Saving Cash by Using Less Cache

Timothy Zhu, Anshul Gandhi, Mor Harchol-Balter[*]
Carnegie Mellon University

Michael A. Kozuch
Intel Labs

Abstract

Everyone loves a large caching tier in their multi-tier cloud-based web service because it both alleviates database load and provides lower request latencies. Even when load drops severely, administrators are reluctant to scale down their caching tier. This paper makes the case that (i) scaling down the caching tier is viable with respect to performance, and (ii) the savings are potentially huge; e.g., a 4x drop in load can result in 90% savings in the caching tier size.

1 Introduction

With the advent of cloud computing, web service providers have the ability to dynamically scale their computing infrastructures to match demand. Further, because cloud resources are often priced per-use, web service providers have a monetary incentive to minimize the number of resources consumed while still meeting the Service Level Agreements (SLAs) of the service.

Web services are often composed of multiple tiers; a common example is shown in Figure 1. The frontend application tier consists of a set of stateless application servers that process requests using data from the backend. The backend data tier consists of a persistent storage system, such as a database. To alleviate load at the backend, a stateful, but non-persistent, distributed caching tier is often used to cache data or partial results.

Each tier must be treated differently when scaling. The application tier is the easiest tier to scale down because it is stateless [9, 7]. In contrast, because web services often impose stringent data availability requirements, options for scaling the data tier are typically limited to adjusting the replication factor of the storage system [4, 14].

Note that the tiers are typically of different sizes, with the servers in the application tier often outnumbering servers in the caching tier at a ratio somewhere around 4:1. While this fact might initially suggest that scaling the caching tier may not lead to significant savings, note that DRAM is an expensive resource. For example, if we were to instantiate our testbed of 28 servers on EC2 [3], our caching tier would represent 37%[1] of the operational cost, despite the fact that our caching tier only comprises 5 servers. Further, at times of lower utilization, after the application tier has been scaled down, an unscaled caching tier becomes a more significant fraction of the service's operational cost. In the above example, if load were to drop by a factor of 4, and we were to scale down our application tier from 20 servers to 5 servers, our caching tier would then represent 63%[1] of the operational cost.

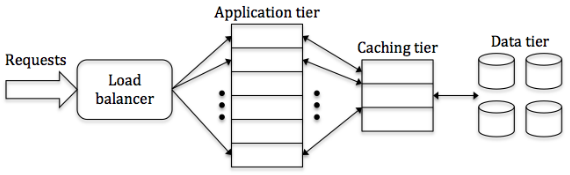


Figure 1: Multi-tier cloud service.

Scaling down the number of cache instances as the load decreases, therefore, could provide cost benefits. However, two technical concerns come immediately to mind. First, as the caching tier scales down, the amount of cached data decreases. Will the cache misses that result overwhelm the data tier? Second, how should the caching tier be managed so that "hot" data is preserved when cache instances are removed and distributed when cache instances are added?

The *key insight* in answering the first question is that when the overall load drops, we can afford a higher fraction of requests going to the data tier; hence, we can tolerate a lower cache hit rate, and this lower cache hit rate translates to a vast reduction in the amount of data cached. To correctly size the caching tier, we work backwards – when the load drops, we determine the minimum cache hit rate needed to ensure that the response time SLA is met (thereby limiting how many requests go to the data tier). We then calculate the cache size that would provide that hit rate. A key parameter that determines the degree to which the cache capacity may be reduced without overwhelming the data tier is the distribution of requests. If the requested data items are uniformly distributed, the degree to which cache capacity may be reduced is small. However, many studies have shown that web requests follow a very skewed distribution (often modeled as a Zipf distribution [6, 2]). In such cases, we will show that significant savings are possible. In Figure 2, we see that a relatively small change in the required hit rate – from 0.95 to 0.8 – may result in a substantial reduction in the cache size needed under a Zipf distribution (79% to 34% of the data), but a more limited reduction (95% to 80%) if the distribution is Uniform.

[*]This work is supported by NSF CSR Grant 1116282 and an Intel ISTC grant.

[1]Assuming application servers cost \$0.165/hr/high-cpu instance [3] and cache servers cost \$0.45/hr/high-memory instance [3].

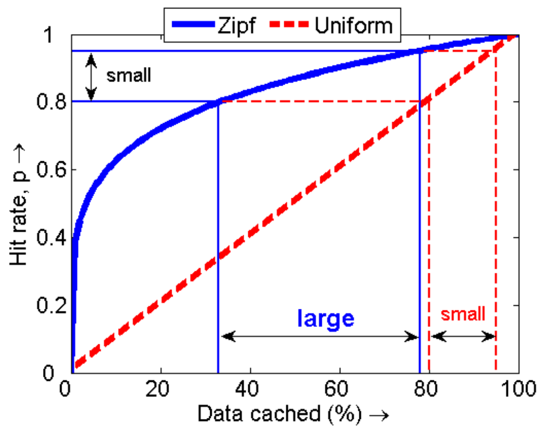


Figure 2: A small decrease in cache hit rate can lead to a large decrease in the amount of cached data.

In answering the second question regarding cache management, we first rely on consistent hashing techniques to ensure that excessive data migration is not needed as instances are added and removed from the caching tier. However, even with consistent hashing, naïvely adding or removing a cache instance can cause significant performance problems, because the hot data is distributed over all instances. In Section 4, we provide a simple solution that reduces the number of cache misses during periods of scaling by redistributing cached items.

Our contributions are:

- We demonstrate that the caching tier can be effectively scaled with load without violating performance goals.
- We develop a model to calculate (i) the required hit rate as a function of load, and (ii) the resulting savings in the size of the caching tier from scaling down during low load (see Section 3).
- We validate our theoretical results via implementation on a 28-server testbed (see Section 3.4).
- We propose a *simple* cache management mechanism to redistribute cached items as the caching tier scales up and down, significantly reducing transient cache misses.

2 Experimental setup

We experiment with a testbed of 28 commodity servers, which are divided into multiple tiers as in Figure 1. We employ one of these servers as the load generator running httperf [11]. Another server is used as the load-balancer running Apache™ HTTP Server, which distributes PHP requests from the load generator to 20 application servers. The application servers (Intel® Xeon® E5520 processor-based) parse the incoming PHP requests and collect the required data from the caching tier and the data tier. In our experimental setup, we use a distributed cache, memcache [10]. The caching tier comprises 20 memcache instances, each with up to 10GB of memory for caching, hosted on 5 servers (Intel® Xeon® X5650 processor-based). The data tier comprises a server (Intel® Xeon® E5520 processor-based) with 5 disks running an Oracle® BerkeleyDB [12] database with a billion key-value pairs (250GB).

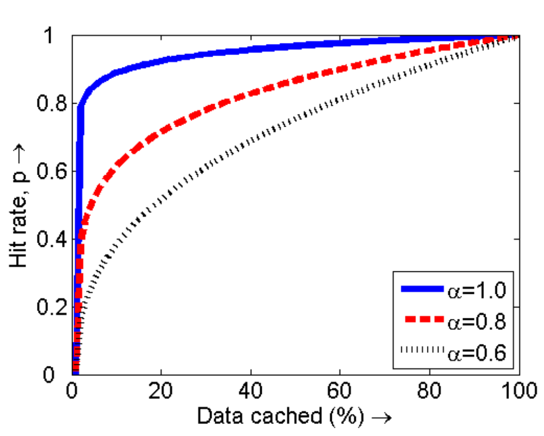


Figure 3: The Zipf popularity distribution.

Each workload (HTTP) request is a PHP script that runs on the application server, and consists of 20 independent key-value fetches. The fetched keys follow a Zipf [6, 2] distribution. Each of the 20 key-value fetches either hits in the memcache, or, if it misses, goes to the database. If a key-value fetch hits in the memcache, its response time is $T_{mem} = 0.3ms$, which is the time to retrieve a key-value pair from memcache. If a key-value fetch goes to the database, its response time is on the order of 8ms, depending on the contention at the database. In this paper, our performance goals take the form of response time SLAs: we require that the average response time for the entire request (collection of 20 key-value fetches), which we refer to as T_{avg}, should be no more than $T_{SLA} = 100ms$. Thus, we require:

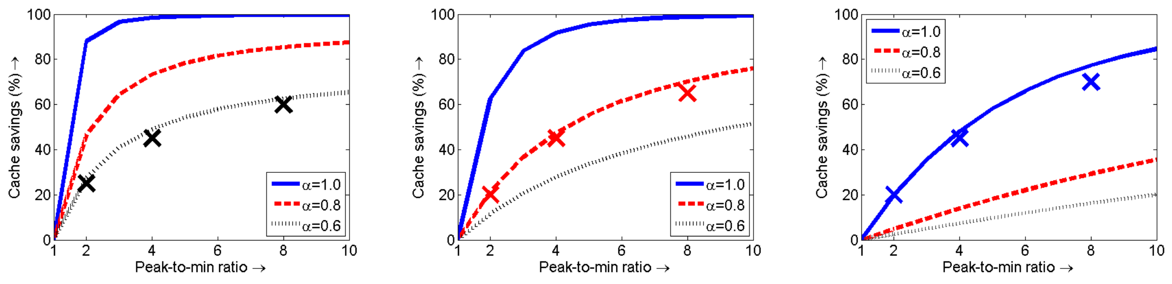
$$T_{avg} \leq T_{SLA} = 100ms. \qquad (1)$$

3 How much cache can we save?

In this section, we investigate the potential savings that can be achieved by scaling down the caching tier, without violating the response time SLA. We first describe our theoretical framework that allows us to estimate these savings, and then report our experimental results, which validate our estimates.

3.1 Popularity distribution

Many websites report that their data popularity distribution is far from being Uniform or Random, and is often very skewed (that is, a small subset of the entire data set is responsible for most of the traffic) [13, 6, 2, 8]. Assuming that we cache the most popular data items for a given



(a) $p_{peak} = 0.80$ (b) $p_{peak} = 0.95$ (c) $p_{peak} = 0.99$

Figure 4: Theoretical savings in cache for a given peak-to-min ratio. Crosses indicate experimental results.

popularity distribution, we can estimate the amount of data to be cached to achieve a given hit rate.

Researchers often use a Zipf distribution to represent the website popularity distribution [13, 6, 2]. Mathematically, if the data items are sorted in decreasing order of popularity, the Zipf distribution states that the probability of seeing a request for data item i, $Pr\{i\}$, is:

$$Pr\{i\} = \frac{C}{i^{\alpha}}, \quad (2)$$

where α is a parameter of the distribution, and C is a normalization constant. For real-world website traffic, α is typically between $0.6 - 1.0$ [13, 6]. A higher value of α corresponds to a more skewed distribution. We can now use Equation (2) to estimate the amount of data to be cached to achieve a given hit rate, p. Figure 3 shows these results for a Zipf popularity distribution with a range of α values. When α is high, say $\alpha = 1.0$, we can achieve a hit rate of $p = 0.8$ by caching only 2% of the data. However, when α is low, say $\alpha = 0.6$, we need to cache almost 60% of the data to achieve the same $p = 0.8$ hit rate. For a Uniform distribution ($\alpha = 0$), we would have to cache 80% of the data to achieve $p = 0.8$ hit rate.

3.2 Theoretical model

In Figure 3, we saw the relationship between the hit rate, p, and the amount of data cached. We now investigate the relationship between a given response time SLA, T_{SLA}, and the *minimum* required hit rate, p. This allows us to calculate the amount of cache required to meet T_{SLA}. The relationship between T_{SLA} and p is given by:

$$T_{avg} = 20\,(p \cdot T_{mem} + (1-p) \cdot T_{DB}) \leq T_{SLA} = 100ms, \quad (3)$$

where $T_{mem} = 0.3ms$ is the latency for fetching a key-value pair from memcache, and T_{DB} is the latency for fetching a value from the database. Here, we use the fact that each request in our system is composed of 20 individual key-value fetches.

Note that T_{DB} in Equation (3) is not a constant, and depends on the request rate into the database, λ_{DB}. As λ_{DB} increases, so does T_{DB}. We model this relationship using an $M/M/1$ queueing model with load-dependent service rates. This gives us T_{DB} as a function of λ_{DB}.

If λ denotes the total arrival rate of requests into the system, then λ_{DB} is:

$$\lambda_{DB} = 20 \cdot \lambda \cdot (1-p). \quad (4)$$

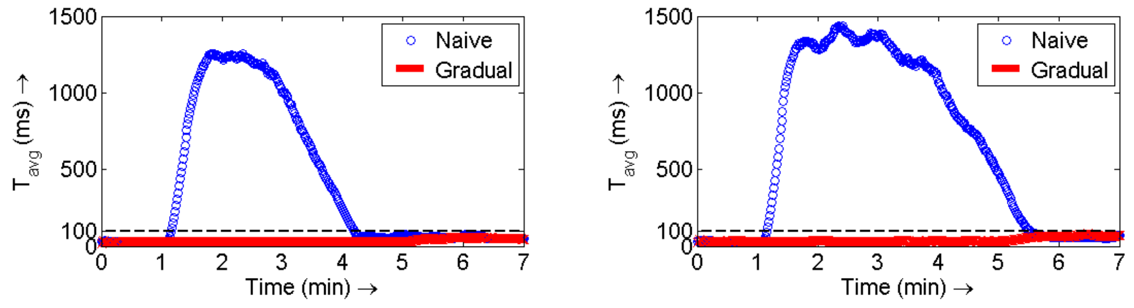
If the request rate into the database, λ_{DB}, is too high, then T_{DB} grows to infinity. Thus, it is important to limit the request rate to the database by keeping $(1-p)$ small (the hit rate, p, should therefore be high). Also, since T_{mem} is orders of magnitude smaller than T_{DB}, it is desirable to keep p high so as to make T_{avg} less than T_{SLA}. While both these goals point towards making p as *high* as possible, our objective in this paper is to find the *lowest possible* p which still meets the desired T_{SLA} constraint (this is because low p implies low cache size from Figure 3).

Given the arrival rate into the system, we can solve Equation (3) for p, which in turn is used to calculate the desired cache size from Figure 3.

3.3 Theoretical results

Web services often exhibit huge variations in their request rates, mostly due to the diurnal/periodic nature of traffic. Assuming that the system is well provisioned to meet T_{SLA} for peak request rate, we are interested in the potential for decreasing the cache size when the request rate decreases. We refer to the ratio between the peak request rate and the lower request rate as the peak-to-min ratio. We pick a range of peak-to-min ratios $(1-10)$ and use our theoretical model to calculate the possible reduction in cache size. We also show results across different values of α: 1.0 (heavily skewed), 0.8, and 0.6 (less skewed). Lastly, we vary the hit rate at the peak request rate: $p_{peak} = 0.80$ (Figure 4(a)), $p_{peak} = 0.95$ (Figure 4(b)), and $p_{peak} = 0.99$ (Figure 4(c)). We choose these parameter values based on recent studies [13, 5, 6, 2].

Figure 4 indicates that *significant cache reductions (up to 90%) are possible even for a 2:1 peak-to-min ratio*. In general, *the potential savings in cache size are higher when α is high*. This is because a higher value of α indicates a more skewed popularity distribution, which allows for more aggressive reduction in cache size (see Figures 2 and 3). Also, *the observed potential savings*



(a) Remove memcache instance (b) Add memcache instance

Figure 5: Gradual is significantly better than Naïve for (a) scaling down and (b) scaling up the caching tier. Here, the dashed line indicates $T_{SLA} = 100ms$.

in cache size are higher when p_{peak} is low. This can be explained as follows. If the request rate into the system, λ, drops by a factor of (say) 2, then Equation (4) tells us the miss-rate, $(1-p)$, can increase by a factor of 2 while maintaining the same request rate into the database, λ_{DB}. For example, when the peak hit rate is low, say, $p_{peak} = 0.8$, the miss-rate $(1-p_{peak}) = 0.2$. If the request rate now drops by a factor of 2, our final hit rate can be as low as $p = 0.6$, since the miss-rate, $(1-p)$, can now be as high as $2\cdot(1-p_{peak}) = 0.4$. However, when the peak hit rate is $p_{peak} = 0.99$ and the request rate drops by a factor of 2, our final hit rate can only be as low as $p = 0.98$. This implies that when the peak hit rate is low, we can afford a larger drop in hit rate, which in turn implies larger cache savings via Figure 3.

These large cache savings translate to huge cash savings. Consider an application hosted on EC2 [3] requiring a 4:1 ratio between application instances (costing \$0.165/hr/high-cpu instance [3]) and cache instances (costing \$0.45/hr/high-memory instance [3]). Suppose load drops by a factor of 4. If we only scale the application tier (by a factor of 4), then we can save 45% of the peak operational cost. But if we also scale the caching tier (assuming a modest 50% cache reduction based on Figure 4), then we can save 65% of the peak operational cost. This is an additional 37% savings relative to only scaling the application tier.

3.4 Experimental results

In order to validate our theoretical results, we experimentally determine the lowest memcache size that we can afford without violating our SLA. We do this by monitoring the mean response time, T_{avg}, for different memcache sizes, and then picking the smallest memcache size which keeps T_{avg} below T_{SLA}. The crosses in Figure 4 show our experimental results, which agree with the theoretical results.

We also investigated 99th percentile response times and preliminary results show similar cache savings as in Figure 4.

4 Managing the caching tier

Thus far, we have investigated how small the cache can be, assuming it contains the most popular data items. However, it is not obvious how to retain the most popular data items when scaling the caching tier because the cached objects are distributed over the cache instances for good load balancing. When the number of cache instances is reduced, popular items may be lost. We use consistent hashing in the libMemcached library [1] to ensure that only the data in the newly added/removed cache instance needs to change.

4.1 Scaling down

If we naïvely remove a cache instance, we immediately lose all the cached data that was stored on the instance. This causes a drop in hit rate, which increases the request rate to the data tier as well as the average response times. For example, consider the case in Figure 5(a) where we scale down from 4 to 3 cache instances at the 1 minute mark. In this experiment, we choose $\alpha = 1.2$ to limit the amount of cached data that is lost. We also pessimistically force cache misses to result in database disk accesses by avoiding the database page cache. As observed in Figure 5(a), the Naïve solution (the circles) creates a spike in average response time.

To avoid this spike in response times, we propose *gradually* migrating the most popular data off of the instance to be removed. Conceptually, we treat the instance to be removed as a second level cache for some period of time. If we miss in the rest of the cache, then we query this instance before going to the database. This will naturally migrate the popular items from this "retiring" instance to the rest of the caching tier.

This leads to the question of how long to keep an instance in this retiring state. Intuitively, we want to stay in this retiring state until the probability of querying this instance is low enough that we can achieve our target hit rate, as calculated from Equation (3), even after this instance is removed. For each item i residing on this instance, let p_i denote the probability of requesting that item. Suppose we have received N requests since entering this retiring state. Then the probability that item i has not yet migrated to the rest of the caching tier is $(1-p_i)^N$. Thus, the probability that the $(N+1)^{th}$ request queries this instance is:

$$\sum_{\text{item } i \in \text{instance}} p_i(1-p_i)^N. \tag{5}$$

Assuming all items are equally distributed among the instances, we can reasonably predict how large N needs to be so that the probability of querying this instance is low enough to achieve our target hit rate.

In Figure 5(a), we see that the Gradual solution (solid line) eliminates the spike in response times. Here, the retiring instance acts as a "second-level cache" between the 1 minute and 5 minute marks, approximated by Equation (5), at which point it is entirely removed.

4.2 Scaling up

If we naïvely add a cache instance, it has a "cold" cache and all requests to that instance result in misses. For example, consider the case in Figure 5(b) where we scale up from 3 to 4 cache instances at the 1 minute mark. We see that the Naïve solution (the circles) creates a spike in average response time.

To avoid this spike in response times, we propose gradually migrating the most popular data from the rest of the caching tier to this new instance. That is, when we miss in this new instance, we query the rest of the caching tier. This will naturally warm up the "hot" data without needing to go to the database. In Figure 5(b), we see that the Gradual solution (solid line) eliminates the spike in response times. Here, the new instance warms up between the 1 minute and 5 minute marks, approximated by Equation (5), using the rest of the caching tier.

Importantly, using the Gradual solution for both adding and removing a cache instance keeps our response times below the 100ms SLA.

We also find the same behavior when looking at 99^{th} percentile response times. The 99^{th} percentile response time under the Naïve solution increases to about 16 seconds, but stays below 700ms under the Gradual solution.

4.3 Alternative approaches

While our proposed solution is very simple, additional benefits may be realized through more dramatic re-architecting of the caching tier. For example, by exposing the LRU lists of cached keys at each memcache instance, more intelligent adding/removing of cache instances may be possible. Further, integration of non-volatile storage, such as flash, may allow removing a cache instance without requiring the migration of hot data from other instances on reactivation. However, this approach will require additional functionality to determine the consistency of cached data upon reactivation.

5 Conclusion

While scaling of the stateless application tier has been proposed in many research papers, there has been almost no discussion of scaling the stateful caching tier. The "seemingly small" benefits associated with scaling the caching tier coupled with the fear of severe performance degradation due to cache misses has deterred this research. In this paper we demonstrate that, given the skewed popularity distribution for data accesses, significant cost savings can be obtained by scaling the caching tier under dynamic load patterns (see Section 3). Furthermore, performance problems associated with scaling the stateful caching tier can be avoided via a simple Gradual algorithm (see Section 4). By combining our stateful scaling solutions with existing stateless scaling solutions, one can realize fully load-proportional cloud systems.

References

[1] libmemcached. http://libmemcached.org/libMemcached.html.

[2] ALMEIDA, V., BESTAVROS, A., CROVELLA, M., AND DE OLIVEIRA, A. Characterizing reference locality in the www. In *DIS 1996* (Miami Beach, FL, United States), pp. 92–107.

[3] AMAZON INC. Amazon elastic compute cloud (Amazon EC2). http://aws.amazon.com/ec2/.

[4] AMUR, H., CIPAR, J., GUPTA, V., GANGER, G. R., KOZUCH, M. A., AND SCHWAN, K. Robust and flexible power-proportional storage. In *SOCC 2010* (Indianapolis, IN, USA), pp. 217–228.

[5] BEAVER, D., KUMAR, S., LI, H. C., SOBEL, J., AND VAJGEL, P. Finding a needle in haystack: facebook's photo storage. In *OSDI 2010* (Vancouver, BC, Canada), pp. 1–8.

[6] BRESLAU, L., CAO, P., FAN, L., PHILLIPS, G., AND SHENKER, S. Web caching and zipf-like distributions: evidence and implications. In *IEEE INFOCOM 1999* (New York, NY, USA), pp. 126–134.

[7] CHEN, G., HE, W., LIU, J., NATH, S., RIGAS, L., XIAO, L., AND ZHAO, F. Energy-aware server provisioning and load dispatching for connection-intensive internet services. In *NSDI 2008* (San Francisco, CA, USA), pp. 337–350.

[8] FALOUTSOS, M., FALOUTSOS, P., AND FALOUTSOS, C. On power-law relationships of the internet topology. In *SIGCOMM 1999* (Cambridge, MA, United States), pp. 251–262.

[9] KRIOUKOV, A., MOHAN, P., ALSPAUGH, S., KEYS, L., CULLER, D., AND KATZ, R. Napsac: Design and implementation of a power-proportional web cluster. In *ACM SIGCOMM Workshop on Green Networking* (New Delhi, India), pp. 15–22.

[10] MEMCACHED. A distributed memory object caching system. http://www.danga.com/memcached.

[11] MOSBERGER, D., AND JIN, T. httperf—A Tool for Measuring Web Server Performance. *ACM Sigmetrics: Performance Evaluation Review 26*, 3 (1998), 31–37.

[12] OLSON, M. A., BOSTIC, K., AND SELTZER, M. Berkeley db. In *USENIX ATC 1999* (Monterey, CA, USA).

[13] SHARMA, N., BARKER, S., IRWIN, D., AND SHENOY, P. Blink: managing server clusters on intermittent power. In *ASPLOS 2011* (Newport Beach, CA, USA), pp. 185–198.

[14] THERESKA, E., DONNELLY, A., AND NARAYANAN, D. Sierra: practical power-proportionality for data center storage. In *EuroSys 2011* (Salzburg, Austria), pp. 169–182.