ElCached: Elastic Multi-Level Key-Value Cache

Rahman Lavaee
University of Rochester

Stephen Choi, Yang-Suk Kee
Samsung Memory Solutions Lab

Chen Ding
University of Rochester

Abstract

Today's cloud service providers (CSPs) use in-memory caching engines to improve application performance and server revenue. However, these caching engines exhibit poor scaling, mainly because of high DRAM cost and energy consumption. On the other hand, the increasing use of multi-tenancy requires effective and optimal resource provisioning.

In this paper, we introduce ElCached, a multi-level key-value cache based on Memcached. ElCached employs low-cost NAND flash memory as a lower layer of caching. ElCached uses the reuse distance model to predict miss ratio, with high accuracy, under all storage capacity limits. The miss ratio prediction allows ElCached to find the best resource allocation under multi-tenant settings. We evaluate Elcached on workloads emulating real-world applications. Our multi-tenant experiment indicates that compared to a proportional allocation technique, ElCached can reduce the cost by up to 26%, while delivering lower average latency. Meanwhile, by utilizing more flash storage, ElCached can reduce the total memory consumption almost by half.

1 Introduction

Distributed memory caching systems such as Memcached [7] and Redis [15] have become vital components of many today's web servers. The strong locality in real world data [2] allows these systems to improve user latency and total cost of ownership of the backend databases. Considering the large gap between memory and database latencies, achieving higher cache hit rates is critical for overall cost and performance. However, increasing DRAM capacity is challenging because of high DRAM cost and power consumption. Thus the system providers need to maximize DRAM efficiency and utilization by deploying enough memory to meet the service level agreement (SLA) while minimizing the total cost of ownership (TCO). This becomes more important as more cloud users (tenants) are added to the system. The service provider must wisely allocate the resource so as to guarantee each tenant's SLA. Moreover, dynamic workload behaviors motivate **elasticity**, a capability to adapt to workload changes by dynamic resource provisioning.

Recent studies have introduced various approaches to utilize low-cost solid-state drives (SSD) to bridge the gap between memory and HDDs [5, 11, 16, 9, 3]. While these techniques properly serve their purpose, they add another level of complexity to resource provisioning. A tenant with a large working set may benefit from caching in SSD, whereas one with smaller working set may be able to achieve its desirable performance by mainly relying on DRAM.

In this paper, we propose ElCached, an elastic key-value cache. ElCached builds on MlCached [3], a multi-level caching system. ElCached exploits MlCached's decoupled caching layers to perform independent resource provisioning for both layers. ElCached implements a new reuse-distance profiler to predict miss ratio for all capacity limits. The profiling information allows the system to find the best partition for both resources, under multi-tenant settings, with various latency and cost constraints.

The rest of the paper is organized as follows. First we give an overview of the base MlCached system and its important decoupling property. Next in section 3, we explain the reuse-distance model and how we adapt it for Memcached applications. Then in section 4 we explain how ElCached formulates a resource provisioning problem, along with the miss ratio information, as a linear programming problem. Finally, in section 5, we evaluate ElCached under single and multi-tenant settings with workloads emulating real-world applications.

2 MlCached

MlCached consists of a multi-level cache hierarchy. The L1 cache is a common DRAM-based Memcached and the L2 cache is an exclusive NAND-flash-based key-value cache.

The key contribution of MlCached is removing the redundant address mappings. Utilizing commodity SSDs as another key-value caching layer is inefficient because of their block interface. In particular, most flash translation layers (FTLs) are designed to translate logical block addresses (LBA) to physical block addresses (PBA). As a result, it is necessary to store key-to-LBA mapping tables in memory. These tables hamper the effective use of memory for key-value pairs. MlCached removes the need for storing redundant key-to-LBA mapping tables by implementing direct key-to-PBA mappings on SSD.

MlCached's new key-value device not only improves efficiency, but also decouples the two caching layers by making the address mappings in the two layer completely

independent of each other. This allows for independent resource provisioning within each layer. Resource provisioning becomes more important and challenging when workloads dynamically change over time. Our new Elastic Key-Value Cache (ElCached) uses the reuse-distance theory to profile the requests and predict the performance under all storage capacity limits. Based on the profiling information, latency, and cost constraints, it allocates resources to deliver the best utilization and performance.

We model the performance of MlCached in terms of end-to-end latencies of MlCached's components: Memcached, exclusive SSD cache, and the backend DB. Let us represent these latencies, respectively, by ℓ_m, ℓ_s, and ℓ_{db}. Additionally, let M_m and M_s be the miss ratio of Memcached and SSD, respectively. The total average latency of MlCached is equal to

$$Lat = \ell_m + \ell_s \times M_m + \ell_{db} \times M_s.$$

The key to optimal resource provisioning is to find the miss ratio curve. That is a function *mr* that gives the miss ratio for all capacities. Given this curve, for Memcached of size c_m and SSD of size c_s, the total average latency is equal to

$$Lat(c_m, c_s) = \ell_m + \ell_s \times mr(c_m) + \ell_{db} \times mr(c_m + c_s).$$

Additionally, The total cost for the system is equal to

$$Cost(c_m, c_s) = p_m c_m + p_s c_s,$$

where p_m and p_s are respectively the price per unit of DRAM and SSD.

3 Reuse Distance

We use the reuse distance theory to compute the miss ratio curve. The reuse distance model is described as follows. Consider a memory access trace. Let x be an arbitrary element in the trace, representing an access to memory location a. Let y be the previous access to a. The reuse distance for x is defined as the number of distinct memory locations accessed between y and x (including a). If x is the first access to a, its reuse distance is ∞ by definition.

For example, Trace 3.1 shows a short sample trace, along with reuse distance for every access. In this trace, four accesses occur between the two accesses to b, which include accesses to only three distinct memory locations (b, c, and d). Thus the reuse distance for the second access to b is 3, as shown.

Trace 3.1

a	b	c	d	b	a	c
∞	∞	∞	∞	*3*	*4*	*4*

The reuse distance information is best represented by the reuse distance histogram, which shows the frequency for every reuse distance. We can compute the miss-rate curve (MRC) from the reuse distance histogram. The miss rate for every cache size is equal to the total number of reuse distances that are larger than that cache size. That is, the fraction of the histogram which lies on the right side of the cache size.

3.1 Reuse Distance for Memcached

Traditionally, the reuse distance model has been used to analyze workloads for LRU-based virtual memory systems and caching systems. Memcached workloads pose several new challenges for this reuse distance model.

First, Memcached distributes items among different slab classes, according to their sizes. The default slab allocation scheme allocates memory during the cold start. Once the Memcached system reaches its memory limit, replacement is done separately for every slab class. Thus, rather than analyzing the whole Memcached system in a single reuse distance model, we model each slab class separately.

Second, Memcached applications have some control over the storage content: loading is fully controlled by the application. For a GET request that misses in Memcached, the Memcached server does not automatically load the item. Rather, the application must issue a SET request to load the item in the Memcached server. Furthermore, apart from the server-level evictions due to the LRU replacement policy, Memcached applications are allowed to issue DELETE requests to remove arbitrary items from the server.

To cope with these challenges, we adapt the reuse distance model and implement it on the Memcached server side. we represent every Memcached request by the three attributes: type of request (GET, SET, DELETE), key hash, and the slab class id for SET requests.

The SET requests may modify the slab class for a key, and therefore, need an additional slab class attribute. Since the model is implemented on the server side, obtaining the slab class for SET requests is straightforward and can be done before searching for the item in Memcached.

We use the Olken algorithm [13] to compute the reuse distance histogram. Our data structure consists of a red-black tree for each slab class. The tree for a slab class includes a node for each key that belongs to that class. Conceptually, the nodes in a tree are ordered with respect to the last (logical) access times of their keys. The Olken tree augments each node by the weight of its subtree (total number of nodes in its subtree). This attribute helps us compute the reuse distance for every incoming access

by following the path along the parent pointers.

Technically, tree nodes are always inserted at the right end of the tree. Therefore, storing the access times is unnecessary. In fact, we only store the weight at tree nodes.

Different Memcached requests are handled in somewhat different ways. For a GET request, we first compute its reuse distance. Subsequently, we update the tree by removing the old node and inserting a new node. Upon a SET request, we remove the corresponding node from its current tree and insert a node into its new tree (based on its new slab class). Finally, for a DELETE request, we only remove the node from its current tree. Each operation takes $O(\lg N)$ where N is the total number of keys.

For reuse distance analysis, the storage overhead is inevitable. It is linear in the total number of distinct keys. However, since we only store hash values, the overhead may only be a fraction of the total key-value storage.

To lower the running time overhead, we run the profiler in a producer-consumer framework. Memcached's server threads insert requests into a lock-free queue, while a separate thread sequentially applies the analysis on the items. Overlapping the analysis with Memcached's service times and inter-arrival times significantly reduces its overhead.

To find the miss-rate curve, we profile the workload under adequate resource provisioning. Memcached allocates slabs on demand, during the initial warmup phase. We assume that under any other provisioning, the slab allocation is proportional to the profiled setting. For instance, suppose that under the profiling scheme, Memcached allocates twice as many slabs for slab 1 as it does for slab 2. The assumption asserts that the same happens under any other provisioning. This is a logical assumption for workloads that can be modeled by a distribution. Moreover, it allows us to distribute the total resource among all slab classes, under any resource provisioning. Once we have the miss rates for all slabs classes, we can compose them to compute the total miss rate for the workload.

4 Resource Provisioning as a Linear Program

In a single-tenant system, the resource provisioning problem can be described in one of the two ways.

- Minimize $Cost(c_m, c_s)$ such that $Lat(c_m, c_s) \leq SLA$.
- Minimize $Lat(c_m, c_s)$ such that $Cost(c_m, c_s) \leq TCO$.

To formulate these problems as linear programs, we must represent cost and latency as linear functions of c_m and c_s. Based on our model, cost is a linear function of c_m and c_s, but latency is only linear in terms of the miss ratio, which is a non-linear function of capacity. However, we observe that the miss ratio curves in our workloads are always convex. That is, the reduction in miss ratio constantly lowers as we increase the capacity. We explain how we can use this assumption to formulate the miss ratio curve as linear constraints. To illustrate, consider the miss ratio curve shown in Figure 1. We approximate this curve using three representative points (A, B, and C). Each two consecutive points specify a line, along with one flat line for each of the two endpoints. For each line, we limit the feasibility region to above the line. The intersection specifies a region above the miss ratio curve (shaded area in Figure 1). Since we are always interested in minimizing cost and latency (alternatively, capacity and miss ratio), the linear program is guaranteed to output a point on the boundary of this region, which approximates the miss ratio curve.

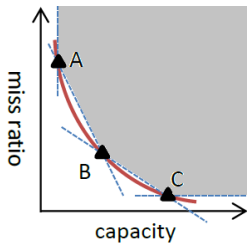


Figure 1: Formulating MRC (red curve) as linear constraints: only the points on the blue dashed lines result in optimal solutions.

A multi-tenant system introduces more constraints. For example, bounding the total available storage for each level (partitioning), or average latency. Nevertheless, our linear program can be solved efficiently and in polynomial time.

5 Experimental Evaluation

In this section we report our off-line study for elastic resource provisioning. We compare costs of two different approaches: elastic and proportional. Figure 1 shows our latency/cost parameters.

cost ($/GB)		latency	
DRAM	10	Memcached	100μs
SSD	0.68	KVD	200μs
		Back-end DB	10ms

Table 1: Cost and latency parameters

The proportional approach fixes the ratio between DRAM and SSD capacities as many current CSPs[1] do.

[1] System configurations of ($CPU \propto Memory \propto Storage$) or ($CPU \propto$

On the other hand, the elastic approach, the mechanism ElCached uses, changes the ratio adaptively based on our reuse-distance based miss-rate prediction to achieve better elasticity. That is, different data locality determines the optimal DRAM/SSD allocations to optimize either latency or cost. We first investigate the accuracy of our miss-rate prediction and then, show the increased elasticity of Web caching service with ElCached.

For these experiments, we use Mutilate [1] to generate our workloads. Mutilate emulates the ETC workloads at Facebook using key size, value size, and inter-arrival time distributions given by Atikoglu et al. [2]. The mean key size and value sizes are respectively 31 and 247 bytes. We use Mutilate to generate two workloads. For the first workload, we use a Zipfian key distribution with $\alpha = 1.15$. For the second workload, we use an exponential distribution with $\lambda = 10^{-6}$. We set the both workload to issue 800 million requests to a range of 4 billion keys. Note that this is a limit study to evaluate elasticity of ElCached compared to existing work. Therefore, it does not include the overheads and other tuning aspects of dynamic resource provisioning; data promotion, demotion, eviction overheads and profiling window, reconfiguration frequencies, and its associated efficiencies in time domain.

Prediction Accuracy We profile the workloads using our model and predict miss rates on a logarithmic scale, and compare the results to measurement. Figure 2 shows the result for the Zipfian workload. The mean relative error of prediction is 4%.

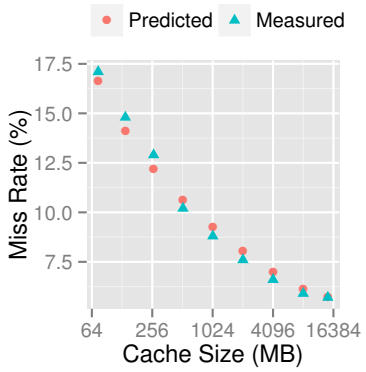


Figure 2: Miss ratio prediction accuracy of our reuse distance model

Elastic Resource Provisioning Although both are dynamically allocating/deallocating resources, the proportional approach does not change the ratio between DRAM and SSD. In particular, we use 1:4 ratio following the Pareto principle. On the other hand, the elastic approach continuously changes the ratio based on the reuse-distance based miss-rate prediction. Figure 3 shows minimum cost per latency for both techniques, and for both workloads. For example, to guarantee the average latency of 700μs for the Zipfian workload, elastic costs 0.7 and proportional costs 8.1 (cost is normalized with respect to the cost of proportional at 1ms). In this case, elastic allocates 60MB/11.7GB while proportional does 1.8GB/7.3GB for DRAM and SSD, respectively. Assuming that CSPs bill tenants proportional to actual cost, elasticity will save costs both for users and CSPs. In particular, for the Zipfian workload, elasticity saves around 60% of cost. Meanwhile, the Exponential workload exhibits around 10% reduction in cost until 250μs latency requirement, but this rapidly changes to more than 70% cost improvement after 300μs. This is because the Exponential workload has much stronger data locality than Zipfian. So DRAM allocation dominates the performance before 300μs.

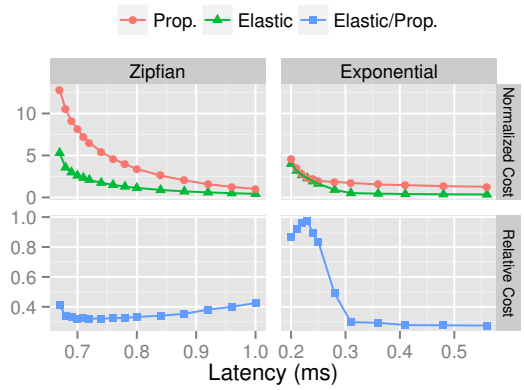
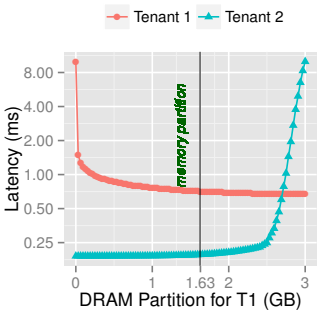


Figure 3: Elastic vs. proportional resource provisioning

We also conduct a comparison between elastic and proportional in a multi-tenant case. Here we assume a virtual cloud instance with 3 GB memory. Figure 4a shows latency for the two tenants when partitioning the fixed amount of memory. In particular, proportional partitions memory and allocates each memory and SSD with the fixed ratio of 1:4. On the other hand, elastic predicts an optimal configuration per tenant and tests its feasibility. We compare the elastic approach to one particular memory partition which allocates 1.63 GBs of DRAM for tenant 1 (shown by a line in Figure 4a). As shown in Figure 4b, both approaches result in the same latency for tenant 2, whereas, for tenant 1, elastic improves latency by 5%. Moreover, elastic reduces the cost for both tenants by around 26%. Elastic also saves the total memory consumption by 46% by allocating only 1.63 GB

Memory + Remote Storage) are common from Amazon EC2, Google Cloud Platform, and Microsoft Azure. Remote storage is out of the scope of this work.

compared to the 3 GB of physical memory. That is the memory consumption of tenant 1 alone, when taking the proportional approach. This means potential revenue for CSP because it could potentially host more tenants as long as other resources (CPU, network, etc.) are available.



(a)

Proportional

	T1	T2
DRAM	1.63	1.37
Lat.	0.71	0.20
Cost	7.20	5.41

Elastic

	T1	T2
DRAM	0.46	1.17
Lat.	0.67	0.20
Cost	5.30	4.01

(b)

Figure 4: Multi-tenant resource provisioning: (a) latency per DRAM partitioning with proportional SSD size (1:4 ratio) (b) elastic vs proportional with the specified memory partition

6 Online Resource Provisioning

Our study of online resource provisioning is preliminary and remains as future work. Here, we explain the basics of our online implementation.

Memcached allocates memory in units of fixed size (called slabs) until it reaches its memory limit. Thus we can control resource provisioning by dynamically reconfiguring the memory limit parameter. Increasing this parameter allows Memcached to boost memory provision to the new limit. To lower the provision, we must find and deallocate victim slabs. Prior to deallocation, all the items in the victim slabs must be removed from the LRU chains. We choose victim slabs from different size classes to ensure that the number of slabs in different size classes remains proportional to the initial setting. To find a victim slab in each size class, we monitor the activity of all the slabs during a short period of time. Slab activity is defined as the number of distinct items accessed during that period. For each slab, we estimate its activity using a Hyperloglog counter [8] with logarithmic space overhead. After the monitoring period, we choose the least active slabs as the victim slabs. We keep deallocating slabs until the new memory limit is satisfied.

Our online analysis consists of a profiling window followed by a release period. During the profiling period, we use the reuse distance analysis to find the miss rate for every cache size. At the end of the profiling window, we formulate and solve a linear program according to the constraints and the objective function. The solution gives the new Memcached and KVD sizes. Then, we reconfigure the ElCached system using our above-mentioned procedure.

The study of online resource provisioning remains as future work. An important question is how the profiling window length affects the accuracy and optimality of solution. Our future work also includes analyzing the memory and time overheads of the online reuse distance analysis.

7 Related Work

Our study is related to previous work on optimizing Web caching applications. For example, Dynacache [4] optimizes Memcached by profiling workloads and dynamically adjusting memory resources and eviction policies. Saemundsson et al. [14] introduced MIMIR, a lightweight online profiler for Memcached. Analogous to us, they used a tree-based scheme to estimate reuse distance. To lower the overhead, MIMIR divides the LRU stack into variable-sized buckets. However, contrary to us, MIMIR and Dynacache both assume a fixed size for all cached elements, which isn't the case in Memcached. Hu et al. [10] introduced LAMA to improve slab allocation in Memcached. Like us, they also estimate the miss ratio for every slab class.

Reuse distance computation has been well-studied in the past. Mattson et al. [12] introduced a basic stack simulation algorithm. Olken [13] gave the first tree-based method using an AVL tree. Ding and Zhong [6] used a similar technique to approximate reuse distance while reducing the time overhead. Xiang et al. [18] defined the footprint of a given trace to be the number of distinct elements accessed in the window. They showed that under a certain condition (reuse-window condition) reuse distance is equal to the finite differential of average footprint. We implemented the footprint approach and found its accuracy to be slightly lower than our tree-based algorithm. These techniques all require a linear space overhead. Wires et al. [17] used counter stacks to approximate the miss ratio curve with sublinear space overhead. These studies can be integrated into ElCached to reduce the space and time overhead in the online analysis.

8 Conclusion

In this paper, we described ElCached, a multi-level key-value caching system that uses a reuse distance profiler to estimate the miss rate curve across all capacity limits with 4% relative error. ElCached solves the resource pro-

visioning problem by formulating it as a linear program. Our single-tenant experiments show that ElCached can reduce the total cost by up to around 60% compared to a proportional resource provisioning scheme. On the other hand, our 2-tenant experiment indicates that by finding the optimal resource provisioning, we can improve latency, cost, and total DRAM usage, compared to a proportional scheme.

References

[1] Mutilate. https://github.com/leverich/mutilate, 2014. [Online].

[2] ATIKOGLU, B., XU, Y., FRACHTENBERG, E., JIANG, S., AND PALECZNY, M. Workload analysis of a large-scale key-value store. In *ACM SIGMETRICS Performance Evaluation Review* (2012), vol. 40, ACM, pp. 53–64.

[3] CHOI, I. S., AHN, B. Y., AND KEE, Y.-S. Mlcached: Multi-level dram-nand key-value cache. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)* (2016).

[4] CIDON, A., EISENMAN, A., ALIZADEH, M., AND KATTI, S. Dynacache: Dynamic cloud caching. In *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)* (2015).

[5] DEBNATH, B., SENGUPTA, S., AND LI, J. Flashstore: high throughput persistent key-value store. *Proceedings of the VLDB Endowment 3*, 1-2 (2010), 1414–1425.

[6] DING, C., AND ZHONG, Y. Predicting whole-program locality through reuse distance analysis. In *ACM SIGPLAN Notices* (2003), vol. 38, ACM, pp. 245–257.

[7] FITZPATRICK, B. Distributed caching with memcached. *Linux journal 2004*, 124 (2004), 5.

[8] FLAJOLET, P., FUSY, É., GANDOUET, O., AND MEUNIER, F. Hyperloglog: the analysis of a near-optimal cardinality estimation algorithm. *DMTCS Proceedings*, 1 (2008).

[9] GARTRELL, A., SRINIVASAN, M., ALGER, B., AND SUNDARARAJAN, K. Mcdipper: A key-value cache for flash storage, 2013.

[10] HU, X., WANG, X., LI, Y., ZHOU, L., LUO, Y., DING, C., JIANG, S., AND WANG, Z. Lama: Optimized locality-aware memory allocation for key-value cache. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (2015), pp. 57–69.

[11] LIM, H., FAN, B., ANDERSEN, D. G., AND KAMINSKY, M. Silt: A memory-efficient, high-performance key-value store. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 1–13.

[12] MATTSON, R. Evaluation of multilevel memories. *IEEE Transactions on Magnetics 7*, 4 (1971), 814–819.

[13] OLKEN, F. Efficient methods for calculating the success function of fixed-space replacement policies. Tech. rep., Lawrence Berkeley Lab., CA (USA), 1981.

[14] SAEMUNDSSON, T., BJORNSSON, H., CHOCKLER, G., AND VIGFUSSON, Y. Dynamic performance profiling of cloud caches. In *Proceedings of the ACM Symposium on Cloud Computing* (2014), ACM, pp. 1–14.

[15] SANFILIPPO, S., AND NOORDHUIS, P. Redis, 2009.

[16] SRINIVASAN, V., AND BULKOWSKI, B. Citrusleaf: A real-time nosql db which preserves acid. *Proceedings of the VLDB Endowment 4*, 12 (2011).

[17] WIRES, J., INGRAM, S., DRUDI, Z., HARVEY, N. J., AND WARFIELD, A. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)* (2014), pp. 335–349.

[18] XIANG, X., BAO, B., DING, C., AND GAO, Y. Linear-time modeling of program working set in shared cache. In *Parallel Architectures and Compilation Techniques (PACT), 2011 International Conference on* (2011), IEEE, pp. 350–360.