Realizing an Elastic Memcached via Cached Data Migration

Ubaid Ullah Hafeez, Deepthi Male, Sharath Kumar Naeni, Muhammad Wajahat, Anshul Gandhi
Department of Computer Science, Stony Brook University
{uhafeez,dmale,snaeni,mwajahat,anshul}@cs.stonybrook.edu

CCS Concepts • **Computing methodologies** → *Distributed computing methodologies*; • **Computer systems organization** → *Cloud computing*; • **Software and its engineering** → *Cloud computing*;

ACM Reference format:
Ubaid Ullah Hafeez, Deepthi Male, Sharath Kumar Naeni, Muhammad Wajahat, Anshul Gandhi. 2017. Realizing an Elastic Memcached via Cached Data Migration. In *Proceedings of Middleware Posters and Demos '17: Proceedings of the Posters and Demos Session of the 18th International Middleware Conference, Las Vegas, NV, USA, December 11–15, 2017 (Middleware Posters and Demos '17),* 2 pages.
DOI: 10.1145/3155016.3155023

1 Introduction

Cloud computing has enabled economical resources, such as storage and VMs, that allow applications to be hosted online at low cost. The pay-as-you-go model in the cloud also enables elasticity – the ability to quickly add and remove VMs from an application in response to changes in workload demand. However, not all applications are amenable to elasticity, e.g., stateful services such as memory caches. Designing such services to be elastic requires careful consideration of the data saved on each VM since the data will no longer be available if the VM is terminated when scaling in.

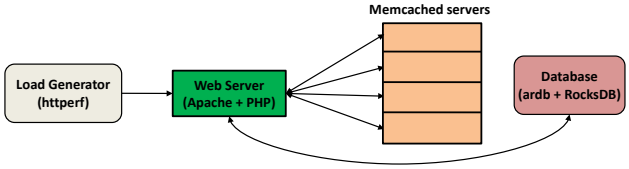

Figure 1. Our multi-tier Memcached-backed application.

We consider a multi-tier web deployment consisting of a Memcached tier comprised of several VMs, as shown in Figure 1. Memcached [3] is a distributed in-memory caching system. In Figure 1, a typical data request at a web server is first tried at the Memcached; if the data is in the Memcached (hit), then it is served from the faster DRAM of the Memcached nodes. If the data is not in the Memcached (miss), then the web server fetches the data from the slower disk-based back-end database.

To save on rental costs, the application owner may choose to scale in the Memcached tier by terminating some Memcached VMs in response to low load, resulting in increased load on the database; this increased load further adds to the request response time (RT). The blue solid line (baseline) in Figure 2 shows the steep increase in 95%ile RT when Memcached is scaled in from 4 VMs to 2 VMs (see Section 2 for details on our experimental setup). The *peak RT*, as shown in the figure, can hurt performance SLAs and the *restoration time* (time to revert to stable RTs), also shown in the figure, dictates the duration of performance degradation.

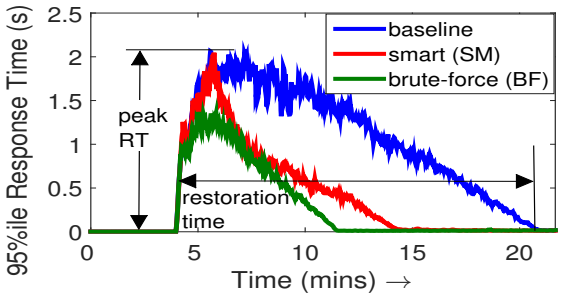

Figure 2. Performance loss following Memcached scaling.

Our goal in this work is to analyze the potential reduction in peak RT and restoration time by proactive cached data migration. We also investigate the impact of size of the Memcached tier on peak RT and restoration time.

2 Experimental Setup

Our experimental setup consists of a custom Memcached-backed web application composed of several VMs (Ubuntu 14.04.3 OS) deployed on a Private OpenStack cloud, as illustrated in Figure 1. We use httperf [5] to generate load for our application in the form of PHP web requests; the requests are directed to an Apache web server. Each request, once parsed, queries 100 random key-value (KV) pairs, whose popularity distribution can be controlled. Our Memcached (version 1.4.31) deployment consists of 4 VMs, each with 2-vCPUs and 4GB memory, mimicking a cost-efficient configuration. The database in our case is ardb [1] (version 0.9.3) that leverages RocksDB [6] as the backend.

In terms of the KV data set, the key size is fixed at 11 bytes and the value sizes range from 1 byte to 19 bytes such that all KVs belong to the same slab. The popularity of value sizes is distributed geometrically such that smaller KV pairs are more popular, as observed by Facebook [2]. The data set contains 19 Million KV pairs.

We define *response time* (RT) for each web request to be the weighted average (over the 100 KV fetches) of the latencies of the get requests that hit in the Memcached and the remaining requests that are served by the database. We report tail RTs (95%ile RTs) when evaluating performance.

3 Evaluation Results

To evaluate the potential benefits of cached data migration, we focus on scenarios where the application scaled in from 4 to 2 Memcached VMs; we refer to the 2 VMs that will be

Middleware Posters and Demos '17, Las Vegas, NV, USA
 978-1-4503-5201-7/17/12...$15.00
DOI: 10.1145/3155016.3155023

terminated as *retiring* VMs and the remaining two as *retained* VMs. Our high-level idea is to migrate hot KV pairs from retiring VMs to retained VMs so we can minimize post-scaling cache misses. We investigate different migration schemes and evaluate their performance benefits and overhead.

3.1 Migration Schemes

We present two different migration schemes:

Brute Force (BF) works by comparing the hotness, based on the timestamp of the last access, of keys on the retiring VMs with keys on the retained VMs; consistent hashing is used to determine the target retained VM for each key on the retiring VM. Based on this comparison, colder KVs on retained VMs are replaced with hotter KVs from retiring VMs.

Smart Migration (SM) approximates BF by only moving a subset of the hottest KV pairs from retiring VMs to retained VMs. Rather than comparing all KV pairs, SM only compares every 100^{th} pair (in LRU order) to estimate, for each retained VM, the subset of KV pairs from retiring VMs that are hotter than its existing KV pairs. We then determine the size of the smallest subset, say N, and move the hottest N KV pairs from among all retiring VMs to the retained VMs via multi-set; this ensures that no KV pair on retained VMs is replaced by a colder one from retiring VMs. SM also reduces the hashing overhead by computing the hash for only N KV pairs instead of all KV pairs on retiring VMs, as in BF.

3.2 Comparison of Schemes

Figure 2 shows a typical Memcached scaling experiment where we initially have very low RTs; at about the 4-min mark, we decide to scale in from 4 to 2 Memcached VMs. The baseline (in blue) represents naive scaling, resulting in poor performance for almost 16 minutes (restoration time). By comparison, we see that both BF (in green) and SM (in red) substantially improve performance in terms of restoration time, reducing it to about 7 and 10 minutes, respectively. Note that we do not immediately scale in the retiring VMs for BF and SM, instead allowing for migration of hot KV pairs before scaling. For BF, the migration of hot KV pairs takes a significant amount of time (7 mins); only after this time do we terminate the retiring VMs, thus losing out on significant cost savings. However, after migration, performance is almost optimal, thus resulting in the low RTs seen in Figure 2. For SM, the migration takes only about 3 minutes, after which the retiring VMs are terminated, thus allowing for moderate cost savings. However, since SM is not as aggressive as BF in terms of hot data migration, the performance after migration still takes some time to converge to the BF levels. In summary, while both BF and SM provide substantially lower RTs compared to the baseline, they do have some migration overhead which lowers the cost savings potential of elasticity.

3.3 Sensitivity Analysis for Smart Migration

We now further evaluate the benefits of SM over the baseline as a function of the Memcached size. Figures 3 and 4 show the restoration time and peak RT, respectively, as a function of relative Memcached size (relative to total database size) for baseline and SM. We see that SM significantly reduces both restoration time and peak RT. As the Memcached size decreases, the absolute hit rate decreases, resulting in higher load on the database. Thus, the restoration time and (to some extent) peak RT increase with a decrease in Memcached size.

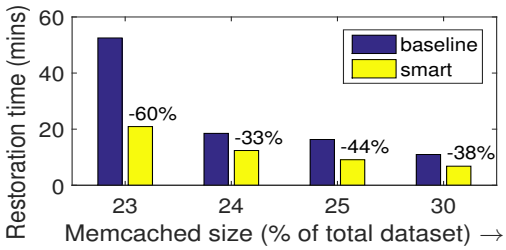

Figure 3. Restoration time versus Memcached size.

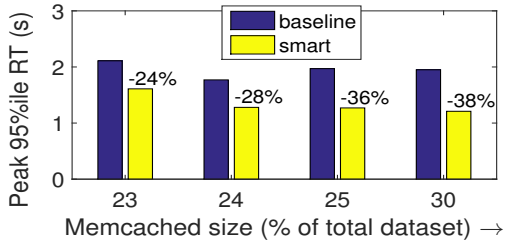

Figure 4. Peak 95%ile RT versus Memcached size.

4 Related Work

Hwang et al. [4] propose an adaptive partitioning algorithm that adds or removes Memcached nodes and migrates data to balance the load created by hot items. The authors suggest that the data migration be performed in the background, but do not discuss this further. CacheScale [7] proposes horizontal scaling of Memcached tiers by passively migrating data between Memcached nodes based on incoming requests. While effective, the restoration time of CacheScale critically depends on the arrival rate and popularity distribution, and can thus be arbitrarily high. By contrast, our data migration is independent of the arrival rate and popularity skew and can be tuned to regulate the network overhead of migration.

5 Conclusion and Future Work

This work focuses on the transient yet severe performance loss that immediately follows a Memcached scaling action. Our preliminary results highlight the potential reduction in peak RT and restoration time that can be realized by proactively migrating hot Memcached data. As next steps, we will (i) investigate techniques to minimize the overhead of our smart migration policy, and (ii) evaluate the impact of system parameters, such as database configuration, request popularity skew, and dataset size, on reduction in peak RT and restoration time.

References

[1] "ardb," https://github.com/yinqiwen/ardb.

[2] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny, "Workload Analysis of a Large-scale Key-value Store," in *SIGMETRICS*, London, England, UK, 2012, pp. 53–64.

[3] B. Fitzpatrick, "Distributed Caching with Memcached," *Linux Journal*, vol. 2004, no. 124, pp. 5–5, Aug. 2004.

[4] J. Hwang and T. Wood, "Adaptive performance-aware distributed memory caching." in *ICAC*, vol. 13, 2013, pp. 33–43.

[5] D. Mosberger and T. Jin, "httperf—A Tool for Measuring Web Server Performance," *ACM Sigmetrics: Performance Evaluation Review*, vol. 26, no. 3, pp. 31–37, 1998.

[6] "RocksDB | A persistent key-value store," http://rocksdb.org/.

[7] T. Zhu, A. Gandhi, M. Harchol-Balter, and M. A. Kozuch, "Saving cash by using less cache." in *HotCloud*, 2012.