Dynacache: Dynamic Cloud Caching

Asaf Cidon[1], Assaf Eisenman[1], Mohammad Alizadeh[2], and Sachin Katti[1]

[1]Stanford University

[2]MIT CSAIL

1. ABSTRACT

Web-scale applications are heavily reliant on memory cache systems such as Memcached to improve throughput and reduce user latency. Small performance improvements in these systems can result in large end-to-end gains, for example a marginal increase in hit rate of 1% can reduce the application layer latency by over 25%. Yet, surprisingly many of these systems use generic first-come-first-serve designs with simple fixed size allocations that are oblivious to the application's requirements. In this paper, we use detailed empirical measurements from a widely used caching service, Memcachier [1] to show that these simple default policies can lead to significant performance penalties, in some cases increasing the number of cache misses by as much as 3×.

Motivated by these empirical analyses, we propose Dynacache, a cache controller that significantly improves the hit rate of web applications, by profiling applications in real-time and dynamically tailoring memory resources and eviction policies. We show that for certain applications in our real-world traces from Memcachier, Dynacache reduces the number of misses by more than 65% with a minimal overhead on the average request performance. We also show that Memcachier would need to more than double the number of Memcached servers in order to achieve the same reduction of misses that is achieved by Dynacache. In addition, Dynacache allows Memcached operators to better plan their resource allocation and manage server costs, by estimating the cost of cache hits as a function of memory.

2. INTRODUCTION

Memory caches like Memcached [8] and Redis [2] have become a vital component of cloud infrastructure. Many major web service providers such as Facebook, Twitter, Pinterest, Box, and Airbnb have large deployments of Memcached, while smaller web applications use caching-as-a-service solutions like Amazon ElastiCache [3] and Memcachier [1]. These applications rely heavily on caching to improve the latency of web requests, reduce load on backend databases, and lower operating costs.

Even modest improvements to the cache hit rate have a significant impact on user perceived performance in modern web services, because reading data from a disk-based database (like MySQL) is orders of magnitude slower than in-memory cache. For instance, the hit rate of one of Facebook's main Memcached pools has been reported to be 98.2% [5]. Assuming the latency for a typical cache hit and MySQL read is $200\mu s$ and 10ms, respectively, increasing the hit rate by just 1% would reduce the average read latency by over 25%. The end-to-end benefits can be substantially larger for user queries, which often must wait on hundreds of reads [13].

Memory caching systems are very simple and are roughly modeled on CPU caches with tables of nearly equal-sized items and Least Recently Used (LRU) evictions. Importantly, current memory caches are *oblivious* to application request patterns and requirements. Memory allocation across slab classes[1] and across different applications sharing a cache server is based on fixed policies like first-come first-serve or static reservations. The eviction policy, LRU, is also fixed.

In this paper, we argue that this one-size-fits-all cache behavior is poorly suited to cloud environments. Our position is that cloud memory caches must be dynamically tuned based on application access patterns, hit rate requirements, and operator policy. Like any other critical resource (e.g., compute, storage, network bandwidth), memory caches require intelligent *application-aware* resource management and policy.

We observe that cloud memory caches are very different from CPU caches in terms of workloads and capabilities. On the one hand, cloud application request patterns are significantly more *variable* than CPU workloads, which access fixed-sized cache lines, and often sequentially due to inherent program structures like loops. On the other hand, cloud caches are much more *capable* than CPU caches. Unlike CPU caches, free from

[1]To avoid memory fragmentation, Memcached divides its memory into several *slabs*. Each slab stores items with size in a specific range (e.g., < 128B, 128-256B, etc)

the restrictions of hardware implementation at GHz frequencies, a memory caching system can use sophisticated application profiling and optimization to adapt to the unique characteristics of each application.

While a dynamic application-aware cache is conceptually appealing, is it worth the additional complexity in practice? We answer this in the affirmative by analyzing of a week-long trace of over 490 web applications at Memcachier [1], a popular Memcached-as-a-service platform. We find significant room for improvement over the application-oblivious baseline cache system. Specifically, Memcached's default first-come first-serve slab memory allocation performs very poorly for some applications. In some cases, the default slab class allocations result in over 3× more misses than could have been achieved with an optimal allocation of the same amount of memory. For the same hit rate, default Memcached sometimes requires over 2.5× more memory.

Our trace analysis demonstrates the need for a cache that can dynamically adjust its memory allocation to satisfy the varying demands of different applications. To this end, we develop Dynacache, a lightweight, minimally invasive, application-aware cache controller for cloud environments. Dynacache detects the applications that are most likely to benefit from optimization using a simple, easy-to-compute entropy metric. It then profiles the request pattern of the selected applications and optimizes their slab allocation to maximize hit rate.

We have built a prototype implementation of Dynacache in C for Memcached. Our preliminary evaluation with realistic workloads based on measurements at Facebook [5] shows that Dynacache's application profiling is feasible and has a low overhead on the performance of other applications.

Our current design focuses on slab memory allocation as evidence of how rigid cache policies hurt performance. However, we believe that other parameters such as eviction policy and memory allocation across applications also hold great promise for improvement. Further, we expect that the output of our application profiling may inform much more intelligent decisions by developers and operators about their cache needs and resource sharing policy, which today must be decided mostly by guesswork. We intend to explore these directions in future work.

3. MEMCACHIER TRACE ANALYSIS

In this section, we provide empirical evidence that the one-size-fits-all resource allocation policies of Memcached are not ideal for real web applications that exhibit variable behavior. We collected a trace of 490 applications on Memcachier, a multi-tenant Memcached service. Each application reserves a certain amount of memory in advance, which is uniformly allocated across multiple Memcached servers comprising the Memcachier cache. We obtained packet captures on one of the Memcachier servers for a week. In all, the raw captures amount to 220 GB of compressed data. We analyzed the packet captures to reconstruct all Memcached requests and responses, including the type of request (e.g., `GET`, `SET`), key and value sizes, whether the request hit or missed, and the slab class for each request.

App	% of Trace	Old Hitrate	New Hitrate	% Miss Reduction	Additional Memory Required with Default Alloc.	Miss Entropy
1	36.8%	69.6%	75.3%	18.6%	20%	0.49
2	8.6%	99.9%	99.9%	0%	0%	0.04
3	5.7%	97.5%	98.7%	51.1%	133%	2.71
4	4.2%	97.5%	97.7%	8.4%	14%	0.01
5	3.6%	98.3%	99.4%	65.2%	145%	2.40
6	3.4%	99.8%	99.8%	0%	0%	0.19
7	2.9%	29%	32.6%	4.99%	70%	0.56
8	2.7%	99.9%	99.9%	0%	0%	0.15
9	2.4%	93.9%	93.9	0%	0%	1.55
10	2%	99.58%	99.63%	11.9%	14%	0.06

Table 1: Analysis of the affect of optimizing the slab class allocation for the top 10 applications in the Memcachier trace. The Miss Entropy is a metric for classifying which application will benefit most from optimizing its slab class allocation, and the last column in the table depicts the amount of memory required to achieve the improved hit rate using the default slab class allocation of Memcachier.

In particular, we focus on one dimension of cache resource allocation, namely, the allocation of memory resources across slab classes for each application. In Memcached, memory pages are divided into 1MB pages assigned to one of several slab classes. Each slab class stores items whose sizes are within a specific range (e.g., between 1KB and 2KB). For example, a 1MB page that belongs to the 1-2KB slab class, would be split into 512 chunks of 2KB in size. Each slab class has its own LRU eviction queue per application.

In Memcachier, memory is allocated to slab classes greedily based on the sizes of the initial items at the beginning of the workload. Since each slab class has its own eviction queue, the length of the queues can vary greatly among slab classes, and some slab classes may be under or over provisioned.

Table 1 presents the hit rate achieved by Memcachier for the top 10 applications (sorted by number of requests) in our trace, and the hit rate that could be achieved with an optimal allocation of memory to slab classes (we will show how to derive the optimal slab class allocation in Section 4).

The results show that some applications can benefit greatly from better slab class allocation, and some do

App	Slab Class	% GETs	Original Hitrate	Original Allocation	Optimal Hitrate	Optimal Allocation
4	0	9%	99.98%	1.38MB	98.11%	0.72MB
4	1	91%	97.32%	4.39MB	97.66%	5.05MB
3	0	9%	98.18%	0.002MB	99.99%	0.25MB
3	1	22%	96.71%	0.002MB	99.98%	0.56MB
3	3	3%	97.27%	0.017MB	99.99%	0.67MB
3	4	45%	98.45%	0.25MB	99.31%	2.27MB
3	5	6%	99.70%	0.016MB	100%	0.55MB
3	6	1%	97.08%	0.07MB	99.95%	1.06MB
3	7	5%	98.89%	0.18MB	99.97%	2.22MB
3	8	6%	98.90%	0.42MB	99.97%	4.24MB
3	9	1%	69.35%	3.75MB	17.16%	0MB
3	10	1%	82.62%	11.13MB	94.51%	15.72MB
3	11	2%	93.12%	32.75MB	77.09%	21.05MB

Table 2: Hit rate and slab class allocations of two applications in the Memcachier trace. In Memcachier, each slab class size is: $64bytes \cdot 2^{SlabId}$. So for example, slab class 0 is for items with value between 0-64 bytes, while slab class 1 is for values between 64-128 bytes. The slab class optimization improved the overall hit rate of application 4 from 97.5% to 97.7%, and of application 3 from 97.5% to 98.7%.

not. For example, the number of misses in applications 3 and 5 is reduced by 51% and 65% respectively. About half of the applications in the trace do not benefit at all from optimized slab allocation, because applications are typically well over provisioned. In some applications (like application 7), while the optimal slab class allocation does improve the hit rate, the improvement is small. The table also shows the amount of memory that would have been required to achieve the improved hit rate, using the default slab allocation policy. For applications 3 and 5, Memcachier must more than double the amount of allocated memory to achieve the same hit rate with the default policy.

The problem with Memcachier's first-come-first-serve memory allocation to slab classes is that if some applications change their request distribution, some slab classes may not have enough memory and their eviction queues will be too short. Some Memcached implementations have tried to solve this problem by periodically evicting an entire slab of a slab class, and reassigning it to the corresponding slab class of new incoming requests [13, 14]. However, even these improved slab class allocation may suffer from sub-optimal slab class allocation. For example, they may tend to favor large slab classes over small ones, because if a request for a large item size is received at the server, it will be treated equally as a request with a small item size, even though the large item size occupies much more memory in the Memcached server.

This is demonstrated by Table 2, which provides an example of the default hit rate and the slab class allocation of two Memcachier applications. In both applications, Memcachier allocates relatively much more memory to the larger slab classes, compared to the number of requests from each slab class. It is clear that in both applications, the Memcachier slab class allocation favors large slab classes relative to the number of times they are requested.

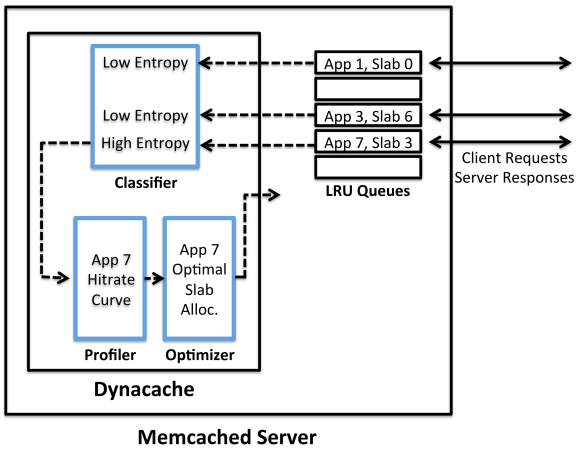

Figure 1: High level architecture of Dynacache.

Intuitively, we expect that in application 3 this problem will be more severe, since the requests are more evenly distributed among the slab classes. In application 4, since the vast majority of the requests originate in slab class 1, the default slab class allocation will perform reasonably well. This is the reason some applications' performance can be improved more than others with an optimal slab class allocation.

4. DESIGN

In this section we describe the design of Dynacache. In order to tailor the resource allocation of Memcached for different applications, Dynacache first classifies and profiles the applications' access patterns, and then computes and sets their optimal slab class allocation. The architecture of Dynacache is depicted in Figure 1. Dynacache runs as a module that integrates with the Memcached server. We describe its three main components below, starting with the optimizer, then the profiler and finally the classifier.

4.1 Slab Class Allocation Optimization

We first describe how to compute the optimal memory allocation to each slab class for a given application. The problem can be expressed by the following optimization problem:

$$\begin{aligned} \underset{m}{\text{maximize}} \quad & \sum_{i=1}^{s} f_i h_i(m_i, e) \\ \text{subject to} \quad & \sum_{i=1}^{s} m_i \leq M \end{aligned} \tag{1}$$

Where s is the number of slab classes, f_i is the frequency of GETs for each slab class, $h_i(m_i, e)$ is the hit rate of each slab class as a function of the its available memory (m_i) and cache eviction policy (e), and M is the amount of memory reserved by the application on the Memcached server.

In order to solve this optimization problem, we need to compute $h_i(m_i, e)$, or the *hit rate curve* for each slab class. Stack distances [12] provide a convenient means of computing the hit rate curve beyond the allocated memory size, for a given eviction policy. The stack distance of a requested item is its rank in the cache, counted from the top of the eviction queue. For example, if the requested item is at the top of the eviction queue, its stack distance is equal to 1. If an item has never been requested before, its stack distance would be infinite.

The stack distances can be computed offline for each incoming request in the Memcachier trace. This allows us to compute the hit rate curve as a function of the memory allocated to each slab class. For example, Figure 2 depicts the hit rate curve of slab class 9 of application 3.

To solve the optimization in Equation (1) efficiently, we need to approximate the hit rate curves as concave functions. Fortunately, hit rate curves in the Memcachier traces are concave or nearly concave to begin with. We use convex piecewise-linear fitting [11] to approximate the hit rate curves. Given the piecewise-linear hit rate curves, the optimization (1) can be solved using a simple LP solver.

4.2 Practical Profiling

Computing the exact stack distance for each incoming request requires maintaining a "shadow" eviction queue and tracking the position of every incoming request in the queue to calculate its stack distance. This can be computationally prohibitive, especially when the application accesses a large number of unique keys (the computation is $O(n)$, where n is the size of the shadow queue).

Instead, Dynacache uses a bucketing scheme similar to Mimir [15]. In this bucketing scheme, instead of keeping track of a shadow eviction queue, there is a linked list of buckets, each containing a fixed number of items. Each incoming request enters the top bucket, and when the top bucket is filled, we remove the bucket at the end of the queue. We maintain a hash function that maps each item to the bucket in which it is stored. We can estimate the stack distance of an incoming request, by summing the size of all buckets that appear in the queue before it, and adding it to the size of its own bucket divided by 2. This stack distance computation algorithm is much faster than the naïve method, since its complexity is $O(B)$, where B is the number of buckets. The difference in the hit rate improvement for the optimized slab class allocation based on the bucket algorithm versus exact stack distances is less than 10% for all the applications we analyzed.

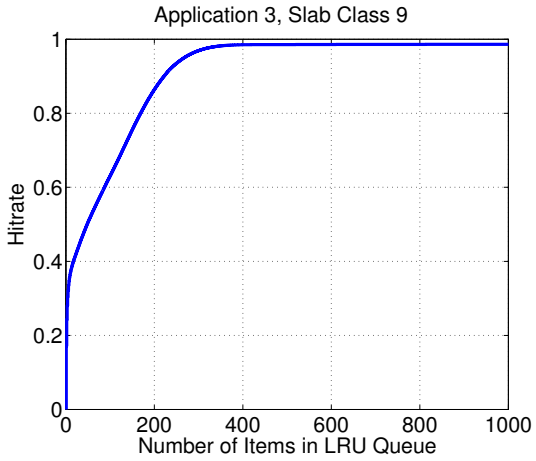

Figure 2: Hit rate curve over an entire week for Application 3, Slab Class 9 (item sizes of 16-32 KB).

4.3 Which Applications to Optimize?

The naïve approach would be to simply calculate the hit rate curves of all the applications and run the optimization function. However, estimating the hit rate curves for hundreds of applications on each server is costly. As our results in Table 1 have shown, not all applications benefit from optimized slab class allocation.

Dynacache should ideally only optimize the slab class allocation of application that are likely to benefit. To this end, we derive a metric that is simple to compute and with a high degree of certainty predicts the hit rate improvement due to optimized slab class allocation.

Intuitively, the default slab class allocation will perform poorly when there is a relatively uniform distribution of unique requests among slab classes. When all the requests are concentrated on a small number of adjacent slab classes (or in a single slab class), there won't be a big difference between a first-come-first-serve slab allocation policy and the optimal slab allocation policy.

Entropy is a metric that provides a measure of the uniformity of a distribution function. When a probability function behaves completely uniformly, its entropy will be the highest, and when it behaves deterministically, its entropy will be 0. The last column in Table 1 shows the *Miss Entropy* across slab classes for the different applications in the trace. The Miss Entropy is calculated by

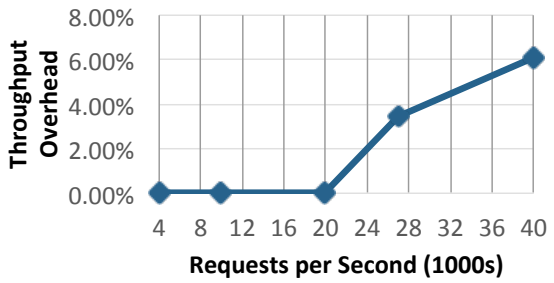

Figure 3: Throughput overhead of Dynacache profiler. The overhead remains 6% after 40,000 requests per second, because the Memcached server becomes saturated.

treating the misses per slab class as a probability density function and calculating its entropy.

The table shows that in general, the Miss Entropy provides a strong indicator for applications that would benefit from optimal slab class allocation. The only exception to this rule is application 9. This is because Memcachier allocated a very small amount of memory (only 1.6MB) to application 9. The optimization function does not work with a very small number of data points, and therefore cannot improve the slab allocation, given the memory constraints. If application 9 had been allocated more memory, it would have benefited significantly from an improved slab class allocation, similar to applications 3 and 5.

5. EVALUATION

We implemented a prototype of the Dynacache profiler in C, and integrated it with Memcached 1.4.22. Our implementation consists of about 170 code lines. In order to measure the overhead added by Dynacache, we leveraged Mutilate[2], a load generator that emulates the Memcached workloads from the 2012 Facebook study [5]. We used the same key, value and read/write distributions as described in the Facebook paper. We used the Facebook workload because it is much more CPU intensive than the applications measured in the Memcachier trace. We ran our simulation on an Intel Xeon E5-2670 system with 32 GB of RAM and an SSD drive, using 5 minutes experiments. We measured the achieved throughput and latency overhead while running the Dynacache profiler, under different request loads.

We examined the throughput and latency achieved using different request loads, ranging from 4000 requests per second (Memcachier's average load) to 100,000 requests per second. Our evaluation shows an average of 5.8% latency slowdown for read queries (GETs) and 9.6% latency slowdown for write queries (SETs), with negligible deviations between the experiments.

[2]https://github.com/leverich/mutilate/

Figure 3 presents the throughput overhead. In order to measure throughput, we generated a series of requests at the client, and measured the number of requests returned during a 5 minute period. The figure shows that throughput was not affected at lower loads, but had an overhead of 6% compared to the default Memcached implementation, once Memcached became CPU bounded. The overhead remained at 6% after 40,000 requests per second, because the Memcached server becomes saturated. Note that in the case of Memcachier, Memcached is memory bound and not CPU bound, and therefore the Dynacache would not impose any overhead on throughput. Initial profiling shows that further optimizations can be made in our implementation by reducing the number of hash computations in cases of updates and bucket deletions, and by running the profiler asynchronously (i.e., not in the critical path of incoming requests).

6. RELATED WORK

This work is related to previous work on improving and profiling the performance of Memcached. Mimir [15] and Blaze [6] also profile cache hit rate curves to enforce QoS guarantees in multi-tenant web caches. Similarly, Wires et. al. profile hit rate curves using Counter Stacks [16] in order to better provision Flash based storage resources. In addition, Hwang et. al. have proposed a dynamic hashing system [10] that can evenly distribute requests across servers, taking into account varying item sizes. A recent study on the Facebook photo cache demonstrates that modifying LRU can significantly improve web cache performance [9]. Twitter [14] and Facebook [13] have tried to improve Memcached slab class allocation to better adjust for varying item sizes, by periodically evicting slabs that exhibit a high number of misses. However, both of these schemes are far from optimal, since they do not take into account the hit rate curves across all the slab classes. There is a body of prior work on algorithms for calculating stack distances in the context of CPU caches [4, 7, 12, 17]. These techniques may be applicable for Dynacache to enhance the performance of the profiler.

7. CONCLUSION

By analyzing a multi-tenant Memcached cluster, we demonstrated that a web-based cache can be improved significantly by tuning its behavior to dynamically adjust to the requirements of different applications. We showed that the performance of certain applications can be significantly improved by simply better allocating the memory slab allocation within the Memcached servers, without interfering with the data path of the cache. Our next step is to generalize dynamic tuning to the other parameters in cache systems such as relative allocations across applications and eviction policies.

8. DISCUSSION TOPICS

Our paper makes an argument for designing memory caches that can dynamically tune their behavior. We motivate the design by analyzing applications in a multi-tenant memory cache environment (Memcachier), and by providing initial results that demonstrate the high degree of variability in slab allocation across applications. Our goal is on the one hand, to preserve the simple interface and deployment of Memcached, but on the other hand allow it to be dynamically tuned to the different requirements of web-scale applications.

We would like to receive feedback on the general concept of a dynamic web-scale caching system, and on particular challenges we may face when implementing Dynacache in a real-world setting. In addition, we plan to explore how the eviction policy affects the performance of different applications, and would be interested to hear feedback on the experience of workshop participants of working on cache eviction algorithms, both for web-scale caches and for CPU caches.

References

[1] `www.memcachier.com`.

[2] `redis.io`.

[3] `aws.amazon.com/elasticache/`.

[4] G. Almási, C. Caşcaval, and D. A. Padua. Calculating stack distances efficiently. In *ACM SIGPLAN Notices*, volume 38, pages 37–43. ACM, 2002.

[5] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review*, volume 40, pages 53–64. ACM, 2012.

[6] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59. ACM, 2013.

[7] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices*, volume 38, pages 245–257. ACM, 2003.

[8] B. Fitzpatrick. Distributed caching with memcached. *Linux journal*, 2004(124):5, 2004.

[9] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An analysis of facebook photo caching. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 167–181. ACM, 2013.

[10] J. Hwang and T. Wood. Adaptive performance-aware distributed memory caching. In *ICAC*, pages 33–43, 2013.

[11] A. Magnani and S. P. Boyd. Convex piecewise-linear fitting. *Optimization and Engineering*, 10(1):1–17, 2009.

[12] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[13] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, et al. Scaling memcache at facebook.

[14] M. Rajashekhar and Y. Yue. `blog.twitter.com/2012/caching-with-twemcache`.

[15] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing*, pages 1–14. ACM, 2014.

[16] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data. Characterizing storage workloads with counter stacks. In *Proceedings of the 11th USENIX conference on Operating Systems Design and Implementation*, pages 335–349. USENIX Association, 2014.

[17] Y. Zhong, S. G. Dropsho, and C. Ding. Miss rate prediction across all program inputs. In *Parallel Architectures and Compilation Techniques, 2003. PACT 2003. Proceedings. 12th International Conference on*, pages 79–90. IEEE, 2003.