International Journal of Computer Applications (0975 – 8887)
Volume 186 – No.64, January 2025

A High-Performance Memcached Implementation using Hopscotch Hashing and Lock-Free Techniques

Udbhav Prasad Suraj Dharmapuram

ABSTRACT
This paper presents 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, they apply more generally to many of today's read-intensive caching systems.

Keywords
Hopscotch Hashing, Optimistic Locking, Distributed Caching, CLOCK Algorithm, Memory Efficiency

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 a wide ranging impact.

Hopscotch hashing [6] is a recent proposal for a resizable hashing algorithm targeted at both uniprocessor and multiprocessor machines. This algorithm describes a multi-phased technique that incorporates chaining, cuckoo-hashing and linear probing. This results in a hash table data structure with high cache hit ratios and low locking overheads. The proposed approach also improves upon the performance of existing approaches at high hash table densities, delivering good performance even when the table is over 90% full.

This project aims to demonstrate the effectiveness of this new hopscotch based hashing algorithm on a production workload. In addition, it also aims to demonstrate the effectiveness of improved algorithm and data structure design while incorporating the hopscotch algorithm. As a case study, it focuses on Memcached [5].

Memcached is a high performance, distributed in-memory caching system. Memcached implements a hash table data structure, handling collisions and LRU cache eviction using linked lists. The design is fairly standard, relying on coarse-grained locking to ensure consistency between multiple processes in a multi-core CPU environment. 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. This project focuses 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] proposes improvements to the Memcached system with the use of techniques like optimistic cuckoo hashing [7, 10] – an approximate LRU algorithm – and comprehensive implementation of optimistic locking. improving memory efficiency and throughput.

MemC3's design takes advantage of cache locality, instruction pipelining and general workload characteristics. Many Memcached workloads are predominantly read, with a few writes. Hence in MemC3, Memcached's 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 requires only two cache loads.

3.1 Prevalent hashing schemes
Chained hashing consists of an array of buckets containing a linked list of items. Though this approach is more competent than other approaches regarding 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 this repeats going forward. If the chain of displacements grows too long, the table is resized. A disadvantage of cuckoo hashing is accessing sequences of unrelated locations on different cache lines. Additionally, Cuckoo hashing performance degrades when the table is more than 50% full, forcing frequent table resizes for growing cache data.

3.2 Hopscotch hashing details

The Hopscotch algorithm integrates advantageous characteristics of cuckoo hashing, chaining, and linear probing through a unique approach. It employs a singular hash function, denoted as *h*. An element subjected to hashing via *h* to a specific hash entry will be located at that precise location or within one of the subsequent (H - 1) entries. In the present implementation, H is a constant value of 32, corresponding to the standard machine word size. Consequently, a virtual bucket possesses a predetermined size and exhibits overlap with the succeeding (H - 1) buckets. Each entry within the hash table incorporates a hop-information bitmap, which delineates which of the ensuing (H - 1) entries were hashed to the virtual bucket associated with the current entry. This mechanism facilitates the rapid retrieval of an element by examining the bitmap to ascertain which entries belong to the bucket, followed by a sequential scan through the fixed number of entries.

In the scenario where element x hashes to location *I*, the insertion algorithm operates as follows: commencing at entry *I*, linear probing is employed to locate an unoccupied slot at index j. If index *j* resides within (H - 1) positions of *i*, *x* is placed at that location, and the algorithm terminates. Otherwise, *j* is deemed excessively distant from *i*. To generate an unoccupied entry closer to *i*, an element *y* is sought whose hash value falls between *i* and *j*, yet remains within (H - 1) positions of *j* and whose entry lies beyond *j*. Displacing *y* to *j* creates a new unoccupied slot nearer to *i*. This iterative process continues until *j* is ultimately displaced to a location that falls within the neighborhood of *i*. Should no such element *y* exist, or if bucket *i* already contains H elements; the table is resized and rehashed. Essentially, the Hopscotch method endeavors to move the vacant slot towards the target bucket, in contrast to the approach of linear probing, where the slot remains at its discovered location, or cuckoo hashing, where an element is displaced from the target bucket prior to seeking an alternative location. The displacement sequence in cuckoo hashing can exhibit cyclical behavior, necessitating implementations to typically abort and resize if the chain of displacements becomes inordinately long. Consequently, cuckoo hashing demonstrates optimal performance when table occupancy remains below 50%. Conversely, in Hopscotch hashing, the displacement sequence cannot form cycles; either the vacant slot is moved closer to the hash value of the new element, or such movement is unfeasible. As a consequence, Hopscotch hashing supports considerably higher load factors.

Hopscotch hashing possesses the advantage of accommodating multiple elements within buckets. However, unlike chaining, it maintains exceptional locality, as elements are situated in contiguous memory locations. Furthermore, it facilitates element insertion in expected constant time, similar to linear probing, while simultaneously guaranteeing that elements are invariably found within their designated buckets in deterministic constant time.

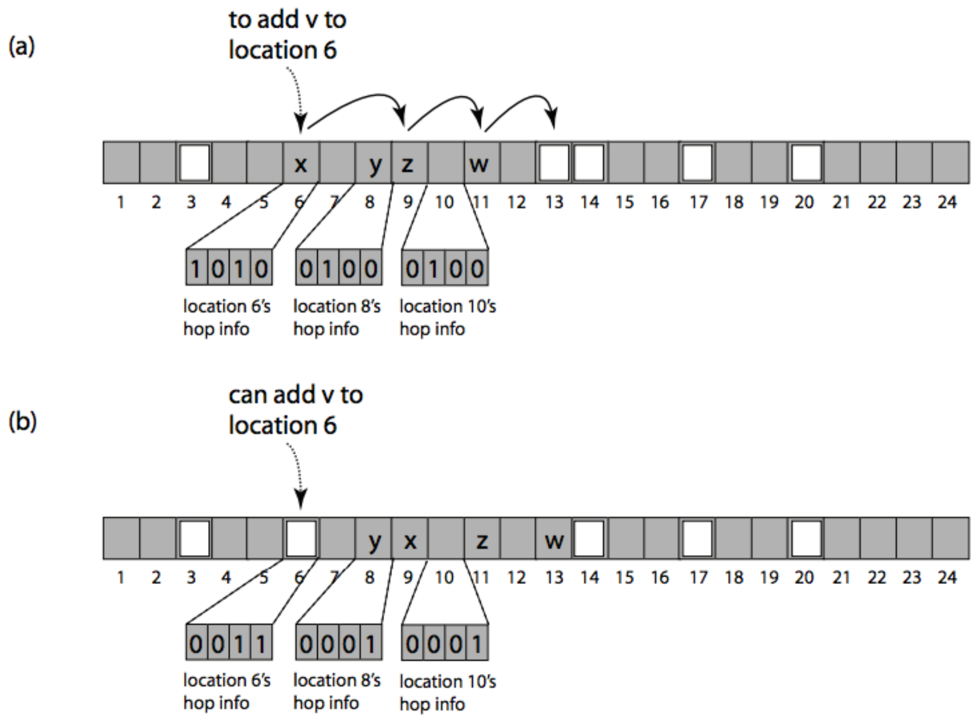

Figure 1: This figure [6] illustrates the insert process in the Hopscotch algorithm when there is no free slot with the neighborhood. Element *v* was hashed to location 6 but there was only a free slot at position 13. Element at index 11 had to be displaced to element at index 13, element at index 9 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 in which clients communicate with servers over the network using a simple *GET/SET/DELETE* interface.

As an in-memory key-value cache, it houses arbitrary data fragments, encompassing strings and objects, produced from database operations, API requests, and page rendering processes.

Internally, Memcached uses a hashtable to index (key, value) tuples. Collisions are stored in a linked list in descending order of their access times. This facilitates the implementation of its least-recently-used (LRU) cache eviction policy. Chaining collisions in a linked list is efficient for updating single keys, but if a large number of keys collide, it could increase lookup times.

Memcached uses slab-based memory allocation. Memory is divided into 1MB pages, each further divided into fixed-size chunks. Key-value pairs are stored in an appropriately sized chunk. The size and number of chunks in the slab class depends on the particular slab class. New keys are inserted into the slab class of appropriate size.

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 keep the LRU nature. The least recently used item is maintained at the tail of the list and belongs to a particular slab class. Upon every access, an item is moved to the head of the corresponding slab list.

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.

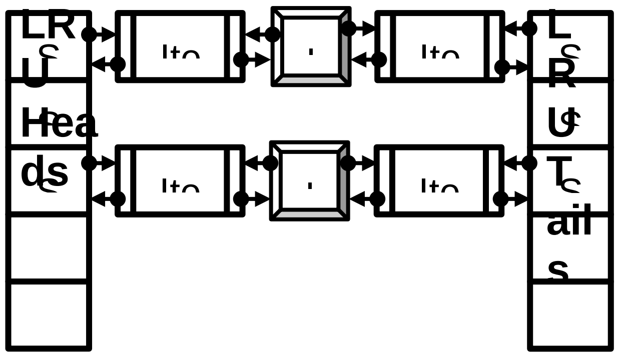

Figure 2: Memcached slab-based LRU [9]

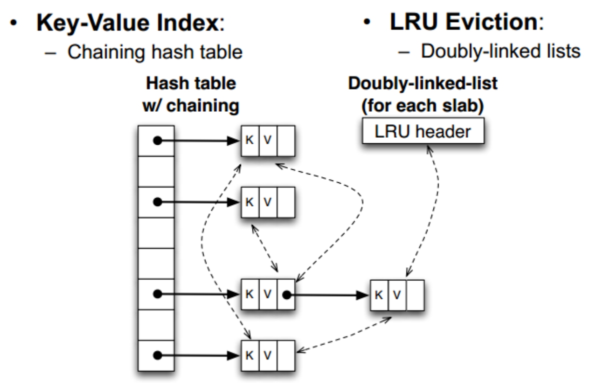

Figure 3: Memcached core data structures [8]

5. REAL LIFE WORKLOADS

Atikoglu et al. [1] have shown that real-life workloads are often read-heavy, and dominated by small objects. However, Memcached does not consider these factors 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.

The following sections demonstrate the techniques employed to remove locks from the query path of READ operations. The use of locks on the read path are removed by using optimistic locking. Since readers dominate writing, it is assumed 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

The implementation ensures that inserts are atomic with respect to lookups. The single writer in the system only needs to consider the interleaving of an insert operation with a lookup operation. All other interleavings (insert-insert, lookup-lookup) are either impossible or non-problematic. This uses a straightforward and intuitive locking mechanism, similar to Memcached's approach, which employs locks at the hash table level and per slab. This simple yet reliable approach, while limiting scalability, provides a robust solution for the system.

The implementation employs a single writer to synchronize *Inserts* and *Lookups* with minimal overhead. To avoid the use of locks, each key in the hash table is associated with a version counter. This counter is updated during insertion operations, and any concurrent modifications are detected by comparing version numbers during lookups. This strategy promotes both efficiency and the preservation of optimal system performance. A version counter for every key in the hash table is not stored, because that would take space proportional to the number of keys in the hash table, which could be millions. This approach could result in a race condition because checking or updating the version of a specific key requires an initial lookup in the hash table to locate the key-value object. However, this initial lookup is not thread-safe as it is not protected by any lock. Instead, a fixed size array of version counters is kept. Each key maps to one of the version counters in this array. In the implementation, the size of this array is 8192 i.e., it allocates 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, meaning 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 ongoing 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.

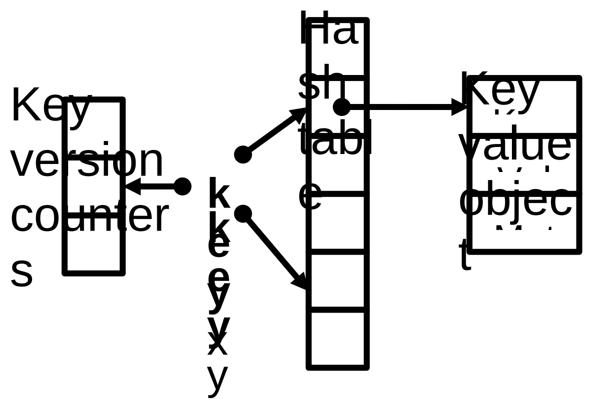

Figure 4: Hash Table Overview: The hash table is an array of buckets where each bucket points into a key-value object. Each key in the hash table is associated with one key version counter. As shown in the figure, more than one key could be associated with the same version counter.

7. CLOCK ALGORITHM

Memcached uses a strict LRU policy for evicting items from the cache. This 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, 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, improving 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 slab class. After each cache query, Update moves the accessed entry to the head of its queue; to free space when the cache is full, Evict replaces the entry at the tail of the queue with 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

This section examines the effects that the proposed techniques and optimizations have had on performance and space efficiency. The investigation starts with the cache system, then evaluates the hash table and analyzes the entire system.

8.1 Platform

All experiments use an Amazon Web Services (AWS) Elastic Compute Cloud instance. Since the machine will spend most of its time performing memory reads and writes, an *r3.xlarge* EC2 instance is used. It has the configuration in **Table 1**.

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

Table 1. Evaluation configuration

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

8.2 Workload Data

A workload of 10 million key-value queries was generated using the YCSB benchmark [2], with the queries following a Zipfian distribution. The Zipfian distribution, which falls under the category of discrete power-law probability distributions, is frequently employed to model empirical data in the physical and social sciences due to its ability to approximate various real-world phenomena.

Two workloads are generated using YCSB: a read-only workload, where 10 million elements are inserted into the hash table (the load stage) and then perform 10 million read-only queries on it, and a read-mostly workload, where 10 million elements are inserted into the hash table (the load stage) and then run 10 million queries on it, 95% of which are read. The rest of this paper refers to these two as Workload C and Workload B, respectively.

The 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 chosen because it represented 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. The cache size ranges 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 2 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 2. The hash table capacity is set to be larger than the maximum number of items that the cache space can possibly store.

A linear scaling is observed 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 this 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 the same slab and cache architecture as MemC3 is followed, and since (as mentioned above), the experiment is bounded by the cache size and not by the hash table size, the cache implementation with Hopscotch has the same number of items as MemC3.

Table 2. 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 3 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, comparing the relative hit ratios between the three systems is more interesting. Predictably, MemC3 performs better than Memcached. The 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 allow 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 can set aside a portion of physical memory to 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 amounts of memory can be configured to utilize the memory more efficiently by setting aside a portion dedicated to huge pages.

This optimization is likely responsible for the improvement in hit ratio that MemC3 is seeing over this paper's implementation, and is planned to be added to the implementation for comparison in the future.

8.4 Hash Table Microbenchmark

8.4.1 Multithreaded scalability

This subsection investigates 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.

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

Recall that the *r3.xlarge* system has 4 cores, so the throughput is expected to scale up to 4 threads. This is observed in both Figure 5 and Figure 6. The throughput scales roughly linearly up to 4 threads. Beyond 4 threads Memcached and Hopscotch plateau.

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, future work will implement this optimization in this paper's 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 plotted are the average of three runs in each configuration, the amount of variance observed 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. The suspicion is that AWS is to blame for this large variance. AWS is a multi-tenant system, and users do not have complete control of the hardware (in fact, a request for an instance returns a virtual machine, not a dedicated bare machine).

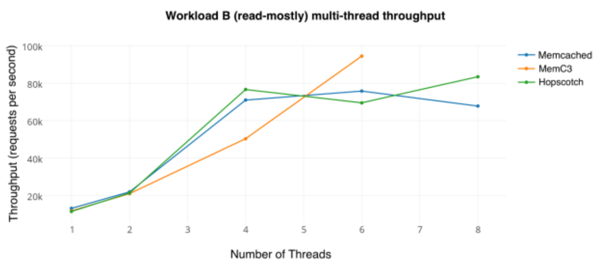

Figure 5: Workload B multithreaded throughput

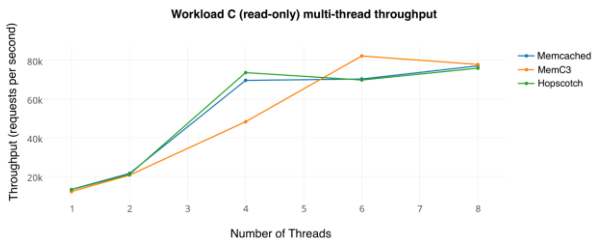

Figure 6: Workload C multithreaded throughput

It would be more informative to run on dedicated machines, so the experiment could record a series of data with low variance. This will give us a better platform to compare the throughputs of Memcached, MemC3 and this paper's version of Hopscotch.

Table 3. 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

9. CONCLUSION

Memcached was 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. The evaluation shows that the implementation achieves 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 this paper's implementation. The main motivation for the project was the significant improvement of MemC3 over original Memcached as claimed by the authors of Memc3. The 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

This 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, 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

11. 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.