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

Aris Paphitis
CUT

Key Point

The authors present a set of algorithmic and engineering improvements of the popular Memcached system, improving its memory efficiency and throughput.

1 Background

Memcached is a popular in-memory caching layer enabled by the demand of low-latency access to data, required by many Internet services. This requirement many system designers to serve data (all or most) from main memory, using the memory either as primary store or as a cache to deflect particular data structures. The most important metrics are performance (measured as throughput: queries pre second) and memory efficiency (measured by the overhead required to store an item). Memcached implements a simple and lightweight key-value interface where all key-value tuples are stored in and served from DRAM. Clients communicate with the Memcached servers over the network using a set of commands. Standard Memcached uses a typical hash table design, with link-list-based chaining to handle collisions. Chaining is efficient for inserting or deleting single keys, however lookup may require scanning the entire chain. Its cache replace algorithm is strict LRU, also based on linked lists. The least recently used entry is evicted and replaced by a newly inserted entry when the cache is full. The design relies on locking to ensure consistency among multiple threads, leading to poor scaling on multi-core CPUs. Memory allocation is slab-based. Memory is divided to 1MB pages, and each page is subdivided into fixed-length chunks. The size of a chunk, and the number of chunks per page, depends on the slab class. Key-value objects are stored in an appropriately-sized chunk. To insert a new key, Memcached looks up the slab class whose chunk size best fits this object. If no vacant chunk is available the search fails and cache eviction is executed. Memcached was originally single threaded. It uses libevent for asynchronous network I/O callbacks. Later versions support multi-threading via global locks, preventing Memcached from scaling on multi-core CPUs.

2 Observations

The authors, studying a recent publication by Facebook on several key-value store workloads, made the following observations. First, queries for small objects dominate. Most keys are smaller than 32 bytes (Memcached max limit is 250 bytes) and most values no more than a few hundred bytes. The consequence of storing such small keys is high memory overhead. Memcached always allocates a 56-byte header (on 64-bit servers) for each key-value object regardless of size. Thus a more efficient data structure for the index cache should be employed. Second, queries are read heavy. The studied workloads are characterized by 30x more reads than writes. Some applications have even more skewed workloads with 500x more writes. Though most queries are GETs, this operation is not optimized and locks are still used extensively. A more optimistic approach can benefit by removing all mutexes from the GET path.

3 Proposed Enhancements

The authors present a compact, concurrent and cache-aware hashing scheme, which they call optimistic concurrent cuckoo hashing. MemC3 manages to achieve high memory efficiency and increased throughput than original memcached by adapting to a more specific type of workload. This is accomplished by a adopting a more optimistic hashing scheme, a CLOCK-based eviction algorithm and an optimistic locking strategy.

3.1 Optimistic Concurrent Cuckoo Caching

The basic idea of cuckoo hashing is the use of two hashing functions instead of one, thus providing each key two possible locations where it can reside. MemC3's hash table is 4-way set-associative by providing an array of buckets, each having 4 slots. This design enables higher (almost double) space utilization. Additionally, to support keys of variable length, each slot stores a pointer to the key-value and a short hash of the key called a tag. This results in a more cache friendly lookup by avoiding long chains of pointer dereferences. The 1-Byte tag is compared in each lookup and the full key is retrieved upon a match. Tag collisions are below 1% and pointer dereferences are almost eliminated. Tags are also used to avoid retrieving full keys during insert operations, originally needed to derive the alternate location to displace keys. The hashing scheme used allows insert operations to operate using only information in the table without the need to ever retrieve keys. To enable concurrency deadlock risks and false misses have to be taken into consideration. Previous studies suggest breaking up inserts into atomic displacement with the effect of adding space overhead much higher than basic cuckoo hashing. MemC3 allows multiple concurrent reads and single writer using optimistic locks. To eliminate false misses MemC3 changes the order of insertion by first searching for a valid cuckoo path and then moving the keys backward along the cuckoo path. As a result, each swap affects only a key at a time, which can be successfully moved to its new location without any kick-out. Inserts are farther improved by employing multiple cuckoo paths.

3.2 Concurrent Cache Management

Basic memcached maintains a doubly linked list to implement its cache eviction strategy which is strictly LRU. This becomes a major source of space overhead, requiring 18 Bytes for each key. This is scheme is also a synchronization bottleneck since all updates to the cache must be serialized. MemC3 uses a CLOCK-based approach which approximates LRU with improved concurrency and space efficiency.

Notes