Dynamic Memory Partitioning for Cloud Caches with Heterogeneous Backends

Cristina L. Abad, Andres G. Abad and Luis E. Lucio
Escuela Superior Politecnica del Litoral, ESPOL. Guayaquil, Ecuador.
{cabadr,agabad,lelucio}@espol.edu.ec

ABSTRACT

Software caches, implemented with in-memory key-value stores, are important components of cloud architectures. In a common scenario, one server may serve requests from several applications with different workloads, each supported by a different backend (database or storage system); these applications compete for an allocation of the total memory. We present a model for dynamic memory partitioning for cloud caches with heterogeneous backends. Our work differs from recent work for cloud caches in that we consider: (1) the effect of having backends with different performance profiles, and (2) the cost of re-partitioning the memory. We discuss implementation issues that must be addressed, including the need for on-line and lightweight mechanisms for estimating the miss rate curves (MRCs) and ways to solve the non-convex optimization problem; specifically, we propose a probabilistic adaptive search algorithm that can be used for discontinuous, non-differentiable, or non-convex MRCs.

1. INTRODUCTION

Providers use caches to reduce latency and serve content faster, which in turn leads to increased sales and user engagement. For example, a 100ms latency penalty can lead to a 1% sales loss for Amazon, and an additional 400ms delay in search responses can reduce search volume by 0.74% [6]. For this reason, software caches—implemented with in-memory key-value stores like Redis and Memcached—have become important components of cloud architectures.

Software caches are simple, built to serve requests at a high-throughput, with minimum latency. Beyond their eviction policies, like Least Recently Used (LRU), they do not have any additional intelligence, making them unable to adapt to workload changes or application demands [3].

In this paper, we consider the case of a single organization that has multiple application workloads to cache. For example, consider a dynamic website that has four workloads to cache: SQL results, dynamic HTML pages, user profiles and user avatars. Each of these workloads can be served by a different Redis instance—in a multi-instance architecture—or by a single Memcached instance—in a multi-tenant architecture. Figure 1 illustrates these architectures. In this scenario, a challenge is how to efficiently share the limited memory resource between the applications. This is currently done statically [3], relying on empirical human expertise and offline workload analysis.

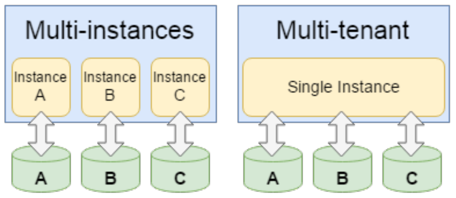

Figure 1: Multi-instance vs. multi-tenant cloud cache deployments; Redis favors the architecture on the left while Memcached, the one on the right.

We propose a model that can be used to dynamically configure memory allocations based on the observed workloads so that the overall system utility is maximized. Our work differs from recent work in memory partitioning for cloud caches [2, 3, 7, 5] in that we consider: (1) the effect of having backends with different performance profiles, and (2) the cost of re-partitioning the memory between applications.

Unlike recent work that has used the cache hit rate as the utility function to maximize [2, 3, 7, 5], we seek to minimize the overall expected access time of objects in the system; this includes the time to access the cache on a hit, and the time to access the backend on a miss. In addition, and given that different backends may be able to support different throughput, we include restrictions for the maximum throughput supported by each backend.

The cache re-partitioning cost, measured in increased latency while the memory is being reassigned, has not been considered by recent work either. The reason for this is that these solutions [2, 3, 7, 5] have been tailored for the multi-tenant architecture, encouraged by Memcached, for which the re-partitioning cost is negligible compared to the cost of re-assigning memory between different software instances[1].

We model these requirements and constraints as an optimization problem, whose utility function could be non-convex (as has been observed in real deployments [3]). We propose a probabilistic adaptive search algorithm that can be used for discontinuous, non-differentiable, or non-convex

ICPE'17, April 22-26, 2017, L'Aquila, Italy
© 2017 ACM. ISBN 978-1-4503-4404-3/17/04...$15.00
DOI: http://dx.doi.org/10.1145/3030207.3030237

[1]We ran experiments and confirmed our hypothesis that reassigning memory between instances incurred in a non-negligible cost; our results are reported in Section 3.

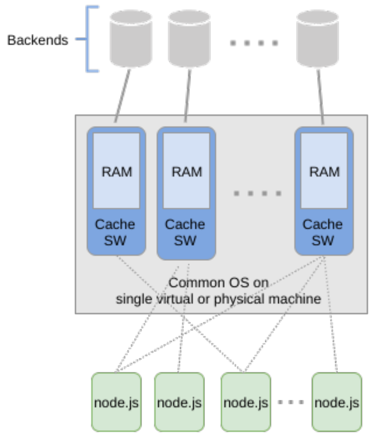

Figure 2: Motivating architecture: multi-instance caching layer with heterogeneous backends. Our model also works for a multi-tenant architecture.

utility functions. We are implementing the solver as a service that can be used by many clients to periodically calculate their optimal partitioning. The miss rate curves (MRCs)—the variable component of our utility functions—will be profiled by our service to determine the best way to solve the problem (LP-solver, hill climbing or our algorithm).

Finally, we discuss the implementation issues that should be addressed to implement our model, including the need for on-line mechanisms for estimating the MRCs, how to do so without adding latency to cache requests, among others.

2. THE CACHE PARTITIONING PROBLEM

We consider a system where a single virtual or physical machine hosts n instances of a software cache like Redis, each serving a different application workload (see Figure 2). These applications compete for the allocation of the total memory M. Our model also works for a multi-tenant architecture, as long as the different applications sharing the cache belong to the same organization (as discussed later in this section). For a solution that considers competing applications possibly trying to game the system, see [5].

In case of a cache miss, each application i is served by a backend with a maximum request throughput T_i. For example, a database server used to construct user profiles may have a lower throughput than a server that generates user identity avatars based on client IP (like Github's identicon).

Our model considers memory as the only shared resource, ignoring the sharing of CPU. In-memory key-value stores are memory- and not CPU-bound. This has been reported by Redis, and observed in caching-as-a-service providers as well as in large Memcached deployments [3].

The goal is *Pareto efficiency*: Fully utilize the memory, compute the ideal memory allocation $\mathbf{m} = [m_1, \ldots, m_n]$ and achieve the highest overall utility, given individual utility functions U_i's and total memory constraint M. This can be expressed as optimization problem:

$$\begin{aligned} \underset{\mathbf{m}}{\text{maximize}} \quad & \mathcal{F}(\mathbf{m}) = \sum_{i=1}^{n} w_i U_i(m_i) \\ \text{subject to} \quad & \sum_{i=1}^{n} m_i \leq M, \\ & m_i \geq \underline{m}_i, i = 1, ..., n, \end{aligned} \tag{1}$$

where $U_i(m_i)$ is the utility function of application i as a function of its assigned memory m_i, and w_i is the weight assigned to i; this lets us indicate that one application is more (or less) important. If all the weights are equal, then all applications are equally important. $\underline{m}_i$ is the minimum memory assignment for application i.

We assume a non-adversarial model in which the applications are not trying to game the system. This is a reasonable assumption when all the applications belong to the same cloud client. Given that we consider a non-adversarial model, we do not seek *strategy proofness* [5]. Some application could be able to issue workloads that lead to a higher memory assignment to said application, but this would be at the cost of reduced overall system performance.

2.1 Backend-agnostic utility function

If we ignore the differences between the performance profiles of the backends, then the utility function can be expressed as the product of the application hit rate and the frequency of requests issued during some period:

$$U_i(m_i) = f_i \times h_i(m_i), \tag{2}$$

where $h_i(m_i)$ is the hit rate of application i as a function of assignment m_i, and f_i is the frequency of requests of i.

Recent work on on-line and lightweight miss rate curve (MRC) estimation [10, 9, 4] can be used to estimate the hit rate curve of each application, h_i, at a low cost. In Section 3 we discuss the overhead associated with these methods.

2.2 Backend performance

A limitation of the utility function described in (2) is that it optimizes the hit rate, regardless of whether an application may require a higher hit rate or tolerate a lower one. In practice, improving the hit rate of one application may not be as equally useful as improving the hit rate of another application, due to differences in the performance profiles of the corresponding backends. For example, all other things being equal, if one of the applications has a slow backend (e.g., if it processes complex SQL queries), increasing the hit rate of the cache serving that application is more useful than increasing the hit rate of a cache in front of a faster backend (e.g., a NoSQL database storing user profiles).

Instead of the hit rate, we use the *effective access time (EAT)* of an application as its utility function. The EAT_i is the time that it takes, on average, to access an object in application i. If we define cd_i as the time to access an object in the caching system and bd_i as the time to access an object in the backend system (including the time to process a cache miss), then the utility function to maximize is:

$$\begin{aligned} & U_i(m_i) = -f_i \times EAT_i(m_i), \text{ where} \\ & EAT_i(m_i) = h_i(m_i) \times cd_i + [1 - h_i(m_i)] \times bd_i. \end{aligned} \tag{3}$$

2.3 Solving the optimization problem

Consider optimization problem (1)[2]. If functions U_i's are all quasi-linear (as assumed in [2]) the resulting optimization problem can be solved by solving a sequence of feasibility problems, with a guaranteed precision of ϵ after $\lceil \log_2 R/\epsilon \rceil$ iterations, where R is the length of the search interval. If

[2]Our discussion henceforth applies to any of the utility functions in this paper.

functions U_i's are all concave, we have a convex optimization problem easily solved with off-the-shelf solvers. When functions U_i's are: discontinuous, non-differentiable, or non-convex, alternative approaches are required due to a lack of gradient or the presence of local optima.

One alternative is to use probabilistic search approaches, in which a generative model is used to generate candidate points in the search of an optimum. An adaptive mechanism may be added to the generative model to improve the performance of the sequentially generated candidates.

Let $\mathbf{x} = [x_1, \ldots, x_n]$ with $x_i = (m_i - \underline{m}_i)/(M - \sum_i \underline{m}_i)$, satisfying $\sum_i x_i = 1$ and $0 < x_i < 1$. We assume that the variability of $\mathbf{x}$ can be well modeled by a Dirichlet[3] distribution $Dir(\mathbf{x}|\alpha)$, with parameter vector $\alpha = [\alpha_1, \ldots, \alpha_n]$. Then, we define:

$$\tilde{\mathcal{F}}(\mathbf{x}) = \mathcal{F}\left((M - \sum_i \underline{m}_i)\mathbf{x} + \underline{\mathbf{m}}_i\right), \qquad (5)$$

where $\underline{\mathbf{m}}_i = [\underline{m}_1, \ldots, \underline{m}_n]$.

We propose the following general approach, inspired in evolutionary strategies, for solving problem (1) in the case of non-convex and non-quasi-linear functions U_i's. We begin by setting α_i to $1/n$, for all i. We then generate K points $\mathbf{x}_k^\star|\alpha \sim Dir(\mathbf{x}|\alpha)$ for $k = 1, \ldots, K$. Note that points generated in this way satisfy all restrictions of problem (1).

Using points in set $\{\mathbf{x}_k^\star\}_{k=1}^K$ we construct the following prior mixture density for α:

$$g(\alpha; \{\mathbf{x}_k^\star\}) = \frac{1}{Z} \sum_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \phi_{\mathbf{x}}(\alpha), \qquad (6)$$

where Z is a normalization constant and $\phi_{\mathbf{x}}$ is a non-negative function with finite mass concentrated around $\mathbf{x}$ (e.g., a *radial basis function* centered at $\mathbf{x}$). We proceed by sampling α from g and generating points $\mathbf{x}|\alpha_\gamma \sim Dir(\mathbf{x}|\alpha_\gamma)$, where α_γ is the vector with elements $\gamma\alpha_i$ for $\gamma > 1$. Note that, while random variables $\mathbf{x}|\alpha$ and $\mathbf{x}|\alpha_\gamma$ have the same expected value, γ has the effect of reducing the variance of $\mathbf{x}|\alpha_\gamma$ by a factor of γ^{-1}.

The above generative procedure corresponds to the following bayesian hierarchical structure:

$$\begin{aligned} \alpha &\sim G(\alpha; \{\mathbf{x}_k^\star\}) \text{ and} \\ \mathbf{x}|\alpha_\gamma &\sim Dir(\mathbf{x}|\alpha_\gamma), \end{aligned} \qquad (7)$$

where G is the distribution function corresponding to mixture density g.

Our method proceeds by alternatively generating parameters α—the exploration stage—and generating J points $\mathbf{x}$'s conditioned on α_γ—the exploitation stage. The prior distribution for α is then updated using the K best cumulatively-observed points $\mathbf{x}_k^\star$'s and the procedure is repeated until a satisfactory solution is found.

Algorithm 1 provides the details of the proposed procedure. It rests to define how to choose $\gamma^{(j)}$ within the inner loop. The idea is to increase its value at each iteration to reduce the variance of α_γ and guarantee its convergence.

[3]The Dirichlet distribution $Dir(\mathbf{x}|\alpha)$ has a density function:

$$f(\mathbf{x}|\alpha) = \frac{\Gamma(\sum_{i=1}^n \alpha_i)}{\prod_{i=1}^n \Gamma(\alpha_i)} \prod_{i=1}^n x_i^{\alpha_i - 1}, \qquad (4)$$

expected value $\mathbb{E}[x_i] = \alpha_i/A$ and variance $\mathbb{V}[x_i] = \alpha_i(A - \alpha_i)/(A^2(A+1))$, where $A = \sum_i \alpha_i$.

Algorithm 1: Probabilistic adaptive search

```
Input: Functions $U_i$'s; number $K$ of points to use;
       number $J$ of rounds; function $\phi_{\mathbf{x}}$
Output: Best point $\mathbf{x}^*$, such that $\tilde{\mathcal{F}}(\mathbf{x}^*) \geq \mathbf{x}$ for every
        point $\mathbf{x}$ generated
1  $\alpha_i := 1/n$ for $i = 1 \ldots, n$
2  Generate $\mathbf{x}_k^\star|\alpha \sim Dir(\mathbf{x}|\alpha)$ for $k = 1, \ldots, K$
3  repeat
4      Generate $\alpha \sim G(\alpha; \{\mathbf{x}_k^\star\})$
5      for $j = 1, \ldots, J$ do
6          Generate $\mathbf{x}|\alpha_{\gamma^{(j)}} \sim Dir(\mathbf{x}|\alpha_{\gamma^{(j)}})$
7          if $\tilde{\mathcal{F}}(\mathbf{x}) > \min_k \{\tilde{\mathcal{F}}(\mathbf{x}_k^\star)\}$ then
8              $\{\mathbf{x}_k^\star\} := \mathbf{x} \cup \{\mathbf{x}_k^\star\} \setminus \arg\min_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \tilde{\mathcal{F}}(\mathbf{x})$
9  until Satisfactory solution $\mathbf{x}^\star = \arg\max_{\mathbf{x} \in \{\mathbf{x}_k^\star\}} \tilde{\mathcal{F}}(\mathbf{x})$ is
   found;
10 return $\mathbf{x}^\star$
```

2.4 Backend constraints

One may also want to add a backend's maximum throughput, T_i, to the problem formulation; increasing the hit rate is a way to ensure that T_i is not exceeded. This can be expressed as a non-linear constraint:

$$f_i \times [1 - h_i(m_i)] \leq T_i, i = 1, ..., n. \qquad (8)$$

However, adding this constraint complicates the probabilistic search: We can reject solutions that do not satisfy the constraint requirements, but this would increase the convergence time. An alternative approach is to incorporate this constraint in the EAT_i formula in (3): When a backend's constraint is exceeded, requests get queued up and the time to access an object in the backend system, bd_i, increases.

2.5 Dynamic partitioning

As the optimal allocation may change over time, the optimal assignment should be recalculated periodically. We must consider the cost of re-partitioning the memory: While the cache instance adapts to the new partition size (e.g., by evicting keys if it is to use less memory than before), it may incur in a penalty of a reduced throughput.

Let $\mathbf{m}^{(t)}$ be a feasible solution to problem (1) at time t. The cost of going from a solution $\mathbf{m}^{(t-1)}$ to solution $\mathbf{m}^{(t)}$ will be represented by non-negative function $C(\mathbf{m}^{(t-1)}, \mathbf{m}^{(t)})$. We do not assume C to be symmetric in its arguments since allocating and deallocating memory may have different operational costs (our preliminary experiments confirm this to be the case; results in Section 3). Cost function C is incorporated to the objective function of (1) at time t as:

$$\mathcal{F}^{(t)}(\mathbf{m}) = \sum_{i=1}^{n^{(t)}} w_i^{(t)} U_i^{(t)}(m_i) - C(\mathbf{m}^{\star,(t-1)}, \mathbf{m}), \qquad (9)$$

where $\mathbf{m}^{\star,(t-1)}$ is the chosen solution to (1) at time $t-1$.

We propose the use of a cost function of the form:

$$C(\mathbf{m}^{(t-1)}, \mathbf{m}^{(t)}) = \sum_{i=1}^n \beta_i(m_i^{(t-1)} - m_i^{(t)}), \qquad (10)$$

where:
$$\beta_i = \begin{cases} \beta_i^+ & \text{if } m_i^{(t-1)} \leq m_i^{(t)}; \\ -\beta_i^- & \text{else.} \end{cases} \tag{11}$$

Thus, we assume a cost (proportional to the memory size) of β_i^+ for allocating and β_i^- for deallocating.

Objective function (9) can be directly handled by the adaptive search approach described above. For solvers that cannot handle the proposed cost function directly, we cast the following equivalent optimization problem:

$$\begin{aligned} \underset{\mathbf{m},\eta^+,\eta^-}{\text{maximize}} \quad & \mathcal{F}^{(t)}(\mathbf{m},\eta^+,\eta^-) = \sum_{i=1}^{n} w_i U_i^{(t)}(m_i) \\ & \qquad - \sum_{i=1}^{n} \beta_i^{+,(t)} \eta_i^+ - \sum_{i=1}^{n} \beta_i^{-,(t)} \eta_i^- \\ \text{subject to} \quad & m_i^{\star,(t-1)} - m_i \leq \eta_i^+, \text{for all } i \\ & - m_i^{\star,(t-1)} + m_i \leq \eta_i^-, \text{for all } i \\ & \sum_{i=1}^{n} m_i \leq M, \\ & m_i - \underline{m}_i^{(t)}, \eta_i^+, \eta_i^- \geq 0, \text{for all } i, \end{aligned} \tag{12}$$

which preserves convexity and quasi-linearity from (1).

3. IMPLEMENTATION ISSUES

We are working on using our model to implement a dynamic memory partitioning tool for Redis. In the process, we need to solve the following problems.

Workload monitoring and hit rate curve estimation: The goal is to implement a very lightweight monitoring mechanism to construct hit rate curves with very little space and time overhead. For our Redis monitoring module we plan on adapting SHARDS [9], a novel algorithm that can construct approximate miss rate curves by sampling a small percentage of requests and has a small memory footprint. For example, it can process as much as 17 million requests per second and build curves with $< 2.6\%$ error, with a constant memory footprint (authors report excellent accuracy in 1 MB footprint for very large workloads [9]). To avoid slowing down cache requests, we are implementing the workload profiler as an asynchronous module with a lock-free buffer using the Disruptor pattern, originally developed for a high-performance financial exchange [8].

Cost of re-sizing the memory: We ran simple tests in which we deallocated 350 MB from one Redis instance and reassigned this RAM to another instance. We report the results averaged across 3 experimental runs: In our tests, deallocating memory from an instance took 28.6 seconds, while adding RAM took 2.4 seconds. During this period, the Redis instance stopped serving requests. Our model considers this cost, which may vary from system to system. Our plan is to implement a profiler that measures the re-sizing cost and can be used to automatically configure our tool.

Solving the optimization problem: Miss rate curves (MRCs) can have performance cliffs [3] which complicate the task of solving the optimization problem. Optimal allocation is NP-complete with non-convex MRCs [1] so no efficient solution is known. In the past, others have tackled the problem as follows: (1) For concave or near concave MRCs, Dynacache [2] applies convex piece-wise fitting and solves the problem with an LP-solver; (2) Cliffhanger [3] modified Memcached's LRU so that up to one performance cliff is removed using shadow queues, an approach inspired by Talus [1]; it then uses a gradient-based algorithm (hill-climbing) to optimize the allocation; since the full MRC curve is never built, the allocation is iteratively optimized, an approach that makes sense when reallocating memory is cheap, but inadequate for the problem being solved in this paper; finally, (3) others [7, 5] have simply proposed solutions that only work for concave MRCs; however, real measurements have revealed that this assumption may not hold [3]. We plan to implement the solver as a cloud service that could be used by many cloud clients to periodically calculate their optimal memory partitioning. The miss rate curves (MRCs) will be profiled by our service to determine the best way to solve the problem at hand: LP-solver, gradient-based method (with or without convex hull elimination) or probabilistic adaptive search. We also plan to explore the option of iterating between methods to speed up convergence.

4. CONCLUSIONS AND FUTURE WORK

We studied the problem of dynamic memory partitioning of cloud caches and modeled it as an optimization problem. Our formulation considers the effect of having backends with different performance profiles, as well as the cost of re-partitioning the memory. We proposed a probabilistic search algorithm that can be used when the utility functions are non-convex (which may be the case in our problem).

We are currently working on implementing our proposed approach as follows: (1) A miss rate curve profiling tool for Redis, leveraging recent high-performance solutions like SHARDS and LMAX's Disruptor pattern; (2) a profiling tool for Redis that can learn the cost of re-partitioning the memory between Redis instances; and (3) a partitioner-as-a-service tool that will implement several ways to solve the optimization problem and profile the utility function to choose the most appropriate one depending on the actual workload.

Acknowledgements. This work was partly funded by a Google Faculty Research Award.

5. REFERENCES

[1] Beckmann and Sanchez. Talus: A simple way to remove cliffs in cache performance. In *HPCA*, 2015.

[2] Cidon, Eisenman, Alizadeh, and Katti. Dynacache: Dynamic cloud caching. In *HotCloud*, 2015.

[3] Cidon, Eisenman, et al. Cliffhanger: Scaling performance cliffs in web memory caches. In *NSDI*, 2016.

[4] Hu, Wang, Zhou, Luo, et al. Kinetic modeling of data eviction in cache. In *Usenix ATC*, 2016.

[5] Pu, Li, Zaharia, Ghodsi, and Stoica. Fairride: Near-optimal, fair cache sharing. In *NSDI*, 2016.

[6] Singla, Chandrasekaran, Godfrey, and Maggs. The Internet at the speed of light. In *HotNets*, 2014.

[7] Stefanovici, Thereska, O'Shea, Schroeder, et al. Software-defined caching: Managing caches in multi-tenant data centers. In *SOCC*, 2015.

[8] Thompson et al. Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads. *LMAX Tech. Rep.*, 2011.

[9] Waldspurger, Park, Garthwaite, and Ahmad. Efficient MRC construction with SHARDS. In *Usenix FAST*, 2015.

[10] Wires, Ingram, Drudi, et al. Characterizing storage workloads with counter stacks. In *OSDI*, 2014.