Storage and Performance Optimization of Long Tail Key Access in a Social Network

John Liang James Luo Mark Drayton Rajesh Nishtala
Richard Liu Nick Hammer Jason Taylor Bill Jia
Facebook
Menlo Park, CA
{johnl, jamesluo,mbd,rajesh.nishtala,richxliu,hammer,jasont,billjia@fb.com}

ABSTRACT

In a social network, it is natural to have hot objects such as a celebrity's Facebook page. Duplicating hot object data in each cluster provides quick cache access and avoids stressing a single server's network or CPU resources. But duplicating cold data in each cache cluster consumes significant RAM. A more storage efficient way is to separate hot data from cold data and duplicate only hot data in each cache cluster within a data center. The cold data, or the long tail data, which is accessed much less frequently, has only one copy at a regional cache cluster.

In this paper, a new sampling technique to capture all accesses to the same sampled keys is created. We then calculate the working set size for each key family for estimating the memory footprint. We introduce an important metric, *duplication factor*, as the ratio between the sum of each individual cluster's working set size and the regional working set size. We analyze why some key families have a higher duplication factor.

It is important to separate hot keys and cold keys from the same key family with minimal overhead. We present a novel cache promotion algorithm based on key access probability. We also proposed a probability model based on the binomial distribution to predict the promotion probability with various promotion thresholds.

Our experiment shows by shrinking the cluster level cache layer and having a fat regional level cache for cold data, we are able to achieve a higher combined cache hit ratio.

Categories and Subject Descriptors

C.2.4 [Distributed Systems]: Distributed Databases; B.3.2 [Design Styles]: Cache memories; D.4.2 [Storage Management]: Distributed Memories

Keywords

Cache Optimization, Workload Analysis, Key-Value Store

1. INTRODUCTION

Memcached is a well-known open source in-memory key-value caching solution. Facebook has the largest Memcached installation in the world, which scales to process billions of users requests per second. To serve 1 billion+ active users, Facebook has improved the open source version of Memcached with an adaptive slab allocator, the transient item cache, and leases[16].

To support the scale of a billion users, we organize our machines into various types of clusters[16]. *Frontend clusters* hold web servers that serve user requests and query databases. *Storage clusters* hold database instances. Memcached servers within frontend clusters cache the results of database requests issued by web servers. This is defined as *cluster layer caching* or *L1 caching*. Many frontend clusters form a region. In any region of clusters, there is exactly one storage cluster whereas there are multiple frontend clusters.

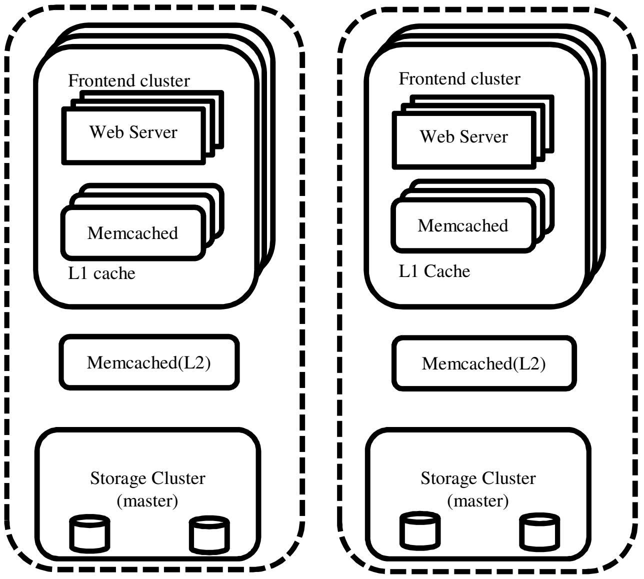

Figure 1: Data center infrastructure.

Memcached servers at the regional level are shared by web servers from all clusters in the same region. In this paper, we define these regional Memcached caches as *L2 caches*. If a key-value pair resides in an L2 cache, one copy can serve all web requests from N frontend clusters in the same region. For example, Derek in New York uploaded a photo into Facebook and shared it with his friends. Travis, one of Derek's friends, immediately saw this photo in his News Feed. At that time, the metadata of this photo, such as creation time and tags, is loaded into L1 caches for the hope this photo will be requested again in a short period of time. Later on, Justin, another friend of Derek's, accessed this photo. This time, Justin's request was served by another frontend cluster routed by a load balancer. Now the same content has 2 copies at L1 caches in the same region. Assuming we have N frontend clusters in a region, we would soon have N copies of the same cache item for this photo. If this photo object stays at L2 cache, only one copy is required to serve both Travis's and Justin's request. This works as long as this photo is not hot. If this photo happens to be hot, too many concurrent requests to the L2 cache will exceed the capacity of a single host in the L2 cache tier and thus deteriorate the user experience. For most users, the average number of photos viewed by their friends is bound by the number of friends. But for a small number of celebrities, their photos are usually hot as they could have millions of fans. This long tail access pattern is the de facto characterization for a social network. Only one copy will be stored if we separate the long tail cold data into L2 and promote only hot data to L1.

In the rest of this paper, we will discuss the proposed long tail access pattern which enables the efficiency savings. We will also present the mathematics model for the probability promotion algorithm. Experiment results are demonstrated in the end.

2. ACCESS PATTERN ANALYSIS

2.1 Concepts

Each key-value pair has a key ID, which is a string. A normalized version of key-id is defined as a normalized key. For example, key ID photos:foobar:{12345}:c{1} has photos:-foobar:{N}:c{N} as its normalized key where N stands for an integer value. For this particular example, '12345' is a user ID and the second N = 1 is for version number. A key family is a set for all cache key-value pairs which have the same normalized key.

The keys are organized as key families and this provides flexibility to 1) assign dedicated Memcached pools, 2) add TTL (time to live) parameters, and 3) migrate processes as we can manage the whole key family as a whole. The prefix of normalized key is used to detect which pool the key belongs to. For example, all keys starting with prefix 'photo' go to the photo tier which has its dedicated Memcached machines.

To increase memory efficiency, we look to the *working set size* for each key family. We define working set size as the aggregate key and value sizes for all *unique* keys for a given time interval. Conceptually, if the same key is accessed multiple times to the same cache host, it is counted only once in calculating the working set because only one copy of the key-value pair is needed to serve the subsequent requests.

$$wss = \sum_t \sum_k (len(key) + len(value))$$

Where k is the number of *unique* keys and t is the time duration for calculating the working set. For example, we can calculate weekly, daily and hourly working sets.

2.2 Key-Based Sampling

For hotness analysis, we need to figure out how many times a key is accessed. This implies the same key always needs to be sampled. We choose a key-based sampling method by using a cheap hash function, as follows:

$$hash(key\_id) \,\%\, sample\_rate == 0$$

Where sample_rate = 1,000,000 to reduce sampling overhead.

This sampling technique has many advantages compared with user- or request-based sampling. Because keys are not sampled by user requests, key-based sampling can capture all cache accesses including ones not associated with user activities. As we capture all accesses to the same key once it is sampled, it can also be used for capacity miss, consistency miss analysis where other sampling techniques do not fit.

In this experiment, each web server in a frontend cluster continuously logs sampled cache key accesses to local disk. The log data is then pushed to a data warehouse by scribeH[1]. We use Hive[2] to analyze it as Hive provides SQL-like queries to allow aggregated size calculations grouped by key family.

2.3 Long Tail Access Pattern

Facebook has the largest social network in the world. It is natural to have hot objects such as a celebrity's Page, photos and News Feed stories. Figure 2 demonstrates the long tail access pattern CDF (Cumulative Distribution Function) for photos.

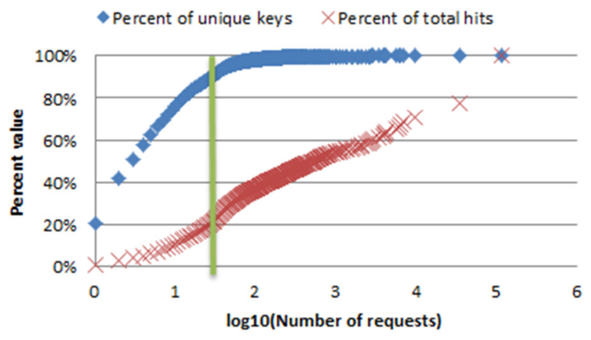

Figure 2: Hotness CDF distribution.

The intersection of the vertical line and top curve illustrates that 90% of unique photo keys are accessed for no more than 32 (log32 = 1.5) times per day. This also means that only 10% of unique keys are accessed more than 32 times. The intersection of the vertical line and red curve demonstrates that 90% of cold keys contribute only 20% for the total photo cache traffic and that 10% of hot keys contribute 80% of the total requests. Each Facebook user has about 100 friends on average. If 1/3 of those friends (100/3 = 33) are active users, this explains how 90% of photos are accessed for up to 32 times per day. On the other hand, there are users who have significantly more friends or fans such as celebrities. A photo from a celebrity with 7 million fans is accessed much more times than the ones from an average user.

A long tail access pattern applies not only to key families that are shared across users, but also to the ones that are associated with a single user. For example, key families associated with News Feed are among the most frequently accessed keys, as users spend considerable time consuming News Feed. We have found the same long tail access pattern as some users are more active than others.

2.4 Duplication Factor

The long tail access pattern in section 2.3 demonstrates that 90% of the working set is contributed by cold keys, which are accessed for no more than 32 times a day. If these cold keys reside in the cluster level (L1) cache, we will have N copies assuming we have N frontend clusters in the same region. It makes perfect sense to move these cold keys to the regional level (L2) cache especially because the network traffic caused by cold keys is only 20%. This new architecture makes the L1 cache thin and the L2 cache fat in terms of consumed memory. The L1 cache layer serves as cache replicas only for hot keys.

To further simplify detecting key families suitable for L1/L2 architect, we introduced a new metric, *duplication factor*. It is defined as the ratio between the sums of each cluster's working set size and the regional working set size.

$$Duplication\ Factor = \frac{\sum_{i=0}^{k-1} cluster\_ws(i)}{region\_ws}$$

When we have k frontend clusters in the same region, the duplication factor value ranges between 1 and k. Its maximum value k is obtained if all key-value pairs for the same key family have been duplicated in all k frontend clusters. Its minimum value is obtained when each cluster has zero overlap of the same key family. The regional_ws can be considered as the memory footprint required if the whole key family is served in L2 cache.

Duplication factor is correlated to the number of average unique users, which is the total count of different users who have accessed a key family. When a key family is shared across more unique users, it has a higher probability of being accessed from more frontend clusters.

3. PROMOTION ALGORITHM

3.1 L1/L2 Architecture

We use photo L1 and L2 caches to demonstrate the proposed algorithm workflow. Photo L1 cache is the front cache for photo-related keys; L2 cache is the second level cache. The cache client resides in the WWW tier or PHP code. The workflow acts as follows:

1. Cache client requests a key k from L1 cache.
2. L1 cache does not have this key, it returns a cache miss to the web server.
3. The web server sends a get request to the L2 cache upon receiving the L1 cache miss.
4. Assuming the L2 cache has its key, it returns the value of this key to the web server.
5. The web server generates the random number and does the following check:if(mt_rand(1,N)%N==1), if true, it sends a set request to the L1 cache to promote this key to L1.
6. Assuming at step 4 the L2 cache does not have its key, it returns miss.
7. The web server then sends a request to the backend database layer and fetches the value for key k.
8. The web server sends a set request to the L2 cache upon receiving the value from the database.

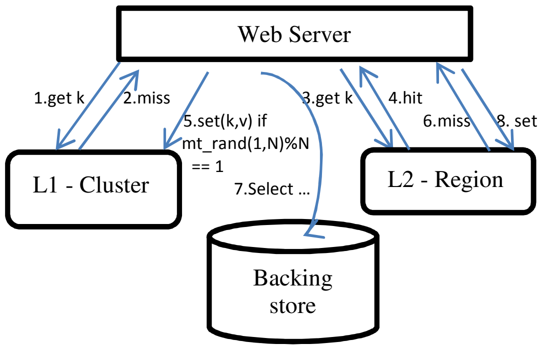

Figure 3: Promote algorithm workflow.

Step 5 is the main innovative idea of separating the hot key from the cold key. Intuitively, hot keys have a higher probability to be promoted simply because they have more trials to pass the check. N is a crucial number which controls the cache traffic between L1 and L2. When N has a larger value, a key in L2 has less chance to be promoted. And, thus, less cache traffic goes to the L1 cache. As in Figure 3, in case of a cache miss in L2, the web server sets the value to only L2. This step prevents keys from polluting L1 if they are accessed only once. In section 3.2, we will discuss more details on how to tune this number.

3.2 Probability Modeling

Assume N is the promotion threshold value. $P(S=k)$ is the probability a key is promoted when it gets accessed at kth time. $P(F=k)$ is the probability a key is not promoted when it gets accessed at kth time.

As the total number of keys is extremely large, we can assume the probability is independent of each trial.

So for each individual trial:

$$P_s = \frac{1}{N}\ where\ P_s\ is\ the\ success\ probability$$

$$P_f = \frac{N-1}{N}\ where\ P_f\ is\ the\ failure\ probality$$

$$P(S=k) = P(F=k-1) * P_s$$

$$= \frac{1}{N} * P_f * P(F=k-2)$$

$$= \frac{1}{N} * (P_f)^{k-1} = \frac{1}{N} * \left(\frac{N-1}{N}\right)^{k-1}$$

For cumulative probability:

$$P(S \le k) = P(S=1) + P(S=2) + \cdots + P(S=k)$$

$$= \frac{1}{N} + \frac{1}{N} * \frac{N-1}{N} + \frac{1}{N} * \left(\frac{N-1}{N}\right)^2 + \cdots + \frac{1}{N} * \left(\frac{N-1}{N}\right)^{k-1}$$

$$= \frac{\frac{1}{N} * \left(1 - \left(\frac{N-1}{N}\right)^{k-1}\right)}{1 - \left(\frac{N-1}{N}\right)}$$

$$= 1 - \left(\frac{N-1}{N}\right)^{k-1}$$

For example, if N=32, to calculate the probability a key can be promoted with 10, 32 and 100 trials:

$$P(S \leq 10) = \quad 1 - \left(\frac{32-1}{32}\right)^{10-1} = 27.2024\%$$

$$P(S \leq 32) = \quad 1 - \left(\frac{32-1}{32}\right)^{32-1} = 63.7945\%$$

$$P(S \leq 100) = \quad 1 - \left(\frac{32-1}{32}\right)^{100-1} = 95.482\%$$

It means a key has 27.2024% probability to be promoted with 10 trials; 63.794% probability to be promoted with 32 gets; and 95.482% probability to be promoted for no more than 100 gets.

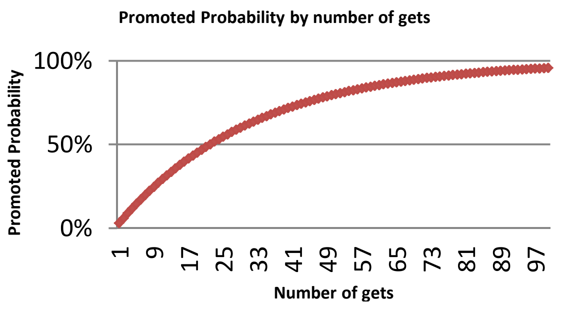

Figure 4: Promotion probability with threshold = 32

Figure 4 demonstrates the relationship between the number of gets and its promoted probability. When the number of trials is 73, a key has a 90% chance of being promoted. Based on this figure, it shows that if a key is hot, it is almost guaranteed that it will be promoted to L1.

3.3 Tuning Promotion Threshold

Promotion threshold is the N value we choose for our promotion algorithm. This value plays an important role for multiple layer cache performance. On one hand, a hot key needs to be able to be promoted to L1 after a reasonable number of gets; on the other hand, a cold key promotion should be avoided as much as possible so the L1 cache would not be polluted with falsely promoted cold keys from L2.

The following chart shows the probability curve for various threshold values of 8, 16, 32, 64 and 128. We choose threshold value T=32 out of the 5 curves presented from Figure 5 because when the number of trials is 100, the promoted probability is high at 95.82%. And when the number of trials is 10, the promoted probability is low at 27.20%. This ensures hot keys will be prompted and cold keys will not.

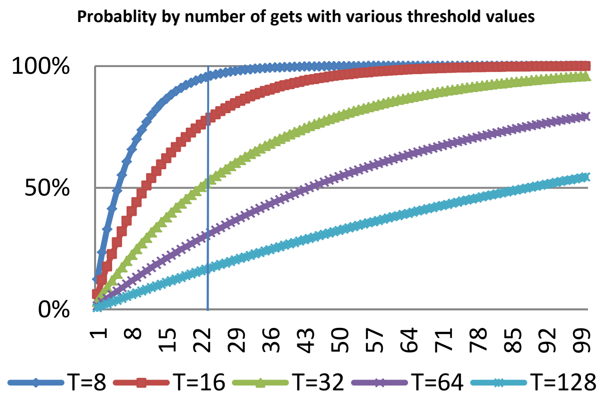

Figure 5: Promotion probability with various threshold values = 8, 16, 32, 64, 128.

4. EXPERIMENT RESULTS

We selected the photo tier for the L1/L2 experiment. A photo tier includes dedicated physical hosts forming a pool of Memcached cache hosts. This tier serves cache queries for only photo-related objects at Facebook. Before this experiment, photo had its own dedicated tier in each frontend cluster. This photo tier is an L1 cache tier and no photo cache traffic went to the L2 cache layer before this experiment.

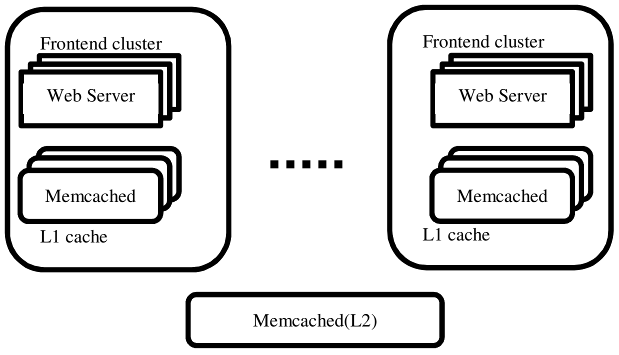

Figure 6: Photo L1/L2 experiment architecture

For this experiment shown in Figure 6, we added a new L2 photo tier and directed all L1 cache misses to this new L2 tier. A L1 miss will be served from L2 first before it goes to database layer. There are multiple frontend clusters in this setup. They are all backed by the same L2 tier. When this tier was added originally, the overall cache hit ratio jumped as we essentially added more hardware to serve the same traffic. We then started shrinking each individual L1 cache layer to the point where it has just sufficient memory to hold hot objects but not cold objects.

4.1 Cache Hit Ratio

Cache hit ratio is an important metric to measure cache performance and efficiency. As in Figure 7, before the L2 cache is enabled, two frontend clusters have hit ratios of 86% at peak; the third frontend cluster has a hit ratio of 89% at peak.

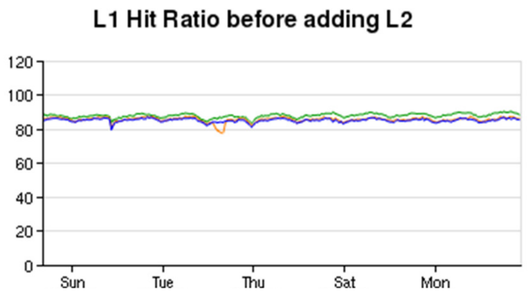

Figure 7: L1 Hit Ratio before adding L2

When L2 is enabled with promotion threshold = 1 on Wednesday in Figure 8, the third cluster has a hit ratio of 88% and L2 has a hit ratio of 51.7%. The aggregated hit ratio is calculated as follows:

88%+(1-88%)*51.7% = 94.2%

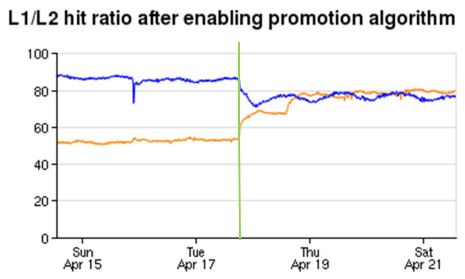

Figure 8: Hit ratio after promotion algorithm enabled

In terms of cache miss, it was 1-89%=11% compared with 1-94.2%=5.8%. This is a 56.36% miss reduction by adding the L2 cache.

Because the promotion threshold = 1, it means a newly fetched key from the database to L2 will be always set to the L1 cache. And, thus, the hot keys are not separated from cold keys. We set the promotion threshold to 32. In Figure 9 as shown in the rectangle, the L1 cache hit ratio starts to decrease while the L2 cache hit ratio jumps. This can be explained because cold keys are removed from the L1 cache and they are served only from the L2 cache. So any cache gets to these cold keys becomes a cache miss in L1 compared with a possible cache hit before the threshold = 32 is set. Consequently there are more cache requests to L2 cache as a direct result of L1 cache misses for cold keys, these requests to L2 increase the L2 cache hit ratio.

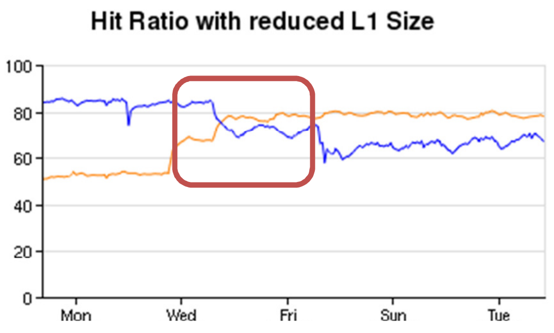

Figure 9: Hit Ratio with reduced L1 size

In Figure 10, the overall hit ratio is as follows:

66%+(1-66%)*80%= 93.2%

This is 4.2% hit ratio increase with a less number of machines in L1.

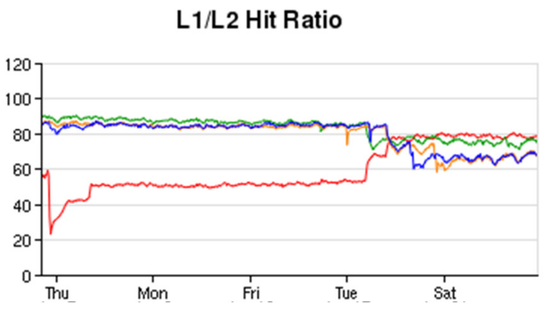

Figure 10: Hit Ratio change for all FE clusters

4.2 Eviction Age with Promotion Algorithm Enabled

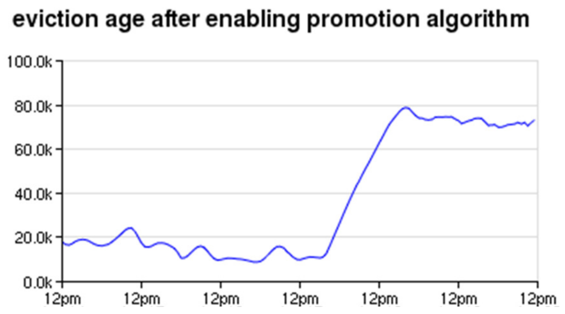

Figure 11: Eviction age increase by promotion algorithm

Eviction age is the age for the last item evicted from the Memcached LRU queue. A longer eviction age means more cache items can be stored. The promotion algorithm plays a crucial role in reducing the number of L1 hosts. In Figure 11, right after the promotion algorithm was enabled for the frontend cluster, the eviction age started to climb up.

When the promotion algorithm is enabled, cold keys have a very low probability of being promoted from L2 to L1. Only hot keys will be promoted and set to L1. This simple algorithm significantly reduced the eviction pressure at L1 as there are fewer items to be set. Figure 11 demonstrates that our promotion algorithm successfully separated hot keys and cold keys.

4.3 Network Traffic Distribution

Our promotion algorithm controls where a key will be stored depending on the hotness of a key and the promotion threshold value. Consequently, it also controls where a cache key is served. This effect has great implications on network traffic distribution for both the cluster and regional levels. A higher promotion threshold value results in fewer keys being promoted to the L1 cache; thus, more cache traffic is served from L2.

In Figure 12, network traffic in the L2 cluster in the red line sharply increased while the three L1 caches had a relatively small decrease in network traffic. Depending on the network bandwidth availability, we can shift traffic between L1 and L2 so that network management becomes more flexible.

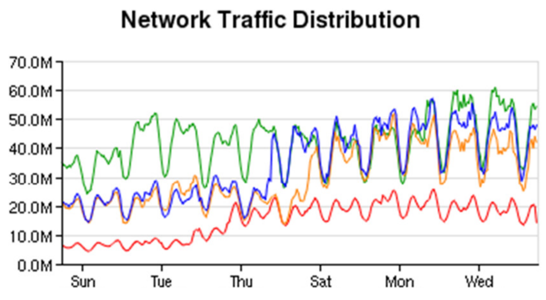

Figure 12: Network traffic distribution

5. RELATED WORK

Workload analysis is crucial for storage and caching system optimization. Accordingly, there is more research work[3,4,5,7,-8,9,10,11] on the sampling, analysis and characterization of the workloads on storage and caching system. Atikoglu et. al. implemented a network sniffer to capture traces from the Memcached server. Cache hit rates over time and access locality were studied in great detail[4]. Ahmad I. proposed a disk I/O workload characterization system using online histograms in a virtual machine environment[7].

The long tail access pattern has been studied in many recent research papers[4,6]. For example, Atikoglu et. al. found that 50% of keys occur in only 1% of all requests. These keys do not repeat many times but a few popular keys or hot keys repeat in millions of requests per day[4].

Cache allocation and placement algorithms have been discussed in many research papers[14,15,16]. Yadgar G. proposed a local management scheme, Karma, which uses readily available information about the client's future access profile to save the most valuable blocks and to choose the best replacement policy[14]. Performance and efficiency improvements based on workload analysis are discussed in [6,12,13]. Traverso et. al. implemented a system to selectively distribute long-tail content across PoPs. This system exploits social network relationships and achieved as much as 80% cost savings for WAN bandwidth[6].

6. CONCLUSION AND FUTURE WORK

This paper presented a novel idea to optimize cache keys with a long tail access pattern by separating hot items to the L1 cache and cold items to the L2 cache. This approach reduces the data duplication in frontend cluster and thus fewer cache servers are required to maintain the same cache hit ratio. To our knowledge, the promotion algorithm based on access frequency probability is a novel idea, which nicely filters out long tail cold items with minimal overhead. Our Memcached prototype with real Facebook production traffic demonstrated better hit ratio and user experience.

ACKNOWLEDGMENTS

We are grateful for the valuable feedbacks provided by Goranka Bjedov, Yee Jiun Song, Yu Chen, Min Ni, Harry Li, Venkat Venkataramani, Frank Frankovsky, and Jay Parikh. We would like to thank Pete Bratach for his help reviewing and editing this paper.

7. REFERENCES

[1] http://wiki.github.com/facebook/scribe

[2] http://wiki.apache.org/hadoop/Hive

[3] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P. Finding a needle in haystack: Facebook's photo storage. *In Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation* (Oct. 2010).

[4] Atikoglu, B., Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Workload analysis of a large-scale key-value store. *In SIGMETRICS '12: Proceedings of the 12th ACM SIGMET-RICS/PERFORMANCE joint international confe-rence on Measurement and Modeling of Computer Systems*, pages 53–64, New York, NY, USA, 2012. ACM.

[5] Feitelson, D. G. Workload modeling for Perform-ance evaluation. In Performance Evaluation of Complex Systems: Techniques and Tools, M. C. Calzarossa and S. Tucci, Eds., vol. 2459 of Lecture Notes in Computer Science. Springer-Verlag, Sept. 2002, pp. 114{141. www.cs.huji.ac.il/~feit/papers/WorkloadModel02chap.ps.gz.

[6] Traverso, S., Huguenin, K., Trestian, I., Erramilli, V.,Laiutaris, N., Papagiannaki, K.: TailGate: Handling Long-Tail Content with a Little Help from Friends. WWW 2012, April 16-20, 2012, Lyon, France.

[7] Ahmad, I. Easy and e_cient disk I/O workload characterization in VMware ESX server. In Proceedings of IEEE International Symposium on Workload Characterization (Sept. 2007).

[8] Kavalanekar, S., Worthington, B., Zhang, Q., and Sharda, V. Characterization of storage workload traces from production windows servers. In Proceedings of IEEE International Symposium on Workload Characterization (Sept. 2008).

[9] Keeton, K., Alistair Veitch, D. O., and Wilkes, J. I/O characterization of commercial workloads. In Proceedings of the 3rd Workshop on Computer Architecture Evaluation using Commercial Workloads (Jan. 2000).

[10] Breslau, L., CAO, P., FAN, L., PHILIPS, G., and SHENKER, S. Web caching and zipf-like distributions: evidence and implications. In Proceedings of the 18th Annual IEEE International Conference on Computer Communications (1999).

[11] Lublin, U., and Feitelson, D.G. The workload on parallel supercomputers: Modeling the characteristics of rigid jobs. Journal of Parallel and Distributed Computing 63, 11 (Nov. 2003), 1105-1122.

[12] Sastry, N., Yoneki, E., Crowcroft. Buzztraq, J. Predicting Geographical Access Patterns of Social cascades Using Social Netowkrs. In SNS, 2009.

[13] Scellato, S., Mascolo, C., Musolesi, M. and Crowcroft. Track Globally, Deliver Locally: Improving Content Delivery Networks by Tracking Geographic Social Cascades. In WWW, 2011.

[14] Jiang, S., Zhang, X. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In Proceedings of the 2002 ACM SIGMETRICS international conference on Measurement and modeling of computer systems (2002), SIGMETRICS'02, ACM, pp. 31-42.

[15] Megiddo, N., Modha, D. S. ARC: A self-tuning, low overhead replacement cache. In Proceedings of the 2nd USENIX Conference on File and Storage Technologies, 2003, pp. 115-130.

[16] Nishtala R., Fugal H., Grimm S., Kwiatkowski M., Lee H., Li C.,Mcelroy R., Paleczny M., Peek D., Saab P., Stafford D., Tung T., and Venkataramani V.: Scaling Memcache at Facebook. NSDI 2013 (to appear).