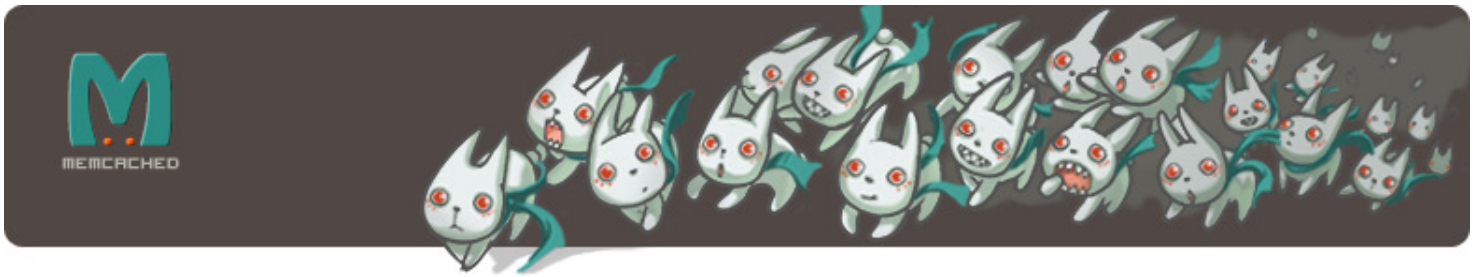


Replacing the cache replacement algorithm in memcached - Dormando (https://twitter.com/dormando) (October 15, 2018)

In this post we delve into a reworking of memcached's Least Recently Used (LRU) algorithm which was made default when 1.5.0 was released. Most of these features have been available via the "-o modern" switch for years. The 1.5.x series has enabled them all to work in concert to reduce RAM requirements.

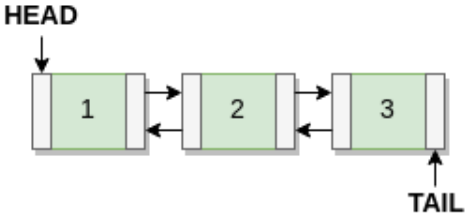
When memcached was first deployed, it was typically co-located on backend web servers, using spare RAM and CPU cycles. It was important that it stay light on CPU usage while being fast; otherwise it would affect the performance of the application it was attempting to improve.

Over time, the deployment style has changed. There are frequently fewer dedicated nodes with more RAM, but spare CPU. On top of this web requests can fetch dozens to hundreds of objects at once, with the request latency having a greater overall impact.

This post is focused on the efforts to reduce the number of expired items wasting cache space, general LRU improvements, as well as latency consistency.

Original Implementation

In 1.4.x and earlier, the LRU in memcached is a standard doubly linked list: There is a head and a tail. New items are inserted into the head, evictions are popped from the tail. If an item is accessed, it is unlinked from its position then re-linked into the head (hereby referred to as "bumping"), back at the top of the LRU.



There were no background threads to actively maintain data. If an item expired, it stayed in memory until it was accessed again. When looking up an expired item, the memory would be freed and a MISS returned to the client.

The only times expired items are removed:

- When intentionally deleted
- When overwritten via SET
- When expired then re-accessed via GET, ADD, and so on.
- When an eviction is required, it would look at a few items around the tail in search of an expired item to reclaim rather than evict the actual tail.

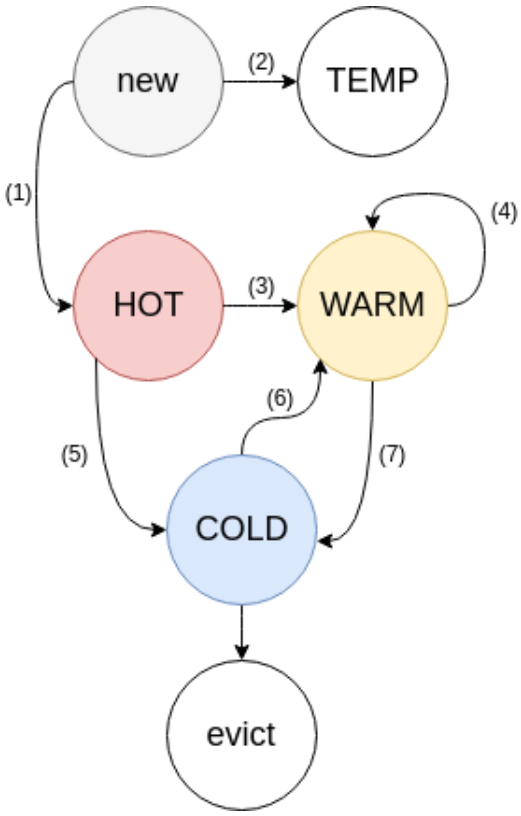
An important optimization was made early on which causes an item to only be bumped once every 60 seconds. Very frequently accessed items thus avoid excessive mutex locks and mutations.

Even if hot items don't always "bump" in the LRU, fetching hundreds of items in one go are likely to bump at least a few of them (or potentially all, if a user is returning from a snack break). This can cause spikes in mutex contention, uneven latency, and excess CPU usage on the server.

Multithreaded scalability is heavily limited by the LRU locking. With the original LRU, scaling beyond 8 worker threads was difficult. Very read heavy workloads have scaled nearly linearly to 48 cores once optimizations were made.

We took a minimalistic appraoch to finding a new algorithm. The code churn must be relatively small, backwards compatible, and easy to test and verify. End users also have a wide variety of unknown load patterns, requiring more generic approaches.

Segmented LRU



First improvement: we built a modified LRU influenced by the designs of 2Q, Segmented LRU, along with the naming from OpenBSD's filesystem buffer cache.

An LRU is split into four sub-LRU's. Each sub-LRU has its own mutex lock. They are all governed by a single background thread called the "LRU maintainer", detailed below.

Each item has two bit flags indicating activity level.

- *FETCHED*: Set if an item has ever been requested
- *ACTIVE*: set if an item has been accessed for a second time. Removed when an item is bumped or moved.

HOT acts as a probationary queue, since items are likely to exhibit strong temporal locality or very short TTLs (time-to-live). As a result, items are never bumped within HOT: once an item reaches the tail of the queue, it will be moved to WARM if the item is *active* (3), or COLD if it is inactive (5).

WARM acts as a buffer for scanning workloads, like web crawlers reading old posts. Items which never get hit twice cannot enter WARM. WARM items are given a greater chance of living out their TTL's, while also reducing lock contention. If a tail item has is *active*, we bump it back to the head (4). Otherwise, we move the inactive item to COLD (7).

COLD contains the least active items. Inactive items will flow from HOT (5) and WARM (7) to COLD. Items are evicted from the tail of COLD once memory is full. If an item becomes *active*, it will be queued to be asynchronously moved to WARM (6). In the case of a burst or large volume of hits to COLD, the bump queue can overflow, and items will remain inactive. In overload scenarios, bumps from COLD become probabilistic, rather than block worker threads.

TEMP acts as a queue for new items with very short TTL's (2) (usually seconds). Items in TEMP are never bumped and never flow to other LRU's, saving CPU and lock contention. It is not currently enabled by default.

HOT and WARM LRU's are limited in size primarily by percentage of memory used, while COLD and TEMP are unlimited. HOT and WARM have a secondary *tail age* limit, relative to the age of the *tail* of COLD. This prevents very idle items from persisting in the active queues needlessly.

LRU Maintainer Thread

This is all tied together by the LRU maintainer background thread. It has a simple job:

- Iterate over every sub-LRU and peek at the tail item.
- Ensure each sub-LRU is respecting its limits, moving items when necessary.
- Reclaim expired tail items.
- Process any asynchronous bumps from the COLD LRU.

This has a caveat of sustained overloading of SETs for a particular slab class could run out of COLD items to evict. If this happens, SETs enter "direct reclaim" mode and worker threads will block while pushing items out of HOT and WARM.

Summary

Besides memory efficiency, segmented LRU has several major effects on performance.

1. Items never bump when directly fetched. You can fetch 500 keys all at once and never have to wait for an LRU lock.
2. Items are bumped asynchronously. Short bursts in set or fetch traffic result in temporary imbalances in HOT or WARM.
3. The extra mutexes help scale write operations.
4. No increase in per-item metadata size.

The efficiency of these systems varies by workload. If you insert many items with a 60 second TTL, mixed in with items which have an 86400 (1D) TTL, the 60 second items will waste memory until they are either fetched or drop to the tail. In the meantime, the system evicts items with hours of life remaining.

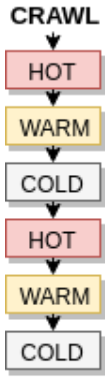
This is still a very effective cache. Since it is an LRU, frequently requested items with longer TTL's will likely stay toward the *head* and are free to live out their TTL's.

LRU Crawler

This implementation still has some outstanding issues:

Sizing the cache is hard. Do I have too much RAM? Too little? With all that waste in the middle, it's hard to tell. Items with inconsistent access patterns (ie; a user goes out to lunch or to sleep) may cause excessive

misses. Larger (multi-kilobyte) expired items could make room for hundreds of smaller items, or allow them to be stored for longer.

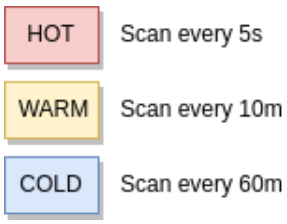


Solving these issues lead to the LRU crawler, which is a mechanism for asynchronously walking through items in the cache. It is able to reclaim expired items, and can examine the entire cache or subsets of it.

The crawler is a single background thread which inserts special crawler items into the *tail* of each sub-LRU in every slab class. It then concurrently walks each crawler item backwards through the LRU's, from bottom to top. The crawler examines each item it passes to see if it's expired, reclaming if so.

It will look at one item in class 1, HOT, then one item in class 1 WARM, and so on. If class 5 has ten items and class 1 has a million, it will complete its scan of class 5 quickly, then spend a long time finishing class 1.

A histogram of TTL remaining is built as it scans each sub LRU. It then uses the histogram to decide how aggressively to re-scan each LRU. For example, if class 1 has a million items with a 0 TTL it will scan class 1 at most once an hour. If class 5 has 100000 items and 1% of them will be expired in 5 minutes, it will schedule to re-run in five minutes. It can rescan every few seconds, if necessary.



Scheduling is powerful: higher slab classes naturally have fewer items that take a lot more space. It can very quickly scan and re-scan large items to keep a low ratio of dead memory. It can scan class 50 over and over, even if it takes 10 minutes to scan class 1 once.

Combined with segmented LRU, the LRU crawler may learn that "HOT" is never worth scanning, but WARM and COLD give fruitful results. Or the opposite: if HOT has many low TTL items, the crawler can keep it clean while avoiding scanning the relatively large COLD. This helps reduce the amount of scan work even within a single slab class.

If we tried to achieve this by walking the hash table from front to back. A 1MB item with 10 seconds of remaining TTL won't be seen again until every single item in the hash table is looked at once.

Summary

This secondary process covers most of the remaining inefficiencies of managing an LRU with TTL'ed data. A pure LRU has no concept of holes or expired items, and filesystem buffer pools often keep data around in similar sizes (say, 8k chunks).

Using a background process to pick at dead data, while self-focusing where it can be the most effective, reclaims almost all of the dead memory. It is now much easier to gauge how much memory a cache is actually taking.

Other features can and have been built on top of the LRU crawler:

```
lru_crawler metadump <classid,classid|all>
example output:
key=foo exp=1539196410 la=1539196351 cas=3 fetch=no cls=1 size=64
```

This command, taking either a list of class ids or "all", will asynchronously print key and metadata information on every item in the cache.

Tuning Memcached

The LRU code has several options which can be changed on a running instance.

```
lru [tune|mode|temp_ttl] [option list]
```

- “mode” flat|segmented: you can switch from segmented LRU to the traditional LRU mode while running, and back again. Items in HOT or WARM will drain to COLD asynchronously, and COLD will become the main LRU.
- “tune”: Takes “percent hot”, “percent warm”, “max hot age factor”, “max warm age factor” as arguments. This allows you to dynamically experiment with hit rates with larger or smaller HOT or WARM queues.

```
lru tune 10 25 0.1 2.0
```

In this example HOT is limited to 10% of memory, or a tail age 10% of COLD’s tail age. WARM is limited to 25% of memory, or 200% of COLD’s tail age. The LRU maintainer will move items around to match new limits if necessary.

- “temp_ttl” ttl: Set to -1 to disable, or higher than 0 to enable usage of the TEMP LRU at runtime. Any objects entered with a TTL less than specified will go directly into TEMP and stay there until expired or otherwise deleted.

These are all tunable via commandline startup options as well, use ‘-h’ command for up to date start options.

doc/protocol.txt from the source has the most up to date detail on live tuning. It also contains explanations for new stats counters related to this code, which mostly appear under “stats items” output.

Conclusion

This post introduced two improvements which affect memory efficiency and were enabled by default in memcached 1.5.0.

There are many new and interesting ideas in the software community, which we will continue to experiment with. Creating a high speed key/value cache for generic work profiles requires carefully evaluating tradeoffs. The implementation work for these features took a couple weeks combined, but production testing lead to small changes over years.

We’ve shown it’s possible to greatly improve the memory efficiency of memcached while simultaneously improving the latency profile. This is achieved by using a limited amount of CPU in background threads.