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

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

CMU-PDL-12-116
November 2012

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

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.

Keywords: hashing, performance, memory efficiency, key-value store

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 [21, 28, 23, 27] or as a cache to deflect hot or particularly latency-sensitive items [12].

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 [21], 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 heavily on locking to ensure consistency among multiple threads, and leads to poor scalability on multi-core CPUs [13].

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 realities 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 any per-byte 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 [25] achieves space efficiency, but is unfriendly for concurrent operations. Optimistic cuckoo hashing (1) achieves high memory efficiency (e.g., 93% table occupancy [19]); (2) supports highly concurrent accesses to the hash table with read-intensive workloads; and (3) keeps hash table operations cache-friendly. (Section 3).

- A compact LRU cache 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 synchronization 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

function	std Memcached	MemC3
Hash Table		
concurrency	serialized	concurrent lookup, serialized insert
throughput		
lookup	$O(\log n / \log\log n)$	$O(1)$
insert	$O(1)$	$O(1)$
space	$13.3n$ Bytes	$\sim 9.7n$ Bytes
Cache Mgmt		
concurrency	serialized	concurrent update, serialized eviction
throughput		
update	$O(1)$	$O(1)$
eviction	$O(1)$	$O(1)$
space	$18n$ Bytes	n bits

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

on Memcached-1.4.13.[1] As shown in Table 1, our system provides higher throughput while requiring significantly less memory and computation than standard Memcached.

2 Background

2.1 Memcached Overview

Interface Memcached implements a simple and light-weight key-value interface where all key-value tuples are stored 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 all 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. 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.*

[1]Our prototype does not yet provide the full Memcached API.

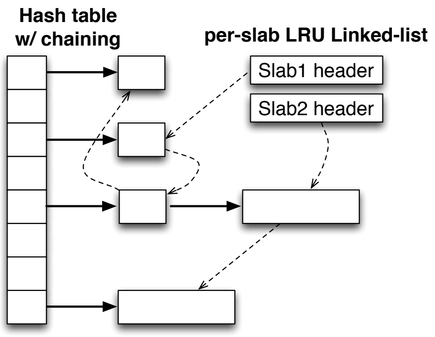


Figure 1: Memcached data structures.

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 appropriatedly-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 [26]. 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 [13].

Performance Enhancement Previous solutions [4, 22, 15] have been proposed to improve Memcached performance [13] by sharding the in-memory data to different cores. Sharding eliminates the inter-thread synchronization and thus permits 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 which 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 strongly 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 can not 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 GETs (e.g., 99.8%). 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: locks are used extensively on the query path. It 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 completely remove all locks 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 significantly improves memory efficiency by applying cuckoo hashing [25]—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 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 [**concurrency, memory efficiency, cache-friendliness**];
- A technique using a short summary of each key to improve the cache locality of hash table operations [**memory efficiency, cache-friendliness**]; and
- An optimization for cuckoo hashing insertion that improves the throughput and the table occupancy [**memory efficiency**].

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 [2]. 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 high concurrency for read-heavy 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`, a pointer is returned referencing 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`.

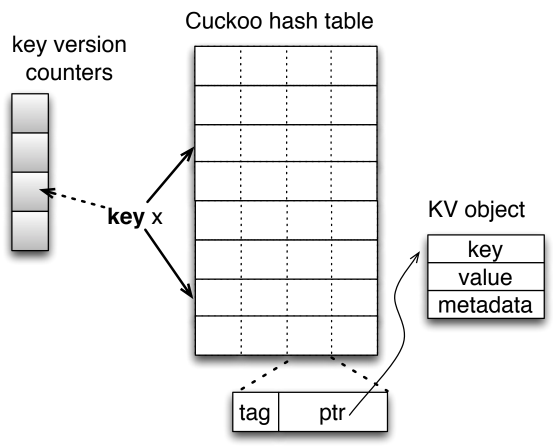


Figure 2: Hash table overview: The hash table is 4-way set-associative. Each key is mapped to 2 buckets by hash functions and associated with 1 version counter; Each slot stores a tag of the key and a pointer to the key-value item. Values in gray are used for optimistic locking and must be accessed atomically.

Basic Cuckoo Hashing Before presenting our techniques in detail, we first briefly describe how to perform basic cuckoo hashing on a 4-way set-associative hash table.[3] As shown in Figure 2, our hash table is an array of *buckets*. Each bucket consists of 4 *slots* and each slot contains a *pointer* to the key-value object and a short summary of 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 each having 4 slots, thus `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., 128 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 still $O(1)$ [25].

3.1 Tag-based Lookup/Insert

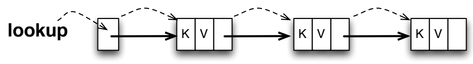
To support keys of variable length and keep the index compact, the actual keys are not stored in the hash table and must be retrieved by following a pointer. We propose a cache-aware technique

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

[3] 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 associativie hash table. [11]

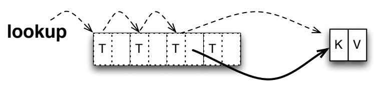
to perform cuckoo hashing with minimum memory references by using *tags*—a short hash of the keys (one-byte in our implementation). This technique is inspired by "partial-key cuckoo hashing" which we proposed in previous work [19], but eliminates the prior approach's limitation in the maximum table size.

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



Neither is the 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 eliminiate 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

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

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 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

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

As a result, `Insert` operations can operate using only information in the table and never has 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 speicific 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.

3.2 Concurrent Cuckoo Hashing

Effectively supporting concurrent access to a cuckoo hash table is challenging. The previously proposed scheme improved concurrency by trading space [14]. 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).

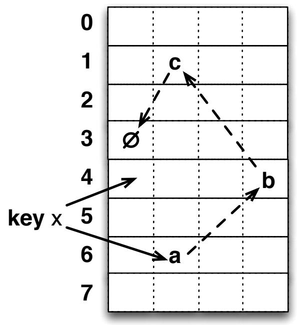


Figure 3: Cuckoo path. ∅ represents an empty slot.

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 acquring 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 [14] that we know of breaks up `Insert`s 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 any 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 single 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 constrast, 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 atomic displacements along the cuckoo path with respect to the readers. The most straightforward scheme is to lock two relevant buckets before each displacement and each `Lookup`. Though simple, this scheme requires locking twice for every `Lookup`. The buckets must also be locked in a careful order to avoid deadlock.

Optimizing for the common case, our approach takes advantage of having a single writer to implement lock-free `Lookup`s. Instead of locking on buckets, it maintains a version counter for each key, updates it on each displacement, and looks for a version change during `Lookup`.

Lock Striping [14] To store all the key versions, we create an array of counters (Figure 2); each counter is initialized to 0 and only read/updated by atomic memory operations. There could be millions of keys in our hash table, so to keep the counter array small, each counter is shared among multiple keys by hashing. Our implementation uses 8192 counters by default to fit the array in cache, but also permit enough parallelism.

Optimistic Locking [16] Before displacing a key, the `Insert` process first increases the relevant counter by one, indicating to the other `Lookup`s 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 the `Lookup` process reads the buckets for a given key, it first checks the counter. If the version is odd, there must be a concurrent writer working on the same key (or another key sharing the same counter), and the reader should wait and retry; otherwise `Lookup` proceeds to the two buckets. After it finishes reading both buckets, it checks the counter again and compares this new version with the old version. If two versions differ, it indicates that the writer has modified this key, and the reader should retry. We attach a proof of correctness in the Appendix in detail including all 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 and thus increase the load factor. The effect of having more cuckoo paths is evaluted in Section 5.

4 Concurrent Cache Management

Cache management is the second important component of MemC3. It evicts keys when space is full, but aims to maintain a high hit ratio. Surprisingly, when serving small key-value objects, it also becomes a major source of space overhead. For example, in Memcached it requires *18 Bytes for each key* (i.e., two pointers and a 2-Byte reference counter) to ensure that keys can be safely evicted in an strict LRU order. In addition, it is also a synchronization bottleneck as all updates to the cache for LRU maintainace are serialized in Memcached.

This section presents our efforts to make the cache management extremely *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 [7]. 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 LRU for approximated LRU allows the cache to store siginifcantly more entries, and in turn improves the hit ratio. As we will show in Section 5, our cache management achieves 3× to 10× of the throughput of query the default cache of Memcached, while also improves the hit ratio.

CLOCK Replacement A cache must implement two functions:

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

In Memcached, each key-value entry is kept 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 for inserting new keys 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 it is 0, `Evict` selects the corresponding key-value object; otherwise we reset this bit to 0 and move forward the hand in the circular buffer until we see a bit of 0.

Integration with Optimistic Cuckoo Hashing The `Evict` process must coordinate with other reader threads to ensure the eviction is safe. Otherwise, a key-value entry may be overwritten by

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
```

a new (key,value) pair after eviction, but some other threads are still accessing this entry for the evicted key and get 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 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 followes 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 function `GET` and `SET` is shown in Algorithm 1.

.

5 Evaluation

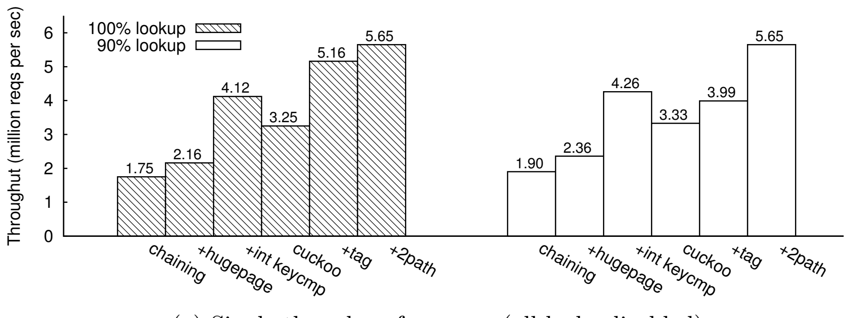
This section investigates how the proposed techniques and optimizations contribute to performance and space efficiency, by "zooming out" the evaluation targets from the hash table, to the cache (including the hash table and cache eviction management) and finally to the full MemC3 system (including the cache and network). The throughput of MemC3 is much higher than Memcached with its default hash table. 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

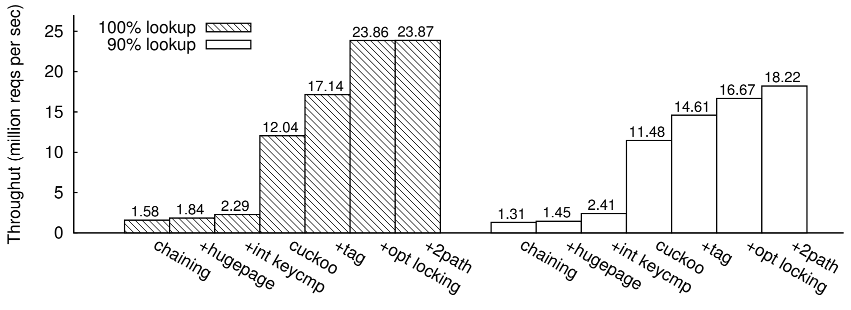
All experiments run on a machine with following configuration: The CPU of this server is optimized for energy efficiency rather than high performance, and our system is CPU intensive, so we expect the absolute performance would be higher on more "beefy" servers.

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



(a) Single thread performance (all locks disabled)



(b) Aggregate performance of 6 threads, with locking

Figure 4: Contribution of optimizations to the hash table performance. Optimizations are cumulative. Each data point is the average of 10 runs.

Workload In the following experiments, we benchmark two types of workloads: one workload has only `Lookup` queries, and the other workload consists of 90% `Lookup` and 10% `Insert`. We first focus on the optimizations not contributing to concurrency by benchmarking the hash table using a single thread without any locks. Then we examine the performance benefit with concurrency optimizations included by measuring the aggregate throughput of 6 threads all accessing the same hash table.

Factor Analysis To investigate how much each optimization in Section 3 improves hash table performance, 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 optimiztions cumulatively as follows:

- **Chaining** is the default hash table of Memcached, serving as the baseline.
- **+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.
- **cuckoo** applies the concurrent cuckoo hashing to replace chaining, using bucket-based locking to coordinate multiple threads (Section 3.2).
- **+tag** stores the 1-Byte hash for each key to improve cache-locality for both `Insert` and `Lookup` (Section 3.1).
- **+opt locking** replaces the per-bucket locking scheme by optimistic locking to ensure atomic displacement (Section 3.2.1).
- **+2path** searches two cuckoo paths for `Insert` in parallel (Section 3.2.2). We justify the choice of two paths in a later experiment.

Single-thread performance is shown in Figure 4a. In general, combining all optimizations improves performance by $\sim 3\times$ compared to the naive chaining in Memcached, and the hash table performs similarly for both workloads. Enabling "hugepage" improves the baseline performance by about 24%; while using "int keycmp" shows a significant improvement (80% to 90% over "hugepage") for both workloads. This is because our keys are relatively small, so the startup overhead in the builtin `memcmp` becomes relatively large. Directly applying cuckoo hashing without using "tag" and "2-path" insertion *reduces* performance compared to "int keycmp", due to increased memory references. The "tag" optimization significantly improves the throughput of read-only workloads by eliminating comparing every candidate key, and "2-path" optimization greatly helps the 10% `Insert` workload.

Multi-thread performance is shown in Figure 4b, measured by aggregating the throughput from 6 threads accessing the same hash table. Different from the previous experiment, a global lock is used for chaining (as in Memcached by default) and optimistic locking is included as one optimization for cuckoo hash table.

First of all, there is a huge performance gap ($\sim 15\times$ for read-only, and $\sim 14\times$ for read-intensive) between our proposed hashing scheme and the default Memcached hash table. 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 4a. The slight improvement ($< 80\%$) from "hugepage" and "int keycmp" indicates that most performance benefit is from making the data structures concurrent. Using the basic concurrent cuckoo improves throughput by $5\times$ to $6\times$,

Hash table	Size (MB)	# keys (million)	Byte/key	Load factor	insert tput (MOPS)	largest bucket
Chaining	1280	100.66	13.33	–	14.38	13
Cuckoo 1path	1152	121.91	9.91	90.84%	6.46	4
Cuckoo 2path	1152	124.52	9.70	92.78%	6.66	4
Cuckoo 3path	1152	126.80	9.53	94.48%	6.19	4
Cuckoo 4path	1152	127.76	9.45	95.19%	6.03	4

Table 2: Comparison of two types of indexes. Results in this table are independent of the key-value size. Each data point is the average of 10 runs.

while optimistic locking further improves the throughput, especially for read-only workloads. For the workload with 90% `Lookup`, having multiple cuckoo paths again improves the throughput.

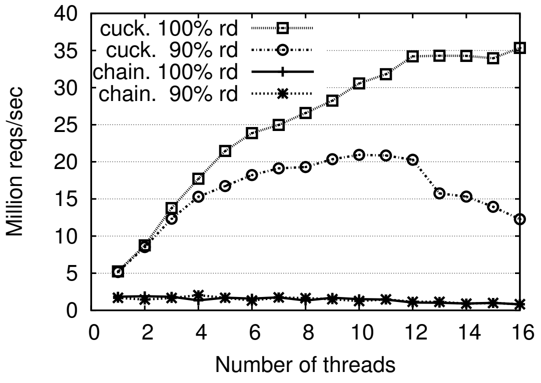


Figure 5: Throughput vs. number of threads. Each data point is the average of 10 runs.

Multi-core Scalability Figure 5 illustrates how the total hash table throughput changes as more threads access the same hash table. The throughput of the default hash table does not scale for either workload, because all hash table operations are serializied. 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 dedicate 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 packed with early threads (hyper-threading is enabled), and thus performance does not increase after 12 threads.

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

Space Efficiency This experiment measures space efficiency by inserting new keys—uniformly distributed in a 16-Byte key space—to an empty hash table (either a cuckoo or a chaining hash table) using a single thread, until the hash table reaches maximum capacity. To prevent imbalanced load across buckets, by default Memcached stops insertion if $1.5n$ objects are inserted to a table of n buckets, whereas our cuckoo hash table stops when a single `Insert` fails to find a empty slot after 128 displacements.

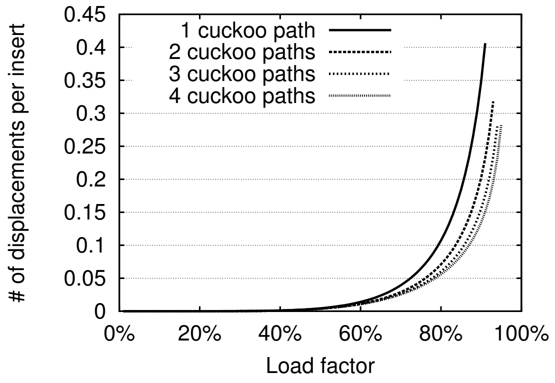


Figure 6: Number of cuckoo operations vs. load factor, with different number of parallel searches. Each data point is the average of 10 runs.

Table 2 shows that the cuckoo hash table is significantly more compact. Chaining requires 1280 MB to store 100.66 million items; while cuckoo hash tables are both smaller in size (1152 MB) but and contain at least 20% more items. Indexing each key, on average, reqiures 13.33 Bytes with chaining, but no more than 10 Bytes by cuckoo hashing. Note that this result is independent of the key-value size as the hash table only stores pointers rather than the real data.

Table 2 also compares cuckoo hash tables using different number of cuckoo paths to search for empty slots. In general, cuckoo hashing ensures high occupancy ($> 90\%$); having more cuckoo paths improves the load factor from 90% with a single path to 95% with 4 paths, because this helps reach an empty slot within the search depth of 128. Accordingly, the average overhead to index one key is reduced to 9.4 Bytes with 4 paths.

In terms of insert throughput, chaining achieves 2x the performance compared to cuckoo hashing. On the other hand, its most loaded bucket contains 13 objects in a chain where on average each bucket only hosts 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 achieves the best throughput (6.66 MOPS) with 2-way search, as searching on two cuckoo paths balances the chance to find an empty slot, and the resource required to keep track of all paths.

Cuckoo Insert Inserting one key with cuckoo hashing is, on average O(1), in theory. We measure the average insertion cost—in terms of the number of displacements per insert— to a hash table with a fraction ($x\%$) of all slots filled, and vary x from 0% to the maximum possible load factor. Figure 6 shows a good property of cuckoo insert: the cost remains low (< 0.1) before the table is 80% filled. Even when the hash table is full, the average cost is still below 0.4 displacements per insert. Using two cuckoo paths further reduces the insertion cost; but using more than two paths has diminishing space benefits and reduces insertion speed.

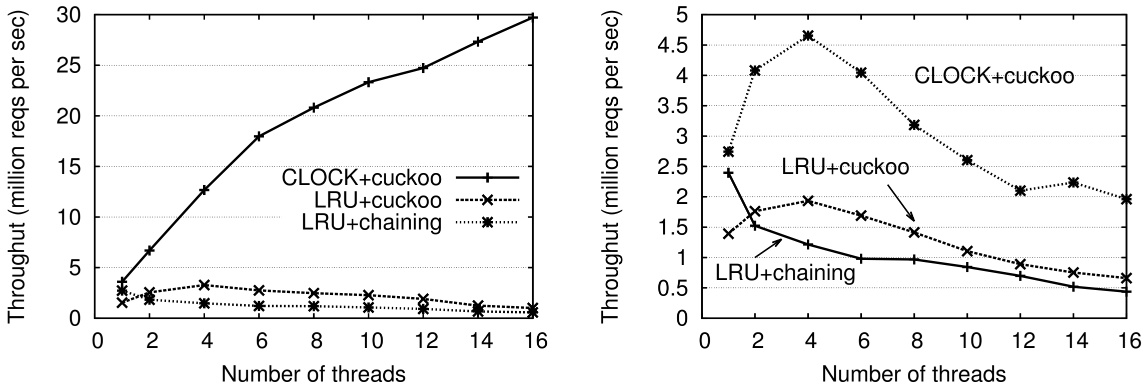
5.3 Cache Microbenchmark

Workload We use YCSB [6] 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 following three different configurations:

- **LRU+chaining**: the default Memcached cache configuration, using chaining hash table to index keys and LRU for replacement;

- **LRU+cuckoo**: keeping the LRU part but replacing the hash table by concurrent optimistic cuckoo hashing with all optimizations proposed;
- **CLOCK+cuckoo**: 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, the cache size does not count the space for hash table, only the space available to store key-value objects.



(a) 10GB cache(> working set), read-ony: 100% `GET`s hit

(b) 1GB cache(< working set), 95% `GET` + 5% `SET`: 85% `GET`s hit

Figure 7: Throughput vs. number of threads. Each data point is the average of 10 runs.

Throughput The total insertion rate to a cache depends not only on the `SET`s requested by the application but also on the hit ratio of `GET`s, because each `GET` miss is followed by an insertion. Therefore, even a `GET`-only workload at the application level may still result in a high insert rate at the cache if most `GET`s miss. To understand the cache performance with heavier or lighter insertion load, we evaluate two different settings:

- a `GET`-only workload on a "big" cache (i.e., 10 GB, larger than the working set), which leads no cache miss or insert and is the best case for performance;
- a `GET`(95%)/`SET`(5%) mixed workload on a "small" cache (i.e., 1 GB, ∼10% of the total working set). About 85% `GET`s hit the cache, and each miss triggers an insert in turn, which is worse than the typical real-world workload reported by Facebook [3].

Figure 7a shows the results of benchmarking the "big cache". Though there are no inserts, the throughput does not scale for the default cache, due to the intensive 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 removing the LRU synchronization bottleneck by using CLOCK, the throughput achieves 30 MOPS at 16 threads.

Figure 7b shows that peak performance is achieved at 4.5 MOPS for the "small cache" by combining CLOCK and cuckoo hashing. The throughput drop is because the 5% `SET`s and 15% `GET` misses result in about 20% hash table inserts, so throughput drops after 6 threads due to serialized inserts.

	cache size					
	64 MB	128 MB	256 MB	512 MB	1 GB	2 GB
# items stored (million)						
LRU+chaining	0.60	1.20	2.40	4.79	9.59	19.17
LRU+cuckoo	0.65	1.29	2.58	5.16	10.32	20.65
CLOCK+cuckoo	**0.84**	**1.68**	**3.35**	**6.71**	**13.42**	**26.84**
hit ratio (95% `GET`, 5% `SET`, zipf distribution)						
LRU+chaining	63.66%	68.09%	72.73%	77.70%	83.20%	89.56%
LRU+cuckoo	**64.13%**	**68.58%**	**73.24%**	78.26%	83.84%	90.20%
CLOCK+cuckoo	63.54%	68.14%	73.08%	**78.62%**	**85.32%**	**92.11%**

Table 3: Comparison of three types of caches. Results in this table is independent of the key-value size. Bolded entries are the best in their columns. Each data point is the average of 10 runs.

Space Efficiency Table 3 compares the maximum number of items (16-Byte key and 32-Byte value) a cache can store. The default LRU with chaining is the least memory efficient scheme. Replacing chaining with cuckoo hashing improves the space utilization slightly (7%), noting that the hash table size is not counted in the cache, and the small improvement is due to the pointer eliminated for chaining. Combining CLOCK with cuckoo increases the space efficiency by 30% over the default. The space benefit arises from eliminating three pointers (two used for LRU linked list and one used for chaining) and one refcount per key.

Hit ratio Compared to the linked list based approach in Memcached, CLOCK approximates LRU eviction with much lower space overhead. Table 3 shows the cache hit ratios measured for `GET` queries by three different configures. When the cache size is smaller than 256 MB, LRU based cache provides higher hit ratio than CLOCK. LRU with cuckoo hashing improves upon LRU with chaining, because it can store more items. In this experiment, 256MB is only about 2.6% of the 10GB working set. Therefore LRU wins only if the cache size is very small, causing popular items to have too large a chance to be evicted. For larger cache, 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% `GET`s and 5% `SET`s generated by YCSB. MemC3 server runs on the same server as before, but the clients are 50 different nodes connected by a 10GB Ethernet. The clients uses libmemcached 1.0.7 [18] to communicate with our MemC3 server over the network. To amortize the network overhead, we use multi-get supported by libmemcached by batching 100 `GET`s.

In this experiment, we compare three different systems: original Memcached, optimized Memcached (with non-algorithmic optimizations such as "hugepage", "in keycmp" and tuned CPU affinity), and MemC3 with all optimizations enabled. Each system is allocated with 1GB memory space (hash table space not included).

Throughput Figure 8 shows the throughput as more server threads are used in MemC3. Overall, the maximum throughput of MemC3 (4.4 MOPS) is almost 3× that of the original Memcached (1.5

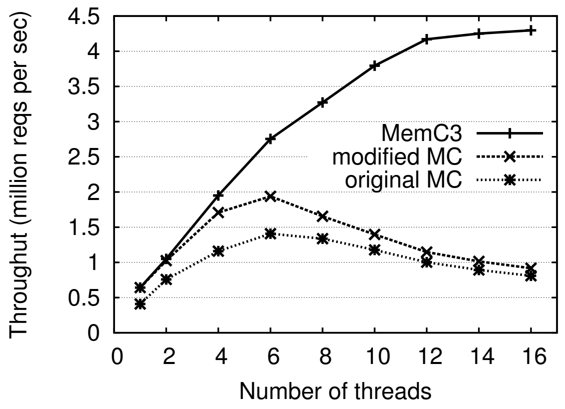


Figure 8: Throughput v.s. number of server threads

MOPS). The non-algorithmic optimizations improve throughput, but their contribution is dwarfed by the algorithmic and data structure-based improvements.

6 Related Work

This section presents the work most related to MemC3 in three parts: efforts to improve individual key-value storage nodes in terms of throughput and space efficiency; the broader context of systems where our techniques might apply; and the related work applying cuckoo hashing.

Flash-based key-value stores such as BufferHash [1], FAWN-DS [2], SkimpyStash [9] and SILT [19] are optimized to better utilize 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, 22, 15] speed the performance on multi-core CPUs or GP-GPUs by sharding data to dedicated cores and thereby avoid synchronization. MemC3 instead targets read-mostly workloads and thus applies optimistic locking to boost performance. Similar to MemC3, Mastree [20] 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 [24] 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.

Recent *distributed key-value systems* such as Google's Bigtable [5], Amazon's Dynamo [10], Apache's Cassandra [17], FAWN-KV [2] and Ramcloud [24] aimed primarily for high availability, scalability and load balancing among a large number of storage nodes. Our system focuses on boosting the performance on a single multi-core server. MemC3 could serve as a building block for large scale distributed key-value systems.

Cuckoo hashing [25] 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 [8] applied cuckoo hashing by assigning each item 16 locations so that each lookup checks up to 16 locations. SILT [19] proposed *partial key cuckoo hashing* to achieve high occupancy with only two hash functions, but in doing so, limited the maximum hash table size. Our improved algorithm eliminates this limitation while still retaining high memory efficiency. To make cuckoo operations concurrent, the prior approach of Herlihy and et. al. [14] traded space for concurrency. In contrast, our optimistic locking scheme achieves high read concurrency without losing space efficiency.

7 Conclusion

MemC3 is an in-memory key-value store that is designed to provide caching service for read-mostly workloads. It is built on carefully engineered algorithms and data structures with a set of architecture-aware and workload-ware 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 a 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 stores 30% more key-value pairs.

References

[1] A. Anand, C. Muthukrishnan, S. Kappes, A. Akella, and S. Nath. Cheap and large CAMs for high performance data-intensive networked systems.

[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.

[3] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload Analysis of a Large-Scale Key-Value Store.

[4] M. Berezecki, E. Frachtenberg, M. Paleczny, and K. Steele. Many-core key-value store.

[5] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A Distributed Storage System for Structured Data.

[6] B. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking Cloud Serving Systems with YCSB.

[7] F. Corbato and M. I. O. T. C. P. MAC. *A Paging Experiment with the Multics System*. Defense Technical Information Center.

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

[9] B. Debnath, S. Sengupta, and J. Li. SkimpyStash: RAM Space Skimpy Key-Value Store on Flash.

[10] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's Highly Available Key-Value Store.

[11] U. Erlingsson, M. Manasse, and F. Mcsherry. A COOL AND PRACTICAL ALTERNATIVE TO TRADITIONAL HASH TABLES. Pages 1–6.

[12] Scaling memcached at Facebook.

[13] N. Gunther, S. Subramanyam, and S. Parvu. Hidden Scalability Gotchas in Memcached and Friends.

[14] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann Publishers Inc.

[15] 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.

[16] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, **6**(2):213–226. ACM.

[17] A. Lakshman and P. Malik. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating System Review*, **44**:35-40.

[18] libMemcached.

[19] H. Lim, B. Fan, D. G. Andersen, and M. Kaminsky. SILT: A Memory-Efficient, High-Performance Key-Value Store.

[20] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. Pages 183–196.

[21] A distributed memory object caching system.

[22] Z. Metreveli, N. Zeldovich, and M. F. Kaashoek. CPHASH: a cache-partitioned hash table.

[23]

[24] D. Ongaro, S. M. Rumble, R. Stutsman, J. Ousterhout, and M. Rosenblum. Fast crash recovery in RAMCloud.

[25] R. Pagh and F. Rodler. Cuckoo Hashing. *Journal of Algorithms*, **51**(2):122–144.

[26] N. Provos. libevent.

[27]

[28] VoltDB, the NewSQL database for high velocity applications.