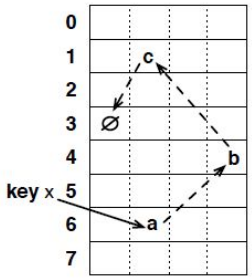


Figure 3: Cuckoo path. ∅ represents an empty slot.

Concurrent Cuckoo Hashing

- Making accessing the Cuckoo hash table concurrently for multiple threads.

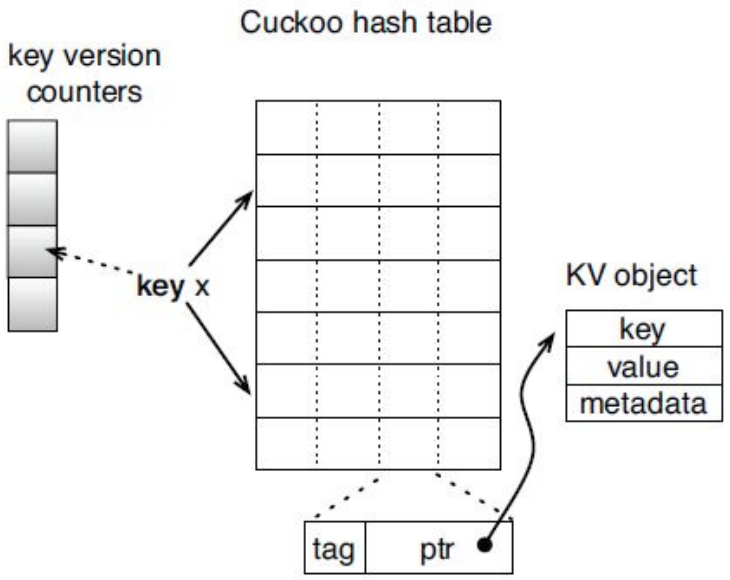
- Allow multiple readers or a single writter, which is a tradeoff considering the real workload characteristics.

- Basic optimization idea:
  - Separate discovering a valid Cuckoo path from the execution of this path.
  - Move keys backwards along the Cuckoo path.
  - Keep a version counter for each key.

Concurrent Cuckoo Hashing

- An arrary of version counters, shared by multiple keys with the same hashing result. (This keeps the size of the array small)

- May "false retry" a given key due to the modification of other keys. But the very rare.



Concurrent Cuckoo Hashing

- The insert will first find the Cuckoo path.
- Then displace the key from backward.

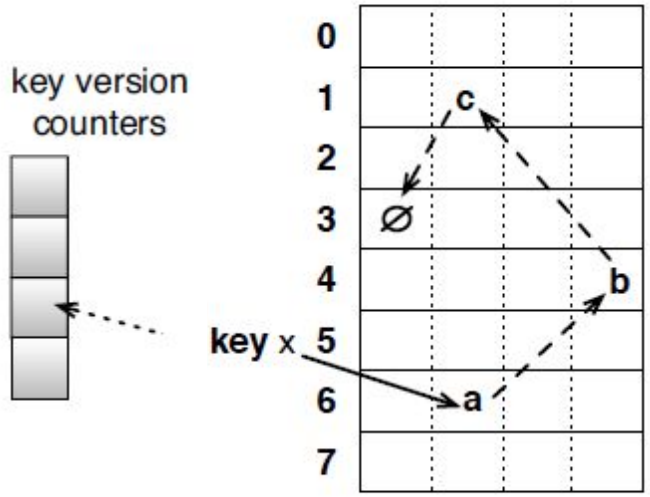


Figure 3: Cuckoo path. ∅ represents an empty slot.

Concurrent Cuckoo Hashing

- Remember that multiple read and a single write are concurrent.

- Before the write(Insert) displacing a key, increase the counter associated with that key by 1. When the displacing of the key is finished, increase that counter by 1 again.

- When the reader(Lookup) tries to read a key, it first keeps the snapshot of the version number associated with that key. If it's odd, wait and retry. If it's even, process with the read. After the read, check the version number again. If it's different with the snapshot, retry.

Concurrent Cache Management

- After each read, update is executed to keep track of the data recency.

- When inserting a new key value pair, old key value pair may be evicted due to a full cache.

- CLOCK based cache management:
  - Keep a ring buffer for each slab class, use one bit to represent each key value pair.
  - Each update set the corresponding bit to 1 and set the virtual hand to ponit to that bit.
  - For eviction, if the bit pointed to by the hand is 0, then evict the corresponding key value pair. Otherwise, set the bit to 0 and advance the virtual hand until it reaches an 0.

Concurrent Cache Management

- Evict uses version counters to ensure consistency:
  - Increase the version counter by 1, this causes all the read to keep retrying.
  - Delete the key value pair and increase the version counter by 1 again.
  - The read will return with a fail.

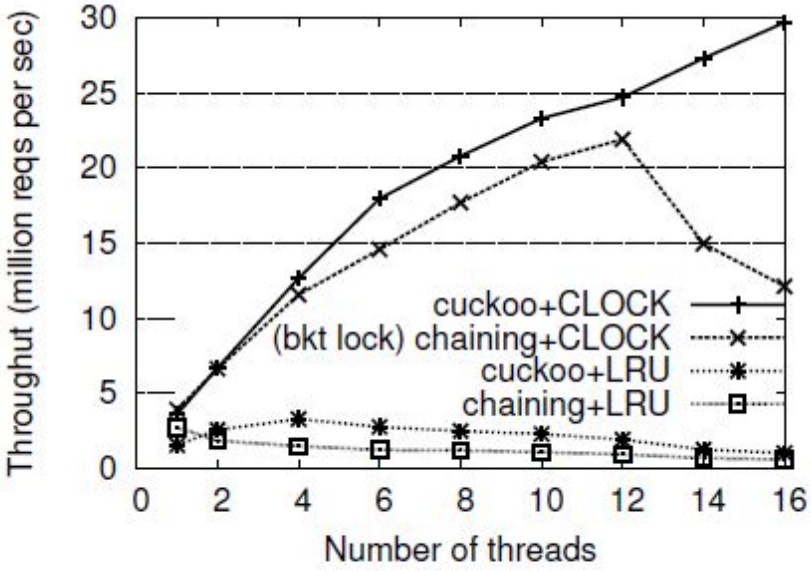
Concurrent Cuckoo Hashing

Algorithm 1: Psuedo code of `SET` and `GET`

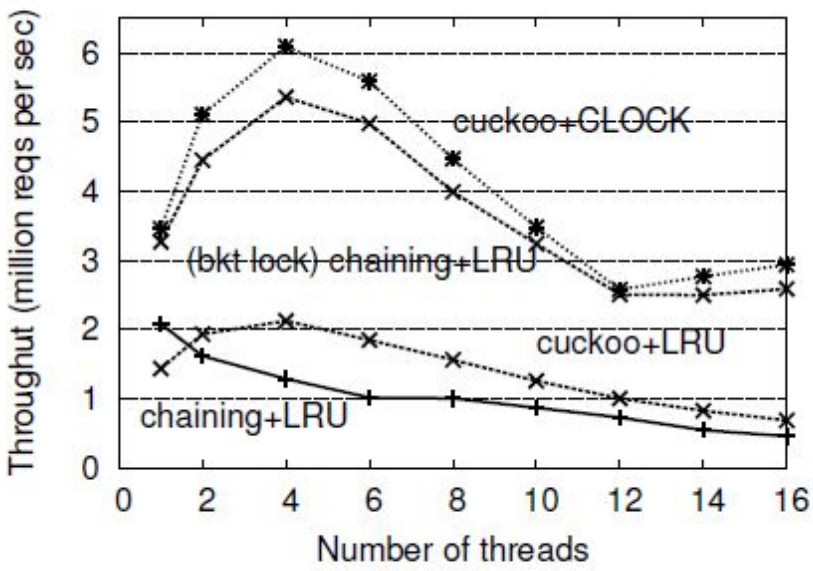
```
SET (key, value)    //insert (key,value) to cache
begin
    lock();
    ptr = Alloc();        //try to allocate space
    if ptr == NULL then
        ptr = Evict();    //cache is full, evict old item
    memcpy key, value to ptr;
    Insert (key, ptr) ; //index this key in hashtable
    unlock();

GET (key)    //get value of key from cache
begin
    while true do
        vs = ReadCounter (key) ;   //key version
        ptr= Lookup (key) ;        //check hash table
        if ptr == NULL then  return NULL ;
        prepare response for data in ptr;
        ve = ReadCounter (key) ;   //key version
        if vs & 1 or vs != ve then
            //may read dirty data, try again
            continue
        Update (key) ;             //update CLOCK
        return response
```

Performance Evaluation



(a) 10GB "big cache" (> working set): 100% GETs hit



(b) 1GB "small cache" (< working set): 85% GETs hit