Advanced Caching Policies for Memcached

Saurabh Mathur Sethu Prakasam

ABSTRACT

Memcached is an open source, distributed caching system whose simple designs solves many problems that arise with large data caches. It uses the Least Recently Used (LRU) policy for eviction. This policy suffers from some shortcomings and the use of advanced policies can make memcached more efficient. We implement Greedy Dual Size(GDS) and Least Inter-Reference Recency Set(LIRS) algorithms and compare them against the LRU policy.

1 Introduction

Memcached is a high performance multithreaded event-based key/value cache store intended to be used in a distributed system. Its simple designs solves many problems that arise with large data caches. The current status of Memcached is that there are currently 127 contributors and the system is on its 79th release. Memcached uses the slab allocation technique and supports the use of two types of hash functions to optimize its operations.

The main selling point of Memcached is that it can utilize many server nodes without without needing the servers to be aware of each other. When there is a request for a particular key and it is not found on the current server, other servers are polled. Thus, memcached logically combines many small caches into one large cache.

When all the storage has been used, entries have to be replaced. The cache eviction policy currently used is the Least Recently Used (LRU) policy.

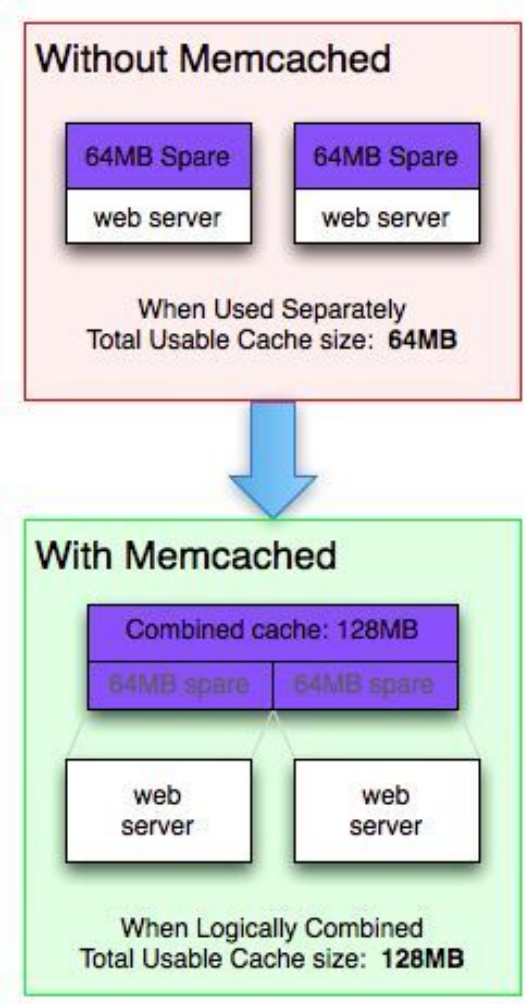

Figure 1: **Memcached logically combines caches from different server nodes.** Reprinted from https://memcached.org/about

.

1.1 Least Recently Used

The LRU policy is specified as follows:

If p is in K

- Move p to top of K

If p is not in K

- Remove an entry from the bottom of K.
- Add p to the top of K
- Bring p into the cache.

While LRU is very simple to implement and works reasonably well, there are some cases when LRU performs poorly. Specifically, they are

1. Values are of very different sizes
2. Repeated sequential scans are performed.
3. Accesses are cyclic. For example, in a for-loop
4. Some block are more likely to be accessed again. For Example. Index blocks

Our contribution is to implement advanced cache eviction policies to overcome the shortcomings of LRU. The eviction policies will be delivered as a add-on onto the existing, fully developed Memcache product. The policies will be incorporated as separate files that can be toggle enabled.

Specifically, we have implemented the following algorithms:

- Greedy Dual Size, which tackles problem 1.
- Low Inter-Reference Recency Set, which tackles problems 2,3,4.

2 Background

2.1 Low Inter-Reference Recency Set

This algorithm was developed by Jiang et. al. [3] in 2002. It is specified as follows:

List Q holds HIR resident blocks and stack S holds LIR, resident and non-resident HIR

Document p is accessed.

p is LIR :

- Move to top of S

p is HIR & resident :

- If p is not in S, move it to end of Q
- Else, set it as LIR, remove it from Q. Move an item from bottom of S to end of Q.

p is HIR & non-resident :

- Evict a block from front of Q.
- Add the new block p to top of S.
- If p is not in S, move it to end of Q.
- Else, set it as LIR, remove it from Q. Move an item from bottom of S to end of Q.

This algorithm categorizes each item into two sets - hot (LIRS) and cold (HIRS). If a hot page is evicted, it becomes cold and if a cold page is evicted, it is removed from the cache. Another key idea is to use the reuse distance to quantify the recency of access.

2.2 Greedy Dual Size

This algorithm was developed by Cao et. al. [2] in 1997. It is specified as :

Initialize L = 0.

Document p is accessed.

If p is already in the cache

- set H(p) to L + c(p)/s(p).

If p is not in the cache

- While there is not enough room in the cache for p,
- Let L be the minimum H(q) over all pages q in the cache.
- Evict q such that H(q) = L.
- Bring p into the cache and set H(p) to be L+c(p)/s(p)

This algorithm maintains two variables - L the inflation factor and H which is the eviction cost for each entry in the cache.

3 Implementation

Memcached uses LRU. We wanted to replace LRU with the advanced policies.. One way is directly replacing LRU in codebase. However, LRU is very tightly coupled with the

Memcached codebase. So, we have implemented a wrapper around the existing Memcached client library. We are preemptively removing entries using caching policies to achieve better hit rate.

2.1 Architecture

Figure 2: **Implementation architecture**

1. The client invokes the wrapper.
2. The wrapper calls the advanced caching policy.
3. The caching policy interacts with the backing store (ie memcached).

The caching policies save only the keys and the metadata (like the eviction cost H in Greedy Dual Size Algorithm). Eviction is done via the delete function of memcached.

Our implementation allows the backing store to be either a python dict or the memcached client. All functionality of memcached works as expected with our wrapper. The only change is in the get and put methods.

2.2 Data

To evaluate the caching policies we used two type of trace data:

1. Data generated from the zipf distribution.
2. Apache web-server access.log data

Since, web server caches follow the zipf distribution, the generated data would be as realistic as possible. For a more practical evaluation, the access.log data was used. (from https://pwning.re/files/enei-ctf/access.log)

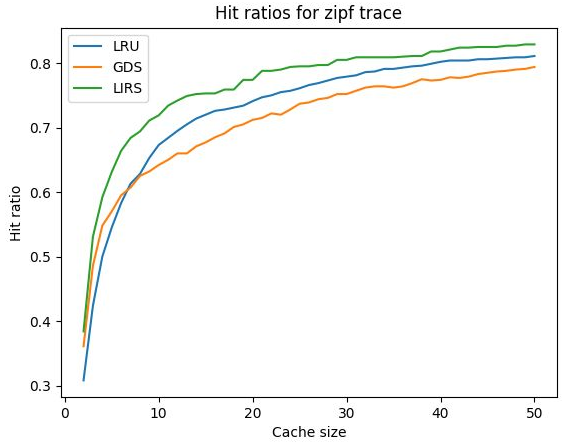

Figure 3: **Hit ratio curve for zipf trace**

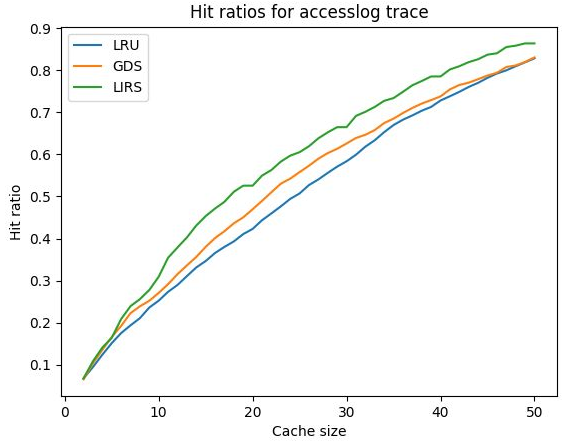

Figure 4: **Hit ratio curve for access.log trace**

4 Evaluation

We evaluated the

Algorithm (zipf)	Average Hit rate

LRU	0.729143
GD	0.709163
LIRS	0.764939

Algorithm (accesslog)	Average Hit rate
LRU	0.504711
GD	0.529682
LIRS	0.576501

5 **Results**

Our evaluation shows that Greedy Dual Size and LIRS algorithms perform better than the LRU algorithm and with our wrappers for the memcached client, better performance can be achieved.

REFERENCES

[1] Fitzpatrick, Brad. "Distributed caching with memcached." Linux journal 2004, no. 124 (2004): 5.

[2] Cao, Pei, and Sandy Irani. "Cost-aware www proxy caching algorithms." In Usenix symposium on internet technologies and systems, vol. 12, no. 97, pp. 193-206. 1997.

[3] Jiang, Song, and Xiaodong Zhang. "LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance." ACM SIGMETRICS Performance Evaluation Review 30, no. 1 (2002): 31-42.