Improved hashing in Memcached using the Hopscotch algorithm

Suraj Dharmapuram
Carnegie Mellon University
sdharmap

Udbhav Prasad
Carnegie Mellon University
udbhavp

Swapnil Pimpale
Carnegie Mellon University
spimpale

Abstract

In this paper we present a modified version of Memcached which uses Hopscotch hashing technique, optimistic locking and the CLOCK algorithm. These techniques combined together enable the resulting system to perform better than the original Memcached. Although these modifications are implemented on top of Memcached we believe that they apply more generally to many of today's read-intensive caching systems.

1 Introduction

Hash tables are fundamental data structures that implement an association between a key and a value. Provided a good hash function, hash tables provide the ability to lookup or insert a key in constant time. Given the ubiquitous nature of their usage in modern computing environments, any improvement in their performance is likely to have wide ranging impact. Recent work [6] has proposed an improved, resizable hashing algorithm targeted at both uniprocessor and multiprocessor machines. This algorithm is based on a novel hopscotch multi-phased probing and displacement technique that has the flavors of chaining, cuckoo-hashing and linear probing. The resulting algorithm provides a hash table with very low synchronization overheads and high cache hit ratios. One of the most interesting features of the hopscotch algorithm is that it continues to deliver good performance even when the table is more than 90% full, increasing its advantage over other algorithms as the table density grows. In this project, we aim to demonstrate the effectiveness of this new hopscotch based hashing algorithm on a production workload. In addition, we also aim to demonstrate the effectiveness of improved algorithm and data structure design while incorporating the hopscotch algorithm. As a case study, we focus on Memcached [5].

Memcached is a high performance, distributed in-memory caching system. Standard Memcached uses a typical hash table design, with linked list based chaining to handle collisions. Its cache eviction policy is strict LRU, also based on linked lists. This design relies on coarse-grained locking to ensure consistency among multiple threads, and leads to poor scalability on multi-core CPUs. Additionally, to maintain LRU state, Memcached stores a significant amount of per-key metadata - the amount of memory consumed by the metadata is often significantly higher than that consumed by the keys themselves for many workloads. In this project, we will focus on redesigning key internal components of Memcached to be more efficient in a multiprocessor environment when deployed with the Hopscotch algorithm.

2 Related work

MemC3[4] is a system that incorporates algorithmic and engineering improvements to the Memcached system with the aim of improving memory efficiency and throughput. These techniques include - optimistic cuckoo hashing [7, 10], an approximate LRU algorithm, and comprehensive implementation of optimistic locking. In particular, their design exploits CPU cache locality to minimize the number of memory fetches required to complete any operation, and exploits instruction level and memory level parallelism to overlap these fetches. In addition, MemC3s design also exploits workload characteristics. Many Memcached workloads are predominantly read, with a few writes. Hence in MemC3, Memcacheds expensive global lock is replaced with an optimistic locking scheme that makes the common case go fast. Exploiting the fact that many common Memcached workloads target very small objects, MemC3 significantly reduces the amount of state stored per key and replaces the strict LRU replacement policy with a CLOCK[3] based approximate LRU replacement.

3 Hopscotch hashing

Hopscotch hashing is a new type of open-addressable resizable hash table that is directed at cache sensitive machines, a class that includes most, if not all of the state-of-the-art uniprocessors and multicore machines. It provides a `contains()` method that runs in deterministic constant time and required only two cache loads.

3.1 Prevalent hashing schemes

Chained hashing is a closed address hashing scheme consisting of an array of buckets each of which holds a linked list of items. Though this approach is more competent than other approaches in terms of the time it takes to locate an item, its use of dynamic memory allocation and indirection makes for poor memory management. This approach is even more expensive in a concurrent environment, as dynamic memory management typically requires a thread-safe memory manager or garbage collector - this adds considerable overhead in a concurrent environment.

Linear Probing is an open-addressable hashing scheme in which items are kept in a contiguous array, each entry of which is a bucket for one item. A new item is inserted by hashing a key to a particular entry, and scanning forward from that entry until an empty bucket is located. Lookup proceeds in a similar manner. Because the array is accessed sequentially, it has good cache locality, as each cache line can hold multiple entries of the array. Unfortunately, linear probing has inherent limitations: because every `contains()` call searches linearly for the key, performance degrades as the table fills up.

Cuckoo hashing [10] is an open-addressed scheme that unlike linear probing requires only a deterministic constant number of steps to locate an item. Cuckoo hashing uses two hash functions. A new item x is inserted by hashing the item to two array indexes. If either slot is empty, x is added there. If both are full, one of the occupants is displaced by the new item. The displaced item is then reinserted using its other hash function, possibly displacing another item, and so on. If the chain of displacements grows too long, the table is resized. A disadvantage of cuckoo hashing is the need to access sequences of unrelated locations on different cache lines. Another disadvantage is that Cuckoo hashing tends to perform poorly when the table is more than 50% full because displacement sequences become too long, and the table needs to be resized.

3.2 Hopscotch hashing details

The Hopscotch idea combines the best features of the *cuckoo*, *chaining* and *probing* schemes in the following way. There is a single hash function *h*. The item that is hashed using *h* to a particular hash entry will either be found at that location or in one of the next *(H - 1)* entries. In our implementation, H is a constant, 32, that is the standard machine word size. In other words, a virtual bucket has a fixed size and overlaps with the next *(H - 1)* buckets. Each entry in the hashtable includes a *hop information bitmap*, that indicates which of the next H - 1 entries were hashed to the current entry's virtual bucket. In this way, an item can be found quickly by looking at the word to see which entries belong to the bucket, and then scanning through the constant number of entries.

The insert algorithm for the case where item *x* hashes to location *i*, works as follows:

1. Starting at item *i*, use linear probing to find an empty slot at index *j*.

2. If index *j* is within *(H - 1)* of i, place *x* there and return.

3. Otherwise, j is too far from i. To create an empty entry closer to *i*, find an item y whose hash value lies between *i* and *j*, but within *(H - 1)* of *j*, and whose entry lies below *j*. Displacing y to *j* creates a new empty slot closer to *i*. Repeat till *j* is finally displaced to a location that is within the neighborhood of *i*. If no such item exists, or if the bucket already *i* contains H items, resize and rehash the table.

   In other words, the idea is that hopscotch moves the empty slot towards the desired bucket instead of leaving it where it was found as in linear probing, or moving an item out of the desired bucket and only then trying to find it a new place as in cuckoo hashing. The cuckoo hashing sequence of displacements can be cyclic, so implementations typically abort and resize if the chain of displacements becomes too long. As a result, cuckoo hashing works best when the table is less than 50% full. In hopscotch hashing, by contrast, the sequence of displacements cannot be cyclic: either the empty slot moves closer to the new items hash value, or no such move is possible. As a result, hopscotch hashing supports significantly higher loads.

   Hopscotch has the advantage of buckets with multiple items, but unlike in chaining they have great locality since they are located in neighboring memory locations. It also inserts elements in expected constant time as in linear probing, but with the guarantee that items are always found in their buckets in deterministic constant time.

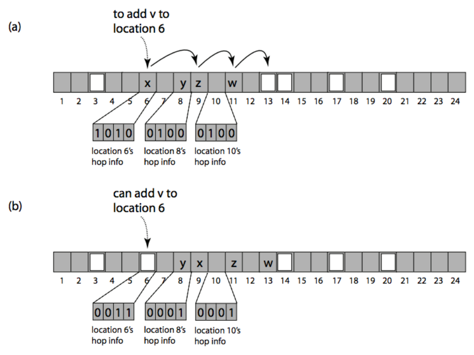

Figure 1: This figure[6] illustrates the insert process in the Hopscotch algorithm when there is no free slot with the neighborhood. Element v was hashed to location 6 but there was only a free slot at position 13. Element at index 11 had to be displaced to element at index 13, element at index 9 was displaced to index 11, element at index 6 was moved over to the element at index 9 and finally the element v could be placed at the now-empty location 6. Note that all these transformations respect the rule that an element must be found with H locations of its original location.

4 Memcached internals

Memcached has a client server architecture where clients communicate with servers over the network using a simple *GET/SET/DELETE* interface. It is an in-memory key-value store for small chunks of arbitrary data (strings, objects) from results of database calls, API calls, or page rendering.

Internally, Memcached uses a hashtable to index (key, value) tuples. These entries are also in a linked list sorted by their most recent access time. The least recently used entry is evicted and replaced by a new entry when the hashtable is full.

Memcached resolves collisions by chaining entries: if more than one entry maps to the same hash table index, they form a linked list. Chaining is efficient for inserting/deleting single keys. However, lookup may require scanning the entire chain.

Naive memory allocation(malloc/free) could result in significant memory allocation. To address this problem, Memcached uses slab-based memory allocation. Memory is divided into 1MB pages, and each page is further divided into fixed size chunks. Key-value pairs are stored in an appropriately sized chunk. The size of the chunk, and thus, the number of chunks in the slab class depends on the particular slab class. To insert a new key, Memcache looks up the slab class whose size best fits the key to be inserted.

Each slab class maintains the items in a strict LRU based order. Figure 3 demonstrates the architecture of a slab class using a doubly linked list. Memcached maintains an array of LRU head and tail pointers that help maintain the LRU nature. The Least Recently Used item is maintained at the tail of the list belonging to a particular slab class. Upon every access, an item is moved to the head of the corresponding slab list.

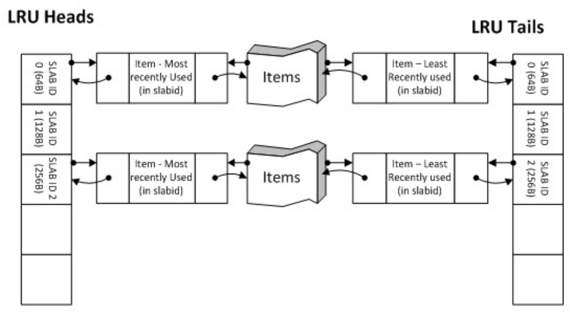
LRU Heads
LRU Tails
SLAB ID 0 (64B)
SLAB ID 1 (128B)
SLAB ID 2 (256B)
Item - Most recently Used (in slabid)
Items
Item – Least Recently used (in slabid)

Figure 2: Memcached slab-based LRU [9]

There are two main components inside Memcached (as shown in Figure 3) - the item cache where the items, i.e (key, value) tuples are actually stored and the hashtable itself, which serves as an index into the cache. Thus, each item is a part of a linked list in the bucket to which it was hashed to, and the LRU list of the slab to which it belongs - this accounts for a total of 3 pointers that Memcache must maintain for each of the items that it stores.

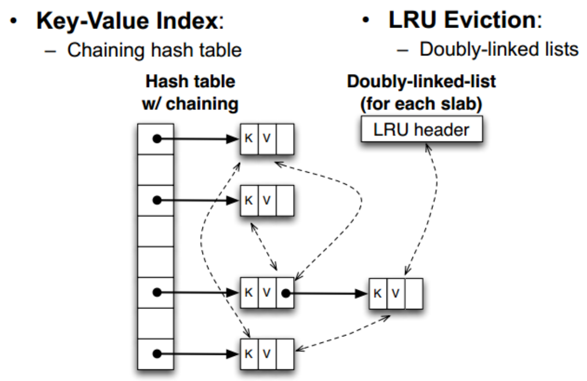

Figure 3: Memcached core data structures [8]

5 Real life workloads

It has been observed that in real workloads [1] are often read-heavy. In general, a GET/SET ratio of 30:1 has been reported for the Memcached workloads in Facebook. Also in real workloads, queries for small objects dominate. Most keys are smaller than a few bytes and most values are not larger than a few hundred bytes.

Memcached however, does not take these factors into account in its current design. Though most queries are GETs, this operation is not optimized, and locks are used extensively on the query path. Each GET operation must acquire a lock for exclusive access to a particular key and after reading the value, it must get a global lock to update the LRU linked list.

In the following sections, we demonstrate how we have tried to remove locks from the query path of READ operations. We remove the usage of locks on the read path by the use of optimistic locking. Since reads dominate writes, we assume there is only a single writer thread in the system, so there can never be writer-writer interleaving. The following section(s) detail the implementation of these ideas.

6 Optimistic Locking

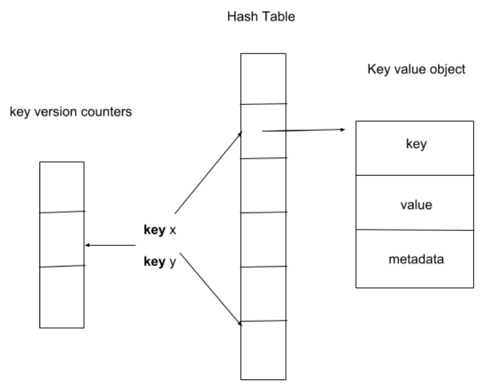

Figure 4: Hash Table Overview: The hash table is an array of buckets where each bucket points into a key-value object. Each key in the hash table is associated with one key version counter. As shown in the figure, more than one key could be associated with the same version counter.

In our implementation, we need to ensure that inserts are atomic with respect to lookups. Since we assume a single writer in our implementation, we only need to worry about the interleaving of an insert operation with a lookup operation. All other interleavings (insert-insert, lookup-lookup) are either impossible or non-problematic. Memcached protects against such interleavings using coarse grained locks. It employs locks at the hash table level as well as locks per-slab. This locking approach is straightforward, intuitive and easy to implement but limits scalability.

In our implementation, we take the advantage of having a single writer to synchronize *Insert* and *Lookups* with low overhead. Instead of taking locks, we assign a version counter for every key in the hash table, update the version on *Insert* and look for a change during *Lookup* to detect any concurrent update.

We do not store a version counter for every key in the hash table because that would take space proportional to the number of keys in the hash table which could be millions. Also, this approach could lead to a race condition: to check or update the version of a given key, we must first lookup the key in the hash table to find the key-value object and this initial lookup is not protected by any lock and hence not thread-safe. Instead, we have a fixed size array of version counters. Each key maps to one of the version counter in this array. In our implementation, the size of this array is 8192 i.e., we allocate space for 8192 version counters (32 KB) irrespective of the number of keys in the hash table. An array of this size can conveniently fit in the processor caches. The number of keys in the hash table will, more often than not, be higher than 8192 which means that multiple keys will share a version counter (as shown in Figure 4). This, in turn, means that an update operation on one key could stall a lookup operation on another key if they happen to share the same version counter. This approach does limit concurrency to some extent but it can be seen that the chance of “false retry” (re-reading a key due to modification of an unrelated key) is roughly about 0.01%. [4]

The actual optimistic locking happens as follows. Before updating a key, the *Insert* process increments the version counter for the key. This indicates to the other *Lookups* that there is an on-going update operation. After the update is done the version counter is again incremented by one to indicate completion. As for *Lookups*, it first snapshots the version for that key. If the version is odd it knows that a concurrent update is going on and retries itself. If the version is even, the *Lookup* operation proceeds to completion. After it finishes reading, it snapshots the counter again and compares the new version with the old one. If the two versions differ, the writer must have modified the version and the *Lookup* should be retried.

7 CLOCK Algorithm

Memcached uses a strict LRU policy for eviction of items from the cache. Using a strict LRU eviction policy has two main problems. First, two pointers (next, prev) are required to maintain the items in the per-slab doubly-linked LRU list. When the items are very small in size, this becomes a major source of space overhead. Second, all updates to a particular LRU list must be serialized and hence this is a synchronization bottleneck.

The CLOCK[3] algorithm helps make the cache management efficient and concurrent. It is a mechanism to implement an approximate LRU instead of a strict LRU. The space saved by replacing pointers with bit entries allows the cache to store more entries which in turn improves the hit ratio.

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

1. *Update* to keep track of the recency after querying a key in the cache

2. *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 at 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.

In the CLOCK algorithm, there is a *circular buffer* and a *virtual hand* for each slab class. A bit in the buffer represents the recency for that item. A value of 1 means that the item is recent, otherwise it is not. Each *Update* simply sets the recency bit to 1. *Evict* checks the bit currently pointed to by the hand. If the bit is 0, the corresponding item is selected for eviction. If the bit is 1, it is reset to 0 and the hand is advanced in the buffer until a bit of 0 is encountered.

8 Evaluation

In this section we take a look at the effects that the proposed techniques and optimizations have had on performance and space efficiency. We start our investigation with the cache system, then move on to the evaluation of the hash table, followed by analysis of the entire system.

8.1 Platform

For all experiments we use an Amazon Web Services (AWS) Elastic Compute Cloud instance. Since the machine will spend most of its time performing memory reads and writes, we decided on an `r3.xlarge` EC2 instance. It has the following configuration

CPU	Intel Xeon E5-2670 v2 @ 2.50GHz
Number of cores	4
DRAM	30.5 GB

Amazon recommends `R3` instances for high performance databases, distributed memory caches and in-memory analytics.

8.2 Workload data

We use YCSB [2] to generate 10 million key-value queries, following a Zipf distribution. Many types of data studied in the physical and social sciences can be approximated with a Zipfian distribution, one of a family of related discrete power law probability distributions.

We generate two workloads using YCSB: a read-only workload, where we insert 10 million elements into the hash table (the load stage) and then perform 10 million read-only queries on it, and a read-mostly workload, where we insert 10 million elements into the hash table (the load stage) and then run 10 million queries on it, 95% of which are reads. We shall refer to these two as Workload C and Workload B respectively for the rest of this paper.

Our motivation behind Workload C (read-only) was to show the advantages of optimistic locking in a read-only scenario, where no locks need to be taken at all. Workload B was chose because it was representative of the most common balance between reads and writes seen in the “real world”.

8.3 Cache Microbenchmark

Each key is 16 bytes and each value is 32 bytes. We vary the cache size from 64MB to 512MB. This cache size parameter does not include the space occupied by the hash table, only the space used to store the item object.

8.3.1 Space efficiency

Table 1 shows how the maximum number of items (16-byte key and 32-byte value) a cache can store given the different cache sizes. The space to store the index hash tables is separate from the given cache space in Table 1. We set the hash table capacity larger than the maximum number of items that the cache space can possibly store.

We observe a linear scaling in the number of items that can be stored with increase in cache size. Both MemC3 and Hopscotch have better space efficiency than Memcached. Memcached incurs a considerable overhead even for small key-value pairs. It always allocates 56 bytes regardless of item size. This is because of the number of pointers that it has to keep track of: two pointers (`next` and `prev` maintaining the LRU) and one for the hash table itself. MemC3 and our Hopscotch implementation remove these pointers in replacing strict LRU with approximate LRU or the CLOCK algorithm.

The space that is saved per item means that more items can be stored in the same amount of cache. Since we follow the same slab and cache architecture as MemC3, and since (as mentioned above), our experiment is bounded by the cache size and not by the hash table size, our cache implementation with Hopscotch has the same number of items as MemC3.

Table 1: Comparison of number of items stored as cache size increases

	Number of items stored (in millions)		
Cache Size (in MB)	Memcached	MemC3	Hopscotch
64	0.56	0.64	0.64
128	1.11	1.29	1.29
256	2.22	2.58	2.58
512	4.47	5.16	5.16

8.3.2 Cache hit ratio

Table 2 compares the hit ratios of Memcached, MemC3 and Hopscotch. As expected from the previous table, showing the maximum number of items stored for each cache size, the hit ratios increase with cache sizes for all three systems. However, it is more interesting to compare the relative hit ratios between the three systems. Predictably, MemC3 performs better than Memcached. We can also see that our modified Hopscotch performs at par with or better than Memcached, but it still has lesser hit ratios than MemC3 for all sizes.

MemC3 makes several optimizations in its system, one of which is to enable "hugepages" in Linux. Hugepages is a mechanism that allows the Linux kernel to utilize the multiple page size capabilities of modern hardware architectures. Linux uses pages as the basic unit of memory, where physical memory is partitioned and accessed using the basic page unit. The default page size is 4096 Bytes in the x86 architecture. Hugepages allows large amounts of memory to be utilized with a reduced overhead. Linux uses Translation Lookaside Buffers (TLB) in the CPU architecture. These buffers contain mappings of virtual memory to actual physical memory addresses. So utilizing a huge amount of physical memory with the default page size consumes the TLB and adds processing overhead.

The Linux kernel is able to set aside a portion of physical memory to be able be addressed using a larger page size. Since the page size is higher, there will be less overhead managing the pages with the TLB. Systems with large amount of memory can be configured to utilize the memory more efficiently by setting aside a portion dedicated for hugepages.

We suspect that this optimization is responsible to the improvement in hit ratio that MemC3 is seeing over our implementation. We intend to add this optimization to our implementation and compare the results in the future.

Table 2: Comparison of hit ratios as cache size increases

	Hit ratio		
Cache Size (in MB)	Memcached	MemC3	Hopscotch
64	0.4956	0.5104	0.4941
128	0.5553	0.5668	0.5516
256	0.6296	0.6453	0.6345
512	0.7408	0.7722	0.7643

8.4 Hash Table Microbenchmark

In this subsection we investigate the lookup performance of a single thread and the aggregate throughput of a varying number of threads all accessing the same hash table. The hash tables are linked into a workload generator directly and benchmarked on a local machine.

8.4.1 Multithreaded scalability

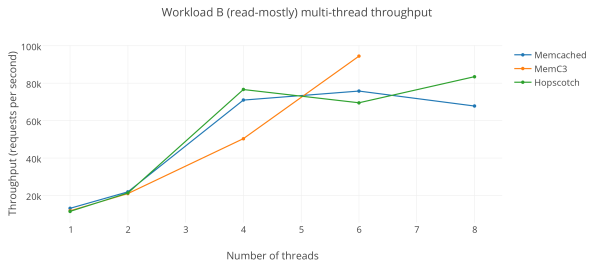

Figure 5: Workload B multithreaded throughput

Figure 5 and 6 show the results of running Workload B and C on Memcached, MemC3 and our implementation of Hopscotch for 1, 2, 4, 6 and 8 threads.

Recall that our `r3.xlarge` system has 4 cores. This means that we should expect the throughput to scale upto 4 threads. This is what we see in both Figure 5 and Figure 6. We see that the throughput scales roughly linearly upto 4 threads. Beyond 4 threads we see that Memcached and Hopscotch plateau.

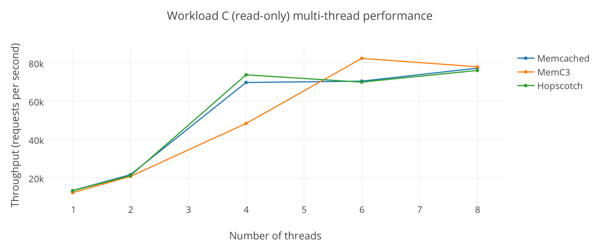

Figure 6: Workload C multithreaded throughput

Interestingly, MemC3 continues to scale to 6 threads. Again, the MemC3 paper mentions CPU-affinity and scheduling optimizations that it makes while handling multiple requests. Again, as part of our future work, we intend to implement this optimization in our version of Hopscotch and compare it against MemC3.

It is important to note that Figure 5 and 6 are only intended as a display of how throughput scales with the number of threads. They are not meant as a comparison of throughput between the three systems. Although the data points we have plotted are the average of three runs in each configuration, the amount of variance we saw in the throughput was high. For example, in Workload C, for 6 threads, MemC3 showed a minimum of 68946.88 requests per second and a maximum of 89911.20 requests per second. We think that AWS is to blame for this large variance. AWS is a multi-tenant system, and we do not have complete control of the hardware (in fact, a request for an instance returns a virtual machine, not a dedicated bare machine).

It will be more informative if we ran on dedicated machines, so that we could record a series of data with low variance. This will give us a better platform to compare the throughputs of Memcached, MemC3 and our version of Hopscotch.

9 Conclusion

We modified Memcached to use the hopscotch hashing technique, optimistic locking for high concurrency and CLOCK-based cache management with only 1-bit per cache entry to approximate LRU eviction. Our evaluation shows that we achieve performance (both throughput and hit ratio) as good as Memcached or even better. The consistent hit ratios observed with different kinds of workloads for multiple runs of the same experiment gives us confidence about the correctness of our implementation. The main motivation for our project was the significant improvement of MemC3 over original Memcached as claimed by the authors of Memc3. Our experiments show performance improvement with both hopscotch and Memc3 over original Memcached but not as high as mentioned in the Memc3 paper.

10 Future Work

We believe our work could be extended both in terms of implementation and evaluation. As for implementation, optimizations employed by MemC3 (like enabling hugepage support, replacing memcmp with integer-based comparison, etc.) could be tried out. In terms of evaluation, we believe that there is a lot of scope for performing extensive scalability testing on multicore processors. This will give us a better idea as to how much performance improvement could be achieved using the optimistic locking approach. Also, both the above changes (namely optimistic locking and CLOCK algorithm) could be applied incrementally and their effects could be profiled individually. The latest version of our code can be found here `https://github.com/surajdharmapuram/memcached`

References

[1] Berk Atikoglu et al. “Workload analysis of a large-scale key-value store”. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 40. 1. ACM. 2012, pp. 53–64.

[2] Brian F Cooper et al. “Benchmarking cloud serving systems with YCSB”. In: *Proceedings of the 1st ACM symposium on Cloud computing*. ACM. 2010, pp. 143–154.

[3] Fernando J Corbato. *A paging experiment with the multics system*. Tech. rep. DTIC Document, 1968.

[4] Bin Fan, David G Andersen, and Michael Kaminsky. “MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing.” In: *NSDI*. Vol. 13. 2013, pp. 385–398.

[5] Brad Fitzpatrick. “Distributed caching with memcached”. In: *Linux journal* 2004.124 (2004), p. 5.

[6] Maurice Herlihy, Nir Shavit, and Moran Tzafrir. “Hopscotch hashing”. In: *Distributed Computing*. Springer, 2008, pp. 350–364.

[7] Hsiang-Tsung Kung and John T Robinson. “On optimistic methods for concurrency control”. In: *ACM Transactions on Database Systems (TODS)* 6.2 (1981), pp. 213–226.

[8] *Memcached chaining*. `https://www.usenix.org/sites/default/files/conference/protected-files/fan_nsdi13_slides.pdf`.

[9] *Memcached slabs*. `https://software.intel.com/sites/default/files/m/a/3/2/a/0/45646-f-4.jpg`.

[10] Rasmus Pagh and Flemming Friche Rodler. "Cuckoo hashing". In: *Journal of Algorithms* 51.2 (2004), pp. 122–144.