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

Bin Fan, David G. Andersen, Michael Kaminsky*
Carnegie Mellon University, *Intel Labs

Abstract

This paper presents a set of architecturally and workload-inspired algorithmic and engineering improvements to the popular Memcached system that substantially improve both its memory efficiency and throughput. These techniques—optimistic cuckoo hashing, a compact LRU-approximating eviction algorithm based upon CLOCK, and comprehensive implementation of optimistic locking—enable the resulting system to use 30% less memory for small key-value pairs, and serve up to 3x as many queries per second over the network. We have implemented these modifications in a system we call MemC3—Memcached with CLOCK and Concurrent Cuckoo hashing—but believe that they also apply more generally to many of today's read-intensive, highly concurrent networked storage and caching systems.

1 Introduction

Low-latency access to data has become critical for many Internet services in recent years. This requirement has led many system designers to serve all or most of certain data sets from main memory—using the memory either as their primary store [19, 26, 21, 25] or as a cache to deflect hot or particularly latency-sensitive items [10].

Two important metrics in evaluating these systems are performance (throughput, measured in queries served per second) and memory efficiency (measured by the overhead required to store an item). Memory consumption is important because it directly affects the number of items that system can store, and the hardware cost to do so.

This paper demonstrates that careful attention to algorithm and data structure design can significantly improve throughput and memory efficiency for in-memory data stores. We show that traditional approaches often fail to leverage the target system's architecture and expected workload. As a case study, we focus on Memcached [19], a popular in-memory caching layer, and show how our toolbox of techniques can improve Memcached's performance by 3× and reduce its memory use by 30%.

Standard Memcached, at its core, uses a typical hash table design, with linked-list-based chaining to handle collisions. Its cache replacement algorithm is strict LRU, also based on linked lists. This design relies on locking to ensure consistency among multiple threads, and leads to poor scalability on multi-core CPUs [11].

This paper presents MemC3 (**Mem**cached with **C**LOCK and **C**oncurrent **C**uckoo Hashing), a complete redesign of the Memcached internals. This re-design is informed by and takes advantage of several observations. First, architectural features can hide memory access latencies and provide performance improvements. In particular, our new hash table design exploits CPU cache locality to minimize the number of memory fetches required to complete any given operation; and it exploits instruction-level and memory-level parallelism to overlap those fetches when they cannot be avoided.

Second, MemC3's design also leverages workload characteristics. Many Memcached workloads are predominately reads, with few writes. This observation means that we can replace Memcached's exclusive, global locking with an optimistic locking scheme targeted at the common case. Furthermore, many important Memcached workloads target very small objects, so per-object overheads have a significant impact on memory efficiency. For example, Memcached's strict LRU cache replacement requires significant metadata—often more space than the object itself occupies; in MemC3, we instead use a compact CLOCK-based approximation.

The specific contributions of this paper include:

- A novel hashing scheme called *optimistic cuckoo hashing*. Conventional cuckoo hashing [23] achieves space efficiency, but is unfriendly for concurrent operations. Optimistic cuckoo hashing (1) achieves high memory efficiency (e.g., 95% table occupancy); (2) allows multiple readers and a single writer to concurrently access the hash table; and (3) keeps hash table operations cache-friendly (Section 3).
- A compact CLOCK-based eviction algorithm that requires only 1 bit of extra space per cache entry and supports concurrent cache operations (Section 4).
- Optimistic locking that eliminates inter-thread syn-

function	stock Memcached	MemC3
Hash Table		
concurrency	serialized	concurrent lookup, serialized insert
lookup performance	slower	faster
insert performance	faster	slower
space	13.3n Bytes	~ 9.7n Bytes
Cache Mgmt		
concurrency	serialized	concurrent update, serialized eviction
space	18n Bytes	n bits

Table 1: Comparison of operations. *n* is the number of existing key-value items.

chronization while ensuring consistency. The optimistic cuckoo hash table operations (lookup/insert) and the LRU cache eviction operations both use this locking scheme for high-performance access to shared data structures (Section 4).

Finally, we implement and evaluate MemC3, a networked, in-memory key-value cache, based on Memcached-1.4.13.[1] Table 1 compares MemC3 and stock Memcached. MemC3 provides higher throughput using significantly less memory and computation as we will demonstrate in the remainder of this paper.

2 Background

2.1 Memcached Overview

Interface Memcached implements a simple and lightweight key-value interface where all key-value tuples are stored in and served from DRAM. Clients communicate with the Memcached servers over the network using the following commands:

- `SET/ADD/REPLACE(key, value)`: add a (key, value) object to the cache;
- `GET(key)`: retrieve the value associated with a key;
- `DELETE(key)`: delete a key.

Internally, Memcached uses a hash table to index the key-value entries. These entries are also in a linked list sorted by their most recent access time. The least recently used (LRU) entry is evicted and replaced by a newly inserted entry when the cache is full.

Hash Table To lookup keys quickly, the location of each key-value entry is stored in a hash table. Hash collisions are resolved by chaining: if more than one key maps into the same hash table bucket, they form a linked list.

[1]Our prototype does not yet provide the full memcached api.

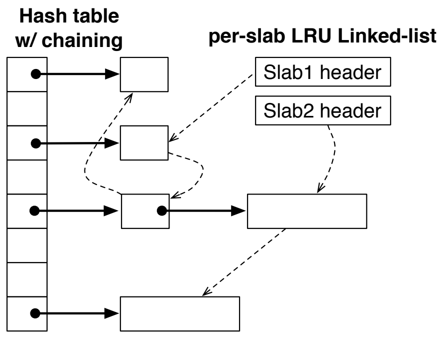


Figure 1: Memcached data structures.

Chaining is efficient for inserting or deleting single keys. However, lookup may require scanning the entire chain.

Memory Allocation Naive memory allocation (e.g., malloc/free) could result in significant memory fragmentation. To address this problem, Memcached uses *slab-based memory allocation*. Memory is divided into 1 MB pages, and each page is further sub-divided into fixed-length *chunks*. Key-value objects are stored in an appropriately-size chunk. The size of a chunk, and thus the number of chunks per page, depends on the particular slab class. For example, by default the chunk size of slab class 1 is 72 bytes and each page of this class has 14563 chunks; while the chunk size of slab class 43 is 1 MB and thus there is only 1 chunk spanning the whole page.

To insert a new key, Memcached looks up the slab class whose chunk size best fits this key-value object. If a vacant chunk is available, it is assigned to this item; if the search fails, Memcached will execute cache eviction.

Cache policy In Memcached, each slab class maintains its own objects in an LRU queue (see Figure 1). Each access to an object causes that object to move to the head of the queue. Thus, when Memcached needs to evict an object from the cache, it can find the least recently used object at the tail. The queue is implemented as a doubly-linked list, so each object has two pointers.

Threading Memcached was originally single-threaded. It uses libevent for asynchronous network I/O callbacks [24]. Later versions support multi-threading but use global locks to protect the core data structures. As a result, operations such as index lookup/update and cache eviction/update are all serialized. Previous work has shown that this locking prevents current Memcached from scaling up on multi-core CPUs [11].

Performance Enhancement Previous solutions [4, 20, 13] shard the in-memory data to different cores. Sharding eliminates the inter-thread synchronization to permit higher concurrency, but under skewed workloads it may also exhibit imbalanced load across different cores or waste the (expensive) memory capacity. Instead of simply sharding, we explore how to scale performance to many threads that share and access the same memory space; one could then apply sharding to further scale the system.

2.2 Real-world Workloads: Small and Read-only Requests Dominate

Our work is informed by several key-value workload characteristics published recently by Facebook [3].

First, *queries for small objects dominate*. Most keys are smaller than 32 bytes and most values no more than a few hundred bytes. In particular, there is one common type of request that almost exclusively uses 16 or 21 Byte keys and 2 Byte values.

The consequence of storing such small key-value objects is high memory overhead. Memcached always allocates a 56-Byte header (on 64-bit servers) for each key-value object *regardless of the size*. The header includes two pointers for the LRU linked list and one pointer for chaining to form the hash table. For small key-value objects, this space overhead cannot be amortized. Therefore we seek more memory efficient data structures for the index and cache.

Second, *queries are read heavy*. In general, a `GET`/`SET` ratio of 30:1 is reported for the Memcached workloads in Facebook. Important applications that can increase cache size on demand show even higher fractions of `GET`s (e.g., 99.8% are `GET`s, or `GET`/`SET`=500:1). Note that this ratio also depends on the GET hit ratio, because each GET miss is usually followed by a SET to update the cache by the application.

Though most queries are GETs, this operation is not optimized and locks are used extensively on the query path. For example, each GET operation must acquire (1) a lock for exclusive access to this particular key, (2) a global lock for exclusive access to the hash table; and (3) after reading the relevant key-value object, it must again acquire the global lock to update the LRU linked list. We aim to remove all mutexes on the GET path to boost the concurrency of Memcached.

3 Optimistic Concurrent Cuckoo Hashing

In this section, we present a compact, concurrent and cache-aware hashing scheme called *optimistic concurrent cuckoo hashing*. Compared with Memcached's original chaining-based hash table, our design improves memory efficiency by applying cuckoo hashing [23]—a practical, advanced hashing scheme with high memory efficiency and O(1) amortized insertion time and retrieval. However, basic cuckoo hashing does not support concurrent read/write access; it also requires multiple memory references for each insertion or lookup. To overcome these limitations, we propose a collection of new techniques that improve basic cuckoo hashing in concurrency, memory efficiency and cache-friendliness:

- An *optimistic version* of cuckoo hashing that supports multiple-reader/single writer concurrent access, while preserving its space benefits;
- A technique using a short summary of each key to improve the cache locality of hash table operations; and
- An optimization for cuckoo hashing insertion that improves throughput.

As we show in Section 5, combining these techniques creates a hashing scheme that is attractive in practice: its hash table achieves over 90% occupancy (compared to 50% for linear probing, or needing the extra pointers required by chaining) [**?**]. Each lookup requires only two *parallel* cacheline reads followed by (up to) one memory reference on average. In contrast, naive cuckoo hashing requires two parallel cacheline reads followed by (up to) $2N$ parallel memory references if each bucket has N keys; and chaining requires (up to) N *dependent* memory references to scan a bucket of N keys. The hash table supports multiple readers and a single writer, substantially speeding up read-intensive workloads while maintaining equivalent performance for write-heavy workloads.

Interface The hash table provides `Lookup`, `Insert` and `Delete` operations for indexing all key-value objects. On `Lookup`, the hash table returns a pointer to the relevant key-value object, or "does not exist" if the key can not be found. On `Insert`, the hash table returns *true* on success, and *false* to indicate the hash table is too full.[2] `Delete` simply removes the key's entry from the hash table. We focus on `Lookup` and `Insert` as `Delete` is very similar to `Lookup`.

Basic Cuckoo Hashing Before presenting our techniques in detail, we first briefly describe how to perform cuckoo hashing. The basic idea of cuckoo hashing is to use two hash functions instead of one, thus providing each key two possible locations where it can reside. Cuckoo hashing can dynamically relocate existing keys and refine the table to make room for new keys during insertion.

Our hash table, as shown in Figure 2, consists of an array of *buckets*, each having 4 *slots*.[3] Each slot contains a *pointer* to the key-value object and a short summary of

[2] As in other hash table designs, an expansion process can increase the cuckoo hash table size to allow for additional inserts.

[3] Our hash table is 4-way set-associative. Without set-associativity, basic cuckoo hashing allows only 50% of the table entries to be occupied before unresolvable collisions occur. It is possible to improve the space utilization to over 90% by using a 4-way (or higher) set associative hash table. [9]

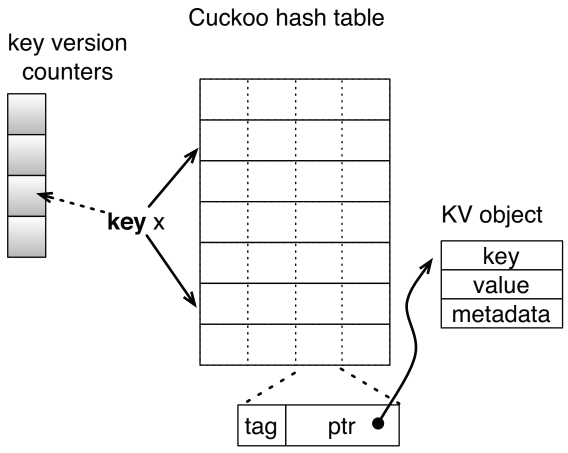


Figure 2: Hash table overview: The hash table is 4-way set-associative. Each key is mapped to 2 buckets by hash functions and associated with 1 version counter; Each slot stores a tag of the key and a pointer to the key-value item. Values in gray are used for optimistic locking and must be accessed atomically.

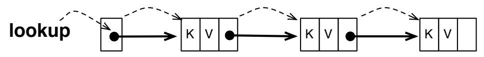
the key called a *tag*. To support keys of variable length, the full keys and values are not stored in the hash table, but stored with the associated metadata outside the table and referenced by the pointer. A null pointer indicates this slot is not used.

Each key is mapped to two random buckets, so `Lookup` checks all 8 candidate keys from every slot. To insert a new key x into the table, if either of the two buckets has an empty slot, it is then inserted in that bucket; if neither bucket has space, `Insert` selects a random key y from one candidate bucket and relocates y to its own alternate location. Displacing y may also require kicking out another existing key z, so this procedure may repeat until a vacant slot is found, or until a maximum number of displacements is reached (e.g., 500 times in our implementation). If no vacant slot found, the hash table is considered too full to insert and an expansion process is scheduled. Though it may execute a sequence of displacements, the amortized insertion time of cuckoo hashing is $O(1)$ [23].

3.1 Tag-based Lookup/Insert

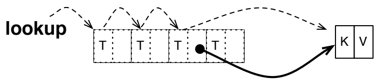
To support keys of variable length and keep the index compact, the actual keys are not stored in the hash table and must be retrieved by following a pointer. We propose a cache-aware technique to perform cuckoo hashing with minimum memory references by using *tags*—a short hash of the keys (one-byte in our implementation). This technique is inspired by "partial-key cuckoo hashing" which we proposed in previous work [17], but eliminates the prior approach's limitation in the maximum table size.

Cache-friendly Lookup The original Memcached lookup is not cache-friendly. It requires multiple *dependent* pointer dereferences to traverse a linked list:



Neither is basic cuckoo hashing cache-friendly: checking two buckets on each `Lookup` makes up to 8 (parallel) pointer dereferences. In addition, displacing each key on `Insert` also requires a pointer dereference to calculate the alternate location to swap, and each `Insert` may perform several displacement operations.

Our hash table eliminates the need for pointer dereferences in the common case. We compute a 1-Byte tag as the summary of each inserted key, and store the tag in the same bucket as its pointer. `Lookup` first compares the tag, then retrieves the full key only if the tag matches. This procedure is as shown below (T represents the tag)



It is possible to have false retrievals due to two different keys having the same tag, so the fetched full key is further verified to ensure it was indeed the correct one. With a 1-Byte tag by hashing, the chance of tag-collision is only $1/2^8 = 0.39\%$. After checking all 8 candidate slots, a negative `Lookup` makes $8 \times 0.39\% = 0.03$ pointer dereferences on average. Because each bucket fits in a CPU cacheline (usually 64-Byte), on average each `Lookup` makes only 2 parallel cacheline-sized reads for checking the two buckets plus either 0.03 pointer dereferences if the `Lookup` misses or 1.03 if it hits.

Cache-friendly Insert We also use the tags to avoid retrieving full keys on `Insert`, which were originally needed to derive the alternate location to displace keys. To this end, our hashing scheme computes the two candidate buckets b_1 and b_2 for key x by

$$\begin{aligned} b_1 &= \text{HASH}(x) && \text{// based on the entire key} \\ b_2 &= b_1 \oplus \text{HASH}(\text{tag}) && \text{// based on } b_1 \text{ and tag of } x \end{aligned}$$

b_2 is still a random variable uniformly distributed[4]; more importantly b_1 can be computed by the same formula from b_2 and tag. This property ensures that to displace a key originally in bucket b—no matter if b is b_1 or b_2— it is possible to calculate its alternate bucket b' from bucket index b and the tag stored in bucket b by

$$b' = b \oplus \text{HASH}(\text{tag}) \qquad (1)$$

As a result, `Insert` operations can operate using only information in the table and never have to retrieve keys.

[4] b_2 is no longer fully independent from b_1. For a 1-Byte tag, there are up to 256 different values of b_2 given a specific b_1. Microbenchmarks in Section 5 show that our algorithm still achieves close-to-optimal load factor, even if b_2 has some dependence on b_1.

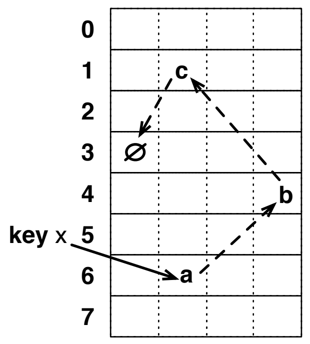


Figure 3: Cuckoo path. ∅ represents an empty slot.

3.2 Concurrent Cuckoo Hashing

Effectively supporting concurrent access to a cuckoo hash table is challenging. A previously proposed scheme improved concurrency by trading space [12]. Our hashing scheme is, to our knowledge, the first approach to support concurrent access (multi-reader/single-writer) while still maintaining the high space efficiency of cuckoo hashing (e.g., > 90% occupancy).

For clarity of presentation, we first define a *cuckoo path* as the sequence of displaced keys in an `Insert` operation. In Figure 3 "$a \Rightarrow b \Rightarrow c$" is one cuckoo path to make one bucket available to insert key x.

There are two major obstacles to making the sequential cuckoo hashing algorithm concurrent:

1. *Deadlock risk (writer/writer):* An `Insert` may modify a set of buckets when moving the keys along the cuckoo path until one key lands in an available bucket. It is not known *before swapping the keys* how many and which buckets will be modified, because each displaced key depends on the one previously kicked out. Standard techniques to make `Insert` atomic and avoid deadlock, such as acquiring all necessary locks in advance, are therefore not obviously applicable.
2. *False misses (reader/writer):* After a key is kicked out of its original bucket but before it is inserted to its alternate location, this key is unreachable from both buckets and temporarily unavailable. If `Insert` is not atomic, a reader may complete a `Lookup` and return a false miss during a key's unavailable time. E.g., in Figure 3, after replacing b with a at bucket 4, but before b relocates to bucket 1, b appears at neither bucket in the table. A reader looking up b at this moment may return negative results.

The only scheme previously proposed for concurrent cuckoo hashing [12] that we know of breaks up `Inserts` into a sequence of atomic displacements rather than locking the entire cuckoo path. It adds extra space at each bucket as an overflow buffer to temporarily host keys swapped from other buckets, and thus avoid kicking out existing keys. Hence, its space overhead (typically two more slots per bucket as buffer) is much higher than the basic cuckoo hashing.

Our scheme instead maintains high memory efficiency and also allows multiple-reader concurrent access to the hash table. To avoid writer/writer deadlocks, it allows only one writer at a time—a tradeoff we accept as our target workloads are read-heavy. To eliminate false misses, our design changes the order of the basic cuckoo hashing insertion by:

1) *separating discovering a valid cuckoo path from the execution of this path.* We first search for a cuckoo path, but do not move keys during this search phase.
2) *moving keys backwards along the cuckoo path.* After a valid cuckoo path is known, we first move the last key on the cuckoo path to the free slot, and then move the second to last key to the empty slot left by the previous one, and so on. As a result, each swap affects only one key at a time, which can always be successfully moved to its new location without any kickout.

Intuitively, the original `Insert` always moves a selected key to its other bucket and kicks out another existing key unless an empty slot is found in that bucket. Hence, there is always a victim key "floating" before `Insert` completes, causing false misses. In contrast, our scheme first discovers a cuckoo path to an empty slot, then propagates this empty slot towards the key for insertion along the path. To illustrate our scheme in Figure 3, we first find a valid cuckoo path "$a \Rightarrow b \Rightarrow c$" for key x without editing any buckets. After the path is known, c is swapped to the empty slot in bucket 3, followed by relocating b to the original slot of c in bucket 1 and so on. Finally, the original slot of a will be available and x can be directly inserted into that slot.

3.2.1 Optimization: Optimistic Locks for Lookup

Many locking schemes can work with our proposed concurrent cuckoo hashing, as long as they ensure that during `Insert`, all displacements along the cuckoo path are atomic with respect to `Lookups`. The most straightforward scheme is to lock the two relevant buckets before each displacement and each `Lookup`. Though simple, this scheme requires locking twice for every `Lookup` and in a careful order to avoid deadlock.

Optimizing for the common case, our approach takes advantage of having a single writer to synchronize `Insert` and `Lookups` with low overhead. Instead of locking on buckets, it assigns a version counter for each key, updates its version when displacing this key on `Insert`, and looks for a version change during `Lookup` to detect any concurrent displacement.

Lock Striping [12] The simplest way to maintain each key's version is to store it inside each key-value object. This approach, however, adds one counter for each key and there could be hundred of millions of keys. More importantly, this approach leads to a race condition: to check or update the version of a given key, we must first lookup in the hash table to find the key-value object (stored external to the hash table), and this initial lookup is not protected by any lock and thus not thread-safe.

Instead, we create an array of counters (Figure 2). To keep this array small, each counter is shared among multiple keys by hashing (e.g., the *i*-th counter is shared by all keys whose hash value is *i*). Our implementation keeps 8192 counters in total (or 32 KB). This permits the counters to fit in cache, but allows substantial concurrent access. It also keeps the chance of a "false retry" (re-reading a key due to modification of an unrelated key) to roughly 0.01%. All counters are initialized to 0 and only read/updated by atomic memory operations to ensure the consistency among all threads.

Optimistic Locking [15] Before displacing a key, an `Insert` process first increases the relevant counter by one, indicating to the other `Lookups` an on-going update for this key; after the key is moved to its new location, the counter is again increased by one to indicate the completion. As a result, the key version is increased by 2 after each displacement.

Before a `Lookup` process reads the two buckets for a given key, it first snapshots the version stored in its counter: If this version is odd, there must be a concurrent `Insert` working on the same key (or another key sharing the same counter), and it should wait and retry; otherwise it proceeds to the two buckets. After it finishes reading both buckets, it snapshots the counter again and compares its new version with the old version. If two versions differ, the writer must have modified this key, and the `Lookup` should retry. The proof of correctness in the Appendix covers the corner cases.

3.2.2 Optimization: Multiple Cuckoo Paths

Our revised `Insert` process first looks for a valid cuckoo path before swapping the key along the path. Due to the separation of search and execution phases, we apply the following optimization to speed path discovery and increase the chance of finding an empty slot.

Instead of searching for an empty slot along one cuckoo path, our `Insert` process keeps track of multiple paths in parallel. At each step, multiple victim keys are "kicked out," each key extending its own cuckoo path. Whenever one path reaches an available bucket, this search phase completes.

With multiple paths to search, insert may find an empty slot earlier and thus improve the throughput. In addition, it improves the chance for the hash table to store a new key before exceeding the maximum number of displacements performed, thus increasing the load factor. The effect of having more cuckoo paths is evaluated in Section 5.

4 Concurrent Cache Management

Cache management and eviction is the second important component of MemC3. When serving small key-value objects, this too becomes a major source of space overhead in Memcached, which requires *18 Bytes for each key* (i.e., two pointers and a 2-Byte reference counter) to ensure that keys can be evicted safely in a strict LRU order. String LRU cache management is also a synchronization bottleneck, as all updates to the cache must be serialized in Memcached.

This section presents our efforts to make the cache management *space efficient* (1 bit per key) and *concurrent* (no synchronization to update LRU) by implementing an *approximate LRU cache* based on the CLOCK replacement algorithm [6]. CLOCK is a well-known algorithm; our contribution lies in integrating CLOCK replacement with the optimistic, striped locking in our cuckoo algorithm to reduce both locking and space overhead.

As our target workloads are dominated by small objects, the space saved by trading perfect for approximate LRU allows the cache to store siginifcantly more entries, which in turn improves the hit ratio. As we will show in Section 5, our cache management achieves 3× to 10× the query throughput of the default cache in Memcached, while also improving the hit ratio.

CLOCK Replacement A cache must implement two functions related to its replacement policy:

- `Update` to keep track of the recency after querying a key in the cache; and
- `Evict` to select keys to purge when inserting keys into a full cache.

Memcached keeps each key-value entry in a doubly-linked-list based LRU queue within its own slab class. After each cache query, `Update` moves the accessed entry to the *head* of its own queue; to free space when the cache is full, `Evict` replaces the entry on the *tail* of the queue by the new key-value pair. This ensures strict LRU eviction in each queue, but unfortunately it also requires two pointers per key for the doubly-linked list and, more importantly, all `Updates` to one linked list are serialized. Every read access requires an update, and thus the queue permits no concurrency even for read-only workloads.

CLOCK approximates LRU with improved concurrency and space efficiency. For each slab class, we maintain a *circular buffer* and a *virtual hand*; each bit in the buffer represents the recency of a different key-value object: 1 for "recently used" and 0 otherwise. Each `Update` simply sets the recency bit to 1 on each key access; each `Evict` checks the bit currently pointed by the hand. If the current bit is 0, `Evict` selects the corresponding key-value object; otherwise we reset this bit to 0 and advance the hand in the circular buffer until we see a bit of 0.

Integration with Optimistic Cuckoo Hashing The `Evict` process must coordinate with reader threads to ensure the eviction is safe. Otherwise, a key-value entry may be overwritten by a new (key,value) pair after eviction, but threads still accessing the entry for the evicted key may read dirty data. To this end, the original Memcached adds to each entry a 2-Byte reference counter to avoid this rare case. Reading this per-entry counter, the `Evict` process knows how many other threads are accessing this entry concurrently and avoids evicting those busy entries.

Our cache integrates cache eviction with our optimistic locking scheme for cuckoo hashing. When `Evict` selects a victim key x by CLOCK, it first increases key x's version counter to inform other threads currently reading x to retry; it then deletes x from the hash table to make x unreachable for later readers, including those retries; and finally it increases key x's version counter again to complete the change for x. Note that `Evict` and the hash table `Insert` are both serialized (using locks) so when updating the counters they can not affect each other.

With `Evict` as above, our cache ensures consistent `GET`s by version checking. Each `GET` first snapshots the version of the key before accessing the hash table; if the hash table returns a valid pointer, it follows the pointer and reads the value assoicated. Afterwards, `GET` compares the latest key version with the snapshot. If the verions differ, then `GET` may have observed an inconsistent intermediate state and must retry. The pseudo-code of `GET` and `SET` is shown in Algorithm 1.

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
```

5 Evaluation

This section investigates how the proposed techniques and optimizations contribute to performance and space efficiency. We "zoom out" the evaluation targets, starting with the hash table itself, moving to the cache (including the hash table and cache eviction management), and concluding with the full MemC3 system (including the cache and network). With all optimizations combined, MemC3 achieves 3× the throughput of Memcached. Our proposed core hash table if isolated can achieve 5 million lookups/sec per thread and 35 million lookups/sec when accessed by 12 threads.

5.1 Platform

All experiments run on a machine with the following configuration. The CPU of this server is optimized for energy efficiency rather than high performance, and our system is CPU intensive, so we expect the absolute performance would be higher on "beefier" servers.

CPU	2× Intel Xeon L5640 @ 2.27GHz
# cores	2 × 6
LLC	2 × 12 MB L3-cache
DRAM	2 × 16 GB DDR SDRAM
NIC	10Gb Ethernet

5.2 Hash Table Microbenchmark

In the following experiments, we first benchmark the construction of hash tables and measure the space efficiency. Then we examine the lookup performance of a single thread and the aggregate throughput of 6 threads all accessing the same hash table, to analyze the contribution of different optimizations. In this subsection, hash tables are linked into a workload generator directly and benchmarked on a local machine.

Space Efficiency and Construction Speed We insert unique keys into empty cuckoo and chaining hash tables using a single thread, until each hash table reaches its maximum capacity. The chaining hash table, as used in Mem-

Hash table	Size (MB)	# keys (million)	Byte/key	Load factor	Construction rate (million keys/sec)	Largest bucket
Chaining	1280	100.66	13.33	–	14.38	13
Cuckoo 1path	1152	127.23	9.49	94.79%	6.19	4
Cuckoo 2path	1152	127.41	9.48	94.93%	7.43	4
Cuckoo 3path	1152	127.67	9.46	95.20%	7.29	4

Table 2: Comparison of space efficiency and construction speed of hash tables. Results in this table are independent of the key-value size. Each data point is the average of 10 runs.

cached, stops insertion if $1.5n$ objects are inserted to a table of n buckets to prevent imbalanced load across buckets; our cuckoo hash table stops when a single `Insert` fails to find an empty slot after 500 consecutive displacements. We initialize both types of hash tables to have a similar size (around 1.2 GB, including the space cost for pointers)

Table 2 shows that the cuckoo hash table is much more compact. Chaining requires 1280 MB to index 100.66 million items (i.e., 13.33 bytes per key); cuckoo hash tables are both smaller in size (1152 MB) and contain at least 20% more items, using no more than 10 bytes to index each key. Both cuckoo and chaining hash tables store only pointers to objects rather than the real key-value data; the *index* size is reduced by 1/3. A smaller index matters more for small key-value pairs.

Table 2 also compares cuckoo hash tables using different numbers of cuckoo paths to search for empty slots (Section 3.2.2). All of the cuckoo hash tables have high occupancy (roughly 95%). While more cuckoo paths only slightly improve the load factor, they boost construction speed non-trivially. The table with 2-way search achieves the highest construction rate (7.43 MOPS), as searching on two cuckoo paths balances the chance to find an empty slot vs. the resources required to keep track of all paths.

Chaining table construction is twice as fast as cuckoo hashing, because each insertion requires modifying only the head of the chain. Though fast, its most loaded bucket contains 13 objects in a chain (the average bucket has 1.5 objects). In contrast, bucket size in a cuckoo hash table is fixed (i.e., 4 slots), making it a better match for our targeted read-intensive workloads.

Cuckoo Insert Although the amortized cost to insert one key with cuckoo hashing is O(1), it requires more displacements to find an empty slot when the table is more occupied. We therefore measure the insertion cost—in terms of both the number of displacements per insert and the latency—to a hash table with x% of all slots filled, and vary x from 0% to the maximum possible load factor. Using two cuckoo paths improves insertion latency, but using more than that has diminishing or negative returns. Figure 4 further shows the reciprocal throughput, expressed as latency. When the table is 70% filled, a

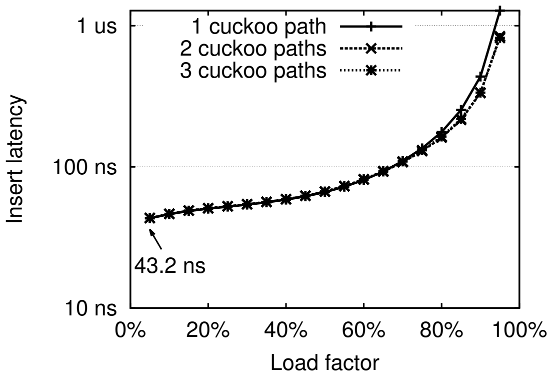
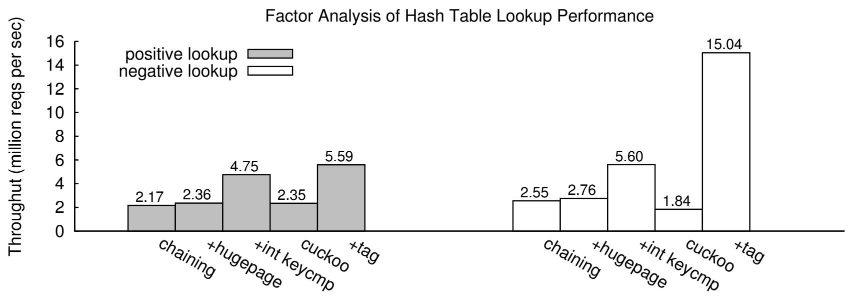


Figure 4: Cuckoo insert, with different number of parallel searches. Each data point is the average of 10 runs.

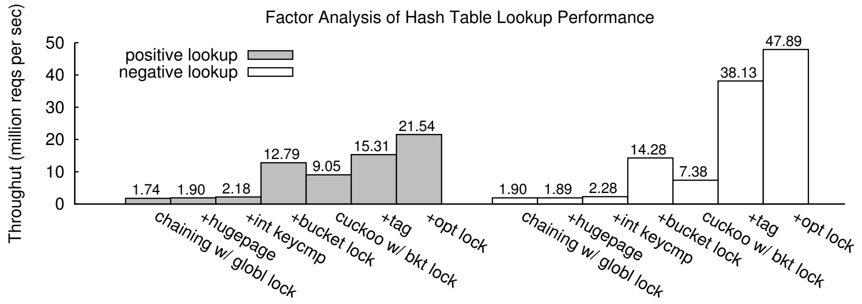
cuckoo insert can complete within 100 ns. At 95% occupancy, insert delay is 1.3 μs with a single cuckoo path, and 0.84 μs using two.

Factor Analysis of Lookup Performance This experiment investigates how much each optimization in Section 3 contributes to the hash table. We break down the performance gap between the basic chaining hash table used by Memcached and the final optimistic cuckoo hash table we proposed, and measure a set of hash tables—starting from the basic chaining and adding optimizations cumulatively as follows:

- **Chaining** is the default hash table of Memcached, serving as the baseline. A global lock is used to synchronize multiple threads.
- **+hugepage** enables 2MB x86 hugepage support in Linux to reduce TLB misses.
- **+int keycmp** replaces the default `memcmp` (used for full key comparison) by casting each key into an integer array and then comparing based on the integers.
- **+bucket lock** replaces the global lock by bucket-based locks.
- **cuckoo** applies the naive cuckoo hashing to replace chaining, without storing the tags in buckets and using bucket-based locking to coordinate multiple threads.



(a) Single-thread lookup performance with non-concurrency optimizations, all locks disabled



(b) Aggregate lookup performance of 6 threads with all optimizations

Figure 5: Contribution of optimizations to the hash table lookup performance. Optimizations are cumulative. Each data point is the average of 10 runs.

- **+tag** stores the 1-Byte hash for each key to improve cache-locality for both `Insert` and `Lookup` (Section 3.1).
- **+opt lock** replaces the per-bucket locking scheme by optimistic locking to ensure atomic displacement (Section 3.2.1).

Single-thread lookup performance is shown in Figure 5a with lookups all positive or all negative. No lock is used for this experiment. In general, combining all optimizations improves performance by ~ 2× compared to the naive chaining in Memcached for positive lookups, and by ~ 5× for negative lookups. Enabling "hugepage" improves the baseline performance slightly; while "int keycmp" can almost double the performance over "hugepage" for both workloads. This is because our keys are relatively small, so the startup overhead in the built-in `memcmp` becomes relatively large. Using cuckoo hashing without the "tag" optimization *reduces* performance, because naive cuckoo hashing requires more memory references to retrieve the keys in all $4 \times 2 = 8$ candidate locations on each lookup (as described in Section 3.1). The "tag" optimization therefore significantly improves the throughput of read-only workloads (2× for positive lookups and 8× for negative lookups), because it compares the 1-byte tag first before fetching the real keys outside the table and thus eliminates a large fraction of CPU cache misses.

Multi-thread lookup performance is shown in Figure 5b, measured by aggregating the throughput from 6 threads accessing the same hash table. Different from the previous experiment, a global lock is used for the baseline chaining (as in Memcached by default) and replaced by per-bucket locking and finally optimistic locking for the cuckoo hash table.

The performance gain (~ 12× for positive and ~ 25× for negative lookups) of our proposed hashing scheme over the default Memcached hash table is large. In Memcached, all hash table operations are serialized by a global lock, thus the basic chaining hash table in fact performs worse than its single-thread throughput in Figure 5a. The slight improvement (< 40%) from "hugepage" and "int keycmp" indicates that most performance benefit is from making the data structures concurrent. The "bucket lock" optimization replaces the global lock in chaining hash tables and thus significantly improves the performance by 5× to 6× compared to "int keycmp". Using the basic concurrent cuckoo reduces throughput (due to unnecessary memory references), while the "tag" optimization is again essential to boost the performance of cuckoo hashing and outperform chaining with per-bucket locks. Finally, the optimistic locking scheme further improves the performance significantly.

Multi-core Scalability Figure 6 illustrates how the total hash table throughput changes as more threads access

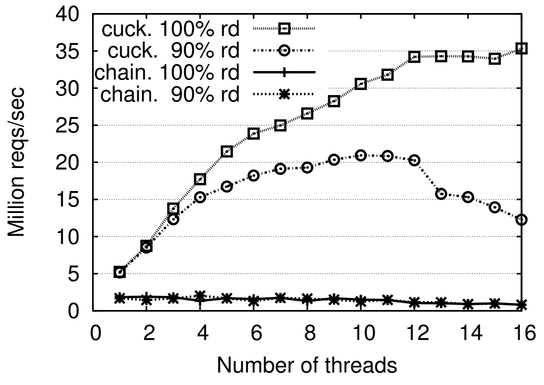


Figure 6: Hash table throughput vs. number of threads. Each data point is the average of 10 runs.

the same hash table. We evaluate read-only and 10% write workloads. The throughput of the default hash table does not scale for either workload, because all hash table operations are serialized. Due to lock contention, the throughput is actually lower than the single-thread throughput without locks.

Using our proposed cuckoo hashing for the read-only workload, the performance scales linearly to 6 threads because each thread is pinned on a dedicated physical core on the same 6-core CPU. The next 6 threads are pinned to the other 6-core CPU in the same way. The slope of the curve becomes lower due to cross-CPU memory traffic. Threads after the first 12 are assigned to already-busy cores, and thus performance does not further increase.

With 10% `Insert`, our cuckoo hashing reaches a peak performance of 20 MOPS at 10 threads. Each `Insert` requires a lock to be serialized, and after 10 threads the lock contention becomes the bottleneck.

We further vary the fraction of insert queries in the workload and measure the best performance achieved by different hash tables. Figure 7 shows this best performance and also the number of threads (between 1 and 16) required to achieve this performance. In general, cuckoo hash tables outperform chaining hash tables. When more write traffic is generated, performance of cuckoo hash tables declines because `Inserts` are serialized and more `Lookups` happen concurrently. Consequently, the best performance for 10% insert is achieved using only 9 threads; while with 100% lookup, it scales to 16 threads. Whereas the best performance of chaining hash tables (with either a global lock or per-bucket locks) keeps roughly the same when the workloads become more write-intensive.

5.3 Cache Microbenchmark

Workload We use YCSB [5] to generate 100 million key-value queries, following a zipf distribution. Each key is 16 Bytes and each value 32 Bytes. We evaluate caches with four configurations:

- **chaining+LRU**: the default Memcached cache configuration, using chaining hash table to index keys and LRU for replacement;
- **cuckoo+LRU**: keeping LRU, but replacing the hash table by concurrent optimistic cuckoo hashing with all optimizations proposed;
- **chaining+CLOCK**: an alternative baseline combining optimized chaining with the CLOCK replacement algorithm. Because CLOCK requires no serialization to update, we also replace the global locking in the chaining hash table with the per-bucket locks; we further include our engineering optimizations such as "hugepage", "int keycmp".
- **cuckoo+CLOCK**: the data structure of MemC3, using cuckoo hashing to index keys and CLOCK for replacement.

We vary the cache size from 64 MB to 10 GB. Note that this cache size parameter does not count the space for the hash table, only the space used to store key-value objects. All four types of caches are linked into a workload generator and micro-benchmarked locally.

Cache Throughput Because each `GET` miss is followed by a `SET` to the cache, to understand the cache performance with heavier or lighter insertion load, we evaluate two settings:

- a read-only workload on a "big" cache (i.e., 10 GB, which is larger than the working set), which had no cache misses or inserts and is the best case for performance;
- a write-intensive workload on a "small" cache (i.e., 1 GB, which is ~10% of the total working set) where about 15% `GETs` miss the cache. Since each miss triggers a `SET` in turn, a workload with 15% inserts is worse than the typical real-world workload reported by Facebook [3].

Figure 8a shows the results of benchmarking the "big cache". Though there are no inserts, the throughput does not scale for the default cache (chaining+LRU), due to lock contention on each LRU update (moving an object to the head of the linked list). Replacing default chaining with the concurrent cuckoo hash table improves the peak throughput slightly. This suggests that only having a concurrent hash table is not enough for high performance. After replacing the global lock with bucket-based locks and removing the LRU synchronization bottleneck by using CLOCK, the chaining-based cache achieves 22

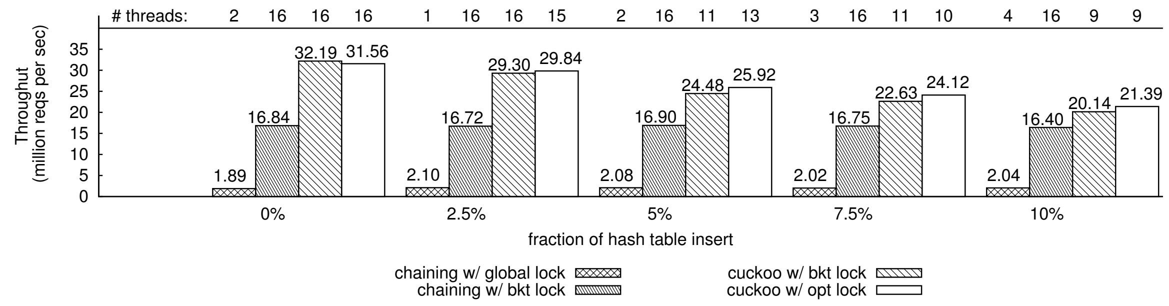


Figure 7: Tradeoff between lookup and insert performance. Best read + write throughput (with the number of threads needed to achieve it) is shown. Each data point is the average of 10 runs.

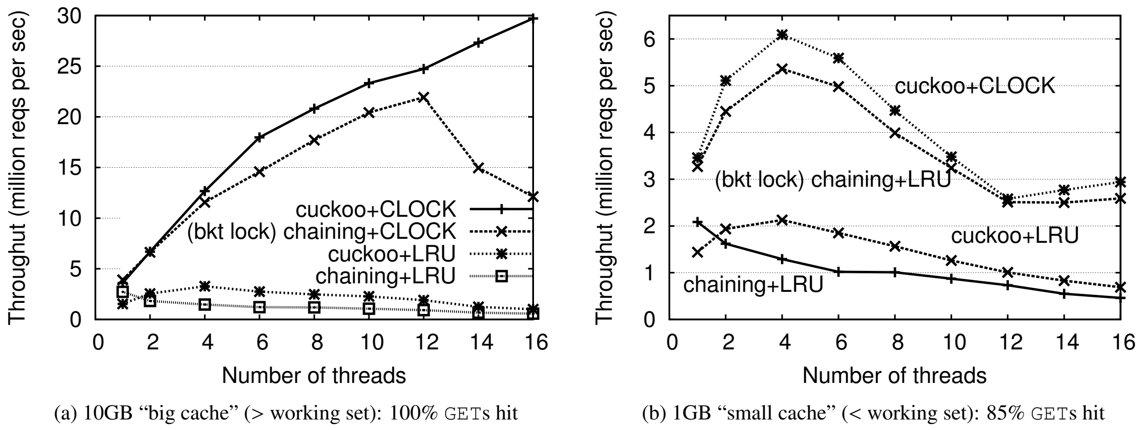


Figure 8: Cache throughput vs. number of threads. Each data point is the average of 10 runs.

MOPS at 12 threads, and drops quickly due to the CPU overhead for lock contention after all 12 physical cores are assigned. Our proposed cuckoo hash table combined with CLOCK, however, scales to 30 MOPS at 16 threads.

Figure 8b shows that peak performance is achieved at 6 MOPS for the "small cache" by combining CLOCK and cuckoo hashing. The throughput drop is because the 15% `GET` misses result in about 15% hash table inserts, so throughput drops after 6 threads due to serialized inserts.

Space Efficiency Table 3 compares the maximum number of items (16-Byte key and 32-Byte value) a cache can store given different cache sizes[5]. The default LRU with chaining is the least memory efficient scheme. Replacing chaining with cuckoo hashing improves the space utilization slightly (7%), because one pointer (for hash table chaining) is eliminated from each key-value object. Keeping chaining but replacing LRU with CLOCK improves space efficiency by 27% because two pointers (for LRU) and one reference count are saved per object. Combining CLOCK with cuckoo increases the space efficiency by 40% over the default. The space benefit arises from eliminating three pointers and one reference count per object.

Cache Miss Ratio Compared to the linked list based approach in Memcached, CLOCK approximates LRU eviction with much lower space overhead. This experiment sends 100 million queries (95% `GET` and 5% `SET`, in zipf distribution) to a cache with different configurations, and measures the resulting cache miss ratios. Note that each `GET` miss will trigger a retrieval to the backend database system, therefore reducing the cache miss ratio from 10% to 7% means a reduction of traffic to the backend by 30%. Table 3 shows when the cache size is smaller than 256 MB, the LRU-based cache provides a lower miss ratio than CLOCK. LRU with cuckoo hashing improves upon LRU with chaining, because it can store more items. In this experiment, 256 MB is only about 2.6% of the 10 GB working set. Therefore, when the cache size is very small, CLOCK—which is an

[5] The space to store the index hash tables is separate from the given cache space in Table 3. We set the hash table capacity larger than the maximum number of items that the cache space can possibly allocate. If chaining is used, the chaining pointers (inside each key-value object) are also allocated from the cache space.

	cache type	cache size					
		64 MB	128 MB	256 MB	512 MB	1 GB	2 GB
# items stored (million)	chaining+LRU	0.60	1.20	2.40	4.79	9.59	19.17
	cuckoo+LRU	0.65	1.29	2.58	5.16	10.32	20.65
	chaining+CLOCK	0.76	1.53	3.05	6.10	12.20	24.41
	cuckoo+CLOCK	**0.84**	**1.68**	**3.35**	**6.71**	**13.42**	**26.84**
cache miss ratio 95% GET, 5% SET zipf distribution	chaining+LRU	36.34%	31.91%	27.27%	22.30%	16.80%	10.44%
	cuckoo+LRU	**35.87%**	**31.42%**	**26.76%**	21.74%	16.16%	9.80%
	chaining+CLOCK	37.07%	32.51%	27.63%	22.20%	15.96%	8.54%
	cuckoo+CLOCK	36.46%	31.86%	26.92%	**21.38%**	**14.68%**	**7.89%**

Table 3: Comparison of four types of caches. Results in this table depend on the object size (16-Byte key and 32-Byte value used). Bold entries are the best in their columns. Each data point is the average of 10 runs.

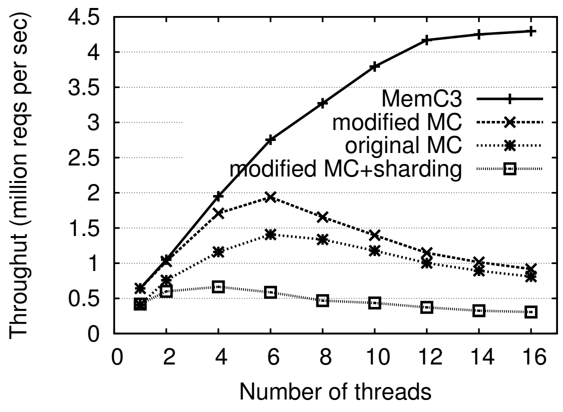


Figure 9: Full system throughput (over network) v.s. number of server threads

approximation—has a higher chance of evicting popular items than strict LRU. For larger caches, CLOCK with cuckoo hashing outperforms the other two schemes because the extra space improves the hit ratio more than the loss of precision decreases it.

5.4 Full System Performance

Workload This experiment uses the same workload as in Section 5.3, with 95% GETs and 5% SETs generated by YCSB with zipf distribution. MemC3 runs on the same server as before, but the clients are 50 different nodes connected by a 10GB Ethernet. The clients use libmemcached 1.0.7 [16] to communicate with our MemC3 server over the network. To amortize the network overhead, we use multi-get supported by libmemcached [16] by batching 100 GETs.

In this experiment, we compare four different systems: original Memcached, optimized Memcached (with non-algorithmic optimizations such as "hugepage", "in key-cmp" and tuned CPU affinity), optimized Memcached with sharding (one core per Memcached instance) and MemC3 with all optimizations enabled. Each system is allocated with 1GB memory space (not including hash table space).

Throughput Figure 9 shows the throughput as more server threads are used. Overall, the maximum throughput of MemC3 (4.4 MOPS) is almost 3× that of the original Memcached (1.5 MOPS). The non-algorithmic optimizations improve throughput, but their contribution is dwarfed by the algorithmic and data structure-based improvements.

A surprising result is that today's popular technique, *sharding, performs the worst* in this experiment. This occurs because the workload generated by YCSB is heavy-tailed, and therefore imposes differing load on the memcached instances. Those serving "hot" keys are heavily loaded while the others are comparatively idle. While the severity of this effect depends heavily upon the workload distribution, it highlights an important benefit of MemC3's approach of sharing all data between all threads.

6 Related Work

This section presents two categories of work most related to MemC3: efforts to improve individual key-value storage nodes in terms of throughput and space efficiency; and the related work applying cuckoo hashing.

Flash-based key-value stores such as BufferHash [1], FAWN-DS [2], SkimpyStash [8] and SILT [17] are optimized for I/O to external storage such as SSDs (e.g., by batching, or log-structuring small writes). Without slow I/O, the optimization goals for MemC3 are saving memory and eliminating synchronization. Previous work in *memory-based key-value stores* [4, 20, 13] boost performance on multi-core CPUs or GP-GPUs by sharding data to dedicated cores to avoid synchronization. MemC3 instead targets read-mostly workloads and deliberately avoids sharding to ensure high performance even for "hot" keys. Similar to MemC3, Masstree [18] also applied extensive optimizations for cache locality and optimistic concurrency control, but used very different techniques because it was a variation of B+-tree to support range queries. RAMCloud [22] focused on fast data reconstruction from on-disk replicas. In contrast, as a cache, MemC3 specifically takes advantage of the transience of the data it stores to improve space efficiency.

Cuckoo hashing [23] is an open-addressing hashing scheme with high space efficiency that assigns multiple candidate locations to each item and allows inserts to kick existing items to their candidate locations. FlashStore [7] applied cuckoo hashing by assigning each item 16 locations so that each lookup checks up to 16 locations, while our scheme requires reading only 2 locations in the hash table. We previously proposed *partial key cuckoo hashing* in the SILT system [17] to achieve high occupancy with only two hash functions, but our earlier algorithm limited the maximum hash table size and was therefore unsuitable for large in-memory caches. Our improved algorithm eliminates this limitation while retaining high memory efficiency. To make cuckoo operations concurrent, the prior approach of Herlihy et al. [12] traded space for concurrency. In contrast, our optimistic locking scheme allows concurrent readers without losing space efficiency.

7 Conclusion

MemC3 is an in-memory key-value store that is designed to provide caching for read-mostly workloads. It is built on carefully designed and engineered algorithms and data structures with a set of architecture-aware and workload-aware optimizations to achieve high concurrency, space-efficiency and cache-locality. In particular, MemC3 uses a new hashing scheme—optimistic cuckoo hashing—that achieves over 90% space occupancy and allows concurrent read access without locking. MemC3 also employs CLOCK-based cache management with only 1-bit per entry to approximate LRU eviction. Compared to Memcached, it reduces space overhead by more than 20 Bytes per entry. Our evaluation shows the throughput of our system is 3× higher than the original Memcached while storing 30% more objects for small key-value pairs.

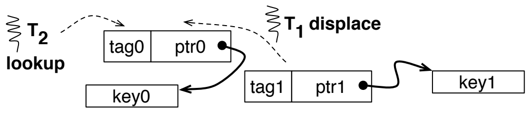
Acknowledgments

This work was supported by funding from the Intel Science and Technology Center for Cloud Computing, National Science Foundation award CCF-0964474, Google, and the PDL member companies. We thank the NSDI reviewers for their feedback, Iulian Moraru, Dong Zhou, Michael Mitzenmacher, and Rasmus Pagh for their valuable algorithmic suggestions, and Matthew Caesar for shepherding this paper.

A Correctness of the optimistic locking on keys

This appendix examines the possible interleavings of two threads in order to show that the optimistic locking scheme correctly prevents `Lookup` from returning wrong or corrupted data. Assume that threads T_1 and T_2 concurrently access the hash table. When both threads perform `Lookup`, correctness is trivial. When both `Insert`, they are serialized (`Insert` is guarded by a lock). The remaining case occurs when T_1 is `Insert` and T_2 is `Lookup`.

During `Insert`, T_1 may perform a sequence of displacement operations where each displacement is proceeded and followed by incrementing the counter. Without loss of generality, assume T_1 is displacing `key1` to a destination slot that originally hosts `key0`. Each slot contains a tag and a pointer, as shown:



`key0` differs from `key1` because there are no two identical keys in the hash table, which is guaranteed because every `Insert` effectively does a `Lookup` first. If T_2 reads the same slot as T_1 before T_1 completes its update:

case1: T_2 is looking for `key0`. Because `Insert` moves backwards along a cuckoo path (Section 3.2), `key0` must have been displaced to its other bucket (say bucket *i*), thus

- if T_2 has not checked bucket *i*, it will find `key0` when it proceeds to that bucket;
- if T_2 checked bucket *i* and did not find `key0` there, the operation that moves `key0` to bucket *i* must happen after T_2 reads bucket *i*. Therefore, T_2 will see a change in `key0`'s version counter and make a retry.

case2: T_2 is looking for `key1`. Since T_1 will atomically update `key1`'s version before and after the displacement, no matter what T_2 reads, it will detect the version change and retry.

case3: T_2 is looking for a key $\neq$ `key0` or `key1`. No matter what T_2 sees in the slot, it will be rejected eventually, either by the tags or by the full key comparison following the pointers. This is because the pointer field of this slot fetched by T_2 is either `ptr(key0)` or `ptr(key1)` rather than some corrupted pointer, ensured by the atomic read/write for 64-bit aligned pointers on 64-bit machines.[6] ∎

[6]quadword memory access aligned on a 64-bit boundary are atomic on Pentium and newer CPUs [14]

References

[1] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems. In *Proc. 7th USENIX NSDI*, Apr. 2010.

[2] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A fast array of wimpy nodes. *Communications of the ACM*, 54(7): 101–109, July 2011.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the SIGMETRICS'12*, 2012.

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store. In *In Proceedings of the Second International Green Computing Conference*, Aug. 2011.

[5] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *Proc. 1st ACM Symposium on Cloud Computing (SOCC)*, June 2010.

[6] F. Corbato and M. I. O. T. C. P. MAC. *A Paging Experiment with the Multics System.* Defense Technical Information Center, 1968. URL `http://books.google.com/books?id=5wDQNwAACAAJ`.

[7] B. Debnath, S. Sengupta, and J. Li. FlashStore: high throughput persistent key-value store. *Proc. VLDB Endow.*, 3:1414–1425, Sept. 2010.

[8] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM space skimpy key-value store on flash. In *Proc. ACM SIGMOD*, June 2011.

[9] U. Erlingsson, M. Manasse, and F. Mcsherry. A cool and practical alternative to traditional hash tables. In *Seventh Whorkshop on Distributed Data and Structures (WDAS'2006)*, pages 1–6, 2006.

[10] Facebook Engineering Notes. Scaling memcached at Facebook. `http://www.facebook.com/note.php?note_id=39391378919`.

[11] N. Gunther, S. Subramanyam, and S. Parvu. Hidden scalability gotchas in memcached and friends. In *VELOCITY Web Performance and Operations Conference*, June 2010.

[12] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916, 9780123705914.

[13] T. H. Hetherington, T. G. Rogers, L. Hsu, M. O'Connor, and T. M. Aamodt. Characterizing and evaluating a key-value store application on heterogeneous cpu-gpu systems. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2012.

[14] intel-dev3a. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 3A. `http://www.intel.com/content/www/us/en/architecture-and-technology/`, 2011.

[15] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2): 213–226, June 1981. ISSN 0362-5915.

[16] libMemcached. libmemcached. `http://libmemcached.org/`, 2009.

[17] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A memory-efficient, high-performance key-value store. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[18] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *Proc. EuroSys*, pages 183–196, Apr. 2012.

[19] Memcached. A distributed memory object caching system. `http://memcached.org/`, 2011.

[20] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. Cphash: a cache-partitioned hash table. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2012.

[21] MongoDB. `http://mongodb.com`.

[22] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in ramcloud. In *Proc. 23rd ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2011.

[23] R. Pagh and F. Rodler. Cuckoo hashing. *Journal of Algorithms*, 51(2):122–144, May 2004.

[24] N. Provos. libevent. `http://monkey.org/~provos/libevent/`.

[25] Redis. `http://redis.io`.

[26] VoltDB. VoltDB, the NewSQL database for high velocity applications. `http://voltdb.com/`.