Cost-based Memory Partitioning and Management in Memcached

Damiano Carra
Computer Science Dept.
University of Verona
Verona, Italy
damiano.carra@univr.it

Pietro Michiardi
Networking and Security Dept.
Eurecom
Sophia Antipolis, France
pietro.michiardi@eurecom.fr

ABSTRACT

In this work we present a cost-based memory partitioning and management mechanism for Memcached, an in-memory key-value store used as Web cache, that is able to dynamically adapt to user requests and manage the memory according to both object sizes and costs. We then present a comparative analysis of the vanilla memory management scheme of Memcached and our approach, using real traces from a major content delivery network operator. Our results indicate that our scheme achieves near-optimal performance, striking a good balance between the performance perceived by end-users and the pressure imposed on back-end servers.

CCS Concepts

•General and reference → Measurement; Experimentation; •Information systems → *Data layout;*

Keywords

In-memory storage; Web cache

1. INTRODUCTION

Modern Web sites and applications attract very large numbers of end-users, which expect services to be responsive at all times: indeed, *latency* plays a crucial role in the perceived Quality of Experience (QoE), which determines to a large extent the popularity and success of competing services.

Today's web pages have a complex structure, as they are composed by tens of objects, often served by a pool of back-end servers. In addition, objects usually do not have the same relevance: central panels, side panels or advertisements may have different values for end-users as well as content providers. To serve Web pages composed by such heterogeneous objects efficiently, modern web architectures make use of fast, in-memory storage systems that work as web caches.

In this context, Memcached [3] is a widely-used caching layer: it is a key-value store that exposes a simple API to store and serve data from the RAM. Thanks to its simplicity and efficiency, Memcached has been adopted by many successful services and companies, such as Wikipedia, Flickr, Digg, WordPress, Craigslist, and, with additional customizations, Facebook and Twitter.

In-memory cache systems keep replicas of the contents stored permanently in the back-end databases. Such objects not only have different sizes – from few bytes corresponding to the text of a web page, to tens or hundreds of kilobytes for pictures, up to few megabytes for multimedia content – but they might have different retrieving costs. In the literature, there are a number of mechanisms [26, 10, 7] which consider object cost to be related to the complexity of the database query to generate an object, which may not be correlated with the size of the object itself. In such a scenario, the traditional *hit ratio* (number of hits divided by the number of requests) may be not sufficient to capture the "pressure" on the back-end. A metric based on the objects cost, such as the *cost hit ratio* (sum of the cost of the hits divided by the sum of the cost of the requests), is thus truly desirable.

Many works, such as Cao *et. al.* in [10], have proposed simple and elegant solutions that take into account the cost when managing objects in a cache. Nevertheless, these solutions can not be directly applied to Memcached: for efficiency reasons, Memcached has a specific memory management scheme. By design, Memcached partitions the memory dedicated to store data objects into different classes; each class is dedicated to objects with a progressively increasing size. When a new object has to be stored, Memcached checks if there is available space in the appropriate class, and stores it. If there is no space, Memcached *evicts* a previously stored object in that class in favor of the new one.

In this paper we implement a cost-based memory management scheme for class-based, in-memory storage systems such as Memcached. A cost-based solution introduces a number of challenges that are not immediately clear when approaching the problem. How can the memory be divided among different classes in an *on-line fashion*, *i.e.,* while the system is running? What happens if the cost associated to objects in a class changes over time? How often should the system re-evaluate the decisions made?

Our contributions: We design and implement a new API for Memcached, in which it is possible to associate the cost of an object to their requests. The API is an extension of the `Set` operation, where, along with the key and the value, a numeric entry corresponding to the cost can be added. Along with the API, we have implemented an on-line scheme that takes into account the cost of the objects stored in

IMDM '15, August 31 2015, Kohala Coast, HI, USA

DOI: http://dx.doi.org/10.1145/2803140.2803146

the different classes to decide how to partition the memory among them. The basic idea used in our scheme is to balance the number of misses, weighted by the cost of the objects, among different classes.

To validate our memory management scheme, we use an experimental approach, and conduct a series of experiments on a testbed which is representative of the typical blueprint of modern web architectures. In our experimental campaign, we use input traces (*i.e.,* events corresponding to storing or fetching objects) that are both real – collected from a vantage point inside a major content delivery network (CDN) – and synthetic, based on statistics from traces in the literature.

We compare our mechanism with an *optimal allocation* that we compute off-line, with a variation of the Mattson stack algorithm [21]. Our results indicate that the memory allocation in Memcached is far from being optimal, even when object costs are not taken into account. With our scheme, we obtain superior hit ratios both when objects have all the same cost and when they have different costs. In summary, our scheme achieves near-optimal performance, striking a good balance between the performance perceived by end-users and the pressure imposed on back-end servers.

The remainder of the paper is organized as follows. In Section 2 we provide some background information on Memcached and we discuss the related works. We present our solution in Section 3, along with a method to compute, off-line, the optimal allocation. Our results are shown in Section 4 and we provide additional observations and discussions in Section 5. We conclude in Section 6.

2. BACKGROUND AND RELATED WORK

2.1 Memcached

Memcached is a key-value store that keeps data in memory, *i.e.,* data is not persistent. Clients communicate with Memcached through a simple set of APIs: `Set, Add, Replace` to store data, `Get` or `Remove` to retrieve or remove data. Memcached has been designed to simplify memory management [29] and to be extremely fast: since every operation requires memory locking[1], data structures must be simple and their access time should be kept as small as possible.

In Memcached, the basic unit of memory is called a *slab* and has fixed size, set by default to 1 MB. A slab is logically sliced into chunks that contain data items (objects[2]) to store. The size of a chunk in a slab, and consequently the number of chunks, depends on the class to which the slab is assigned. A class is defined by the size of the chunks it manages. Sizes are chosen with a geometric progression: for instance, Twitter uses common ratio 1.25, and scale factor 76, therefore the sizes of the chunks in class 1, 2, 3, ..., are 76, 96, 120, ... Bytes respectively. An object is stored in the class that has chunks with a size sufficiently large to contain it. As an illustration, using the classes defined at Twitter, objects with sizes 60 Bytes, 70 Bytes, and 75 Bytes are all assigned to class 1, while objects with sizes 80 Bytes and 90 Bytes are assigned to class 2.

[1]Note that memory locking is necessary even in case of a `Get`, since access time statistics need to be updated for the eviction policy to work properly.

[2]Throughout the paper we will use the terms "object" and "item" interchangeably.

The total available memory to Memcached is allocated to classes in a slab-by-slab way. The assignment process follows the object request pattern: when a new request for a particular object arrives, Memcached determines the class that can store it, checks if there is a slab assigned to this class, and if the slab has free chunks. If there is no free chunk (and there is available memory), Memcached assigns a new slab to the class, it slices the slab into chunks (the size of which is given by the class the slab belongs to), and it uses the first free chunk to store the item. When all slabs have been assigned to the classes, Memcached adopts the Least Recently Used (LRU) policy for eviction. Note that LRU is applied on a per-class basis: items in other classes are stored in chunks of memory with different sizes, and chunks can not be moved.

Once an appropriate portion of memory has been assigned to a class, it is *permanently* associated to such class (unless the Memcached server is restarted). Recently, it has been shown that the current memory management policy induces *slab calcification* [1, 6], which may have a negative impact on the system performance.

Even if there have been many attempts to solve slab calcification – examples are the Memcached Automove policy and the Twitter Twemcache policies [5] – it is still not clear if the slab assignment process itself is optimal. Moreover, none of the above polices takes into account the different costs that can be associated to objects.

2.2 Related Work

The analysis of cache performance has been the subject of many past studies. In this paper we consider specifically Memcached, therefore we first focus on the literature about such system. Even if Memcached is widely used, the study of its performance has received only little attention. Atikoglu *et. al.* provide in [6] a set of measurement results from a production site – in our experiments we use these statistics to generate our "synthetic" workload. However, the work does not analyze eviction policies, and it does not consider the impact of memory partitioning on the hit ratio.

Gunther *et. al.* [18] highlight that Memcached has scalability issues, since threads access the same memory, and locks prevent the exploitation of parallelism. For this reason, a number of works [29, 17] consider the throughput of Memcached, proposing a set of mechanisms and data structures to decrease the overall latency. These works do not consider explicitly the impact of the memory partitioning on the hit ratio as we do. Nishtala *et. al.* [23] study scalability problems, *i.e.,* how to manage a multi-server architecture, but they do not study the eviction policies and memory partitioning.

Overall, the literature on caching mechanisms is vast: CPU [9], browser [26], Web [10], and DNS caches [19], as well as Content Delivery Networks [25] and P2P networks [8, 11, 15, 12, 14, 22] are each characterized by different problems. Among previous works, CPU caches need to solve similar problems to ours. In a CPU cache, many processes share the same memory space, and a single process may "pollute" the cache with its data [27], which has a negative impact on performance. Similarly, in Memcached, different classes share the memory, and the space taken by a class may hurt the performance of other classes and therefore the overall hit ratio. The solution adopted for CPU caches [28, 27, 24] are based on a common idea, in which the memory partition-

ing process tries to balance the number of misses among the processes. In [13] the authors propose a scheme for dynamically allocating the memory to different classes of objects: in all these cases the solutions do not consider the impact of cost in their decisions.

In Web caches, there are a number of examples [26, 10, 7] which consider object cost to be related to the complexity of the database query to generate the object, and not their size. Solutions that are able to handle this situations are presented by Cao *et. al.* in [10]. Nevertheless, such approach can not be directly adapted to the specific memory partitioning mechanism adopted by Memcached, since, for performance reasons, objects are divided into classes, and eviction is done on a per-class basis.

3. COST-BASED MEMORY MANAGEMENT

In this section we present our approach to a cost-based memory allocation and management scheme for Memcached. In addition – to obtain a baseline for a comparative analysis – we discuss an off-line algorithm that computes an optimal memory allocation.

3.1 Miss-Ratio Curve

The analysis of cache performance has been the subject of many studies in the last 30 years. Analytical models are usually based on the Independent Reference Model (IRM), in which objects, and their access pattern, are modeled by independent random variables. Unfortunately this model has some limitation, as it fails to capture the performance of the storage system when, for instance, request arrivals are correlated, or objects have different sizes. For these reasons, storage systems are usually studied with trace-driven numerical analysis: given a trace and an eviction policy, it is possible to compute the *miss-ratio curve*, *i.e.,* the miss ratio that is obtained for different sizes of the cache.

The calculation of the miss ratio curve can be done with a single pass of the traces using the Mattson stack algorithm [21]. A typical output of the analysis of a trace is shown in Figure 1. Even if the figure shows a specific trace, its concave shape is representative of most common cases, which exhibit diminishing returns: the gain that can be obtained with the first few blocks of allocated memory is usually much higher than the one that can be obtained by additional allocations. For instance, it is clear that, if the available memory is sufficient for storing all of the content, adding more memory will not further decrease the miss ratio of a cache.

The Mattson stack algorithm assumes that all the objects have the same cost. In order to compute the miss-ratio curve in the general case where objects have different costs, we propose a variant: at each requests, we update the stack distance with the corresponding cost, instead of simply incrementing the stack by one. In this way, the vector that keeps track of the hits for different values of memory sizes, takes correctly into account the sum of the costs of each object stored in that position.

The considerations made so far focus on a single-class cache, where all objects have the same size. What happens when we have different classes? How can the memory be divided among different classes, so that each class achieves the best possible miss-ratio? In its on-line version, a solution to this problem is the main contribution of our work. Here, for the sake of building a baseline comparison, we consider the off-line version, *i.e.,* we first execute our variant of the

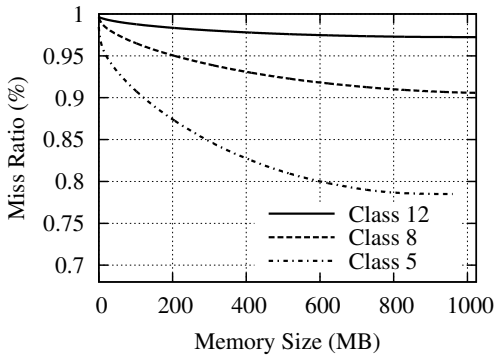

Figure 1: Examples of the miss ratio curve for different classes. The miss ratio is computed considering in the denominator the total number of requests for all classes (this is why it is so high, on a per-class basis). The composition of all classes yields an overall miss ratio which goes, as the memory increases, below 20%.

Mattson stack algorithm for all the classes, then we optimally assign portions of the memory to the different classes.

Next, we discuss some necessary assumptions to make the off-line problem tractable. We assume that memory can be divided into a finite number of blocks (in Memcached, they are the *slabs*), and that each class receives and integer number of blocks. We assume also that the relative decrease of the miss-ratio, as the memory grows, is monotonically decreasing: in practice, given a memory size i and the miss ratio m_i for that memory size, then $(m_i - m_{i+1}) \geq (m_{i+1} - m_{i+2})$, $\forall i$. Both assumptions are reasonable, and the second has been confirmed by analyzing the real-life traces we use in this work, as Figure 2 shows.

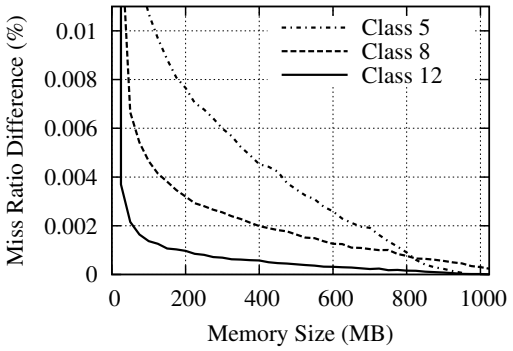

Figure 2: Example of the miss ratio difference curve for different classes. The miss ratio difference is the decrease in the miss ratio when slabs are added to the class, one at a time.

Given the above assumptions, a simple heuristic can be used to compute the optimal assignment (see Algorithm 1). At each iteration, classes are sorted by the miss-ratio difference $m_{j,k} - m_{j,k+1}$, where j is the class index and k is the number of blocks assigned to class j so far: a memory block is assigned to the class with the highest miss-ratio difference. This procedure (sorting and block assignment) is repeated

for all the available blocks.

Algorithm 1 Optimal Off-line Slab Assignment

1. **Input:** Miss ratio curves for all classes
2. **Input:** Number of available slabs
3.
4. **repeat**
5. Sort classes by their miss ratio difference
6. Assign a slab to the class with the highest difference
7. **until** All slabs are assigned

The output of the procedure represents the optimal assignment for a given memory size. This output can be used as a reference, since it can be computed only off-line, after a trace has been analyzed. The aim of our work is to find the on-line counterpart.

3.2 On-line Mechanism

Memcached provides a set of APIs for storing key-value pairs. To enable cost-based memory management, we extend the APIs to handle costs when storing or modifying the data. In particular, we consider the interfaces `Set(key,value)`, `Add(key,value)` and `Replace(key,value)`, and introduce the new siblings `Set(key,value,cost)`, `Add(key,value,cost)` and `Replace(key,value,cost)`. For the purpose of our memory management scheme, we not only store the cost of each object inside Memcached, but we also keep a set of counters that summarizes the cumulative cost for each class, and the counters are updated when objects are added or evicted from the storage.

Once the costs have been set, the memory management module periodically evaluates the memory assignment. The frequency of this evaluation may depend on the the number of requests or on the number of misses. Since our aim is to control the number of misses, we have experimentally observed that using the number of misses produces more stable results. The length of the period is less relevant, since it influences only the convergence speed.

At every assignment interval, we maintain a number of auxiliary counters, that we use for computing slab allocation. For each class, we maintain the sum of the costs of the requested objects (that resulted in a hit) and the sum of the cost of the misses. Memcached does not hold information about the cost for requested objects that result in a miss; therefore we consider the cost of the storage operation (`Set(key,value,cost)`), since we expect that, after a miss, the object is retrieved from the back-end and stored in Memcached. We also distinguish between the misses for objects that have never been asked before, and *misses for objects that have been evicted*. This distinction is important, since we cannot avoid the misses for objects that have never been asked before, while knowing the misses for objects that have been evicted is an indication of potential hits if we increase the memory for that class.

In order to distinguish between these two types of misses, we use a technique developed in [20], which uses two Bloom filters, b_1 and b_2. When a request to store an object arrives, the signature of the key is checked in both Bloom filters. If it is found in either b_1 or b_2, it is registered as a miss due to eviction. If it is not found, it is stored only in b_1. Periodically, b_2 is reset, the content of b_1 is copied in b_2 and b_1 is reset. This approach represent a sort of moving window Bloom filter, and it is done to avoid the saturation of the filter.

Next, assume that all auxiliary information described above – the cost of the misses due to eviction per class **m**, the cost of the requests per class **r**, and the number of slabs allocated to each class **s** – is available. Our memory management scheme uses a *slab allocation algorithm* that we label SAS. The algorithm, outlined in Algorithm 2, evaluates a single slab movement, from the class with the lower risk of increasing the number of misses to the one that has registered the largest number of misses.

For a given class i, we define its risk as the ratio between the cost of the misses due to eviction and the number of slabs allocated to the class, $m_i/(r_i s_i)$: in other words, "moving" one slab from one class to another, increases the number of misses, as a first approximation, by a value equal to $m_i/(r_i s_i)$. This is equivalent to assuming that the miss ratio difference is proportional to $1/s_i$. While more sophisticated measures can be used to estimate the variation in the number of misses when slabs are removed, our measurements have shown that the approach we propose is fairly accurate. If the class with the lowest risk has more than one slab, the slab reassignment follows a Least Recently Accessed (LRA) approach within the class, *i.e.,* we select the slab that has not been accessed for the longest time. Once slab allocation completes, LRU-based eviction within each class ensures an efficient memory utilization, until the next round.

Algorithm 2 Slab Allocation Scheme (SAS)

1. **Input:** **s** // vector of slabs allocated to each class
2. **Input:** **r** // vector of requests in each class
3. **Input:** **m** // vector of misses due to eviction in each class
4.
5. **Every** M misses **do**
6. $id_{\text{take}} \leftarrow \arg\min_i \frac{m_i}{r_i s_i}$;
7. $id_{\text{give}} \leftarrow \arg\max_i m_i$;
8. MoveOneSlab($id_{\text{take}}, id_{\text{give}}$);

Note that SAS considers the cost of the misses per class *and* per slab: SAS aims at finding a working point where a change in the memory partitioning does not increase the miss ratio. In summary, SAS can be thought of as a mechanism that caters to a high hit ratio by *adapting how memory is partitioned* to mirror object popularity dynamics (as memory allocation is continuously re-evaluated) and variations in object size distribution and cost.

4. COMPARATIVE ANALYSIS

We now present our experimental results, where we compare the performance of vanilla Memcached to that of a Memcached server that uses our memory partitioning management scheme. First, we discuss our experimental setup, then present our comparative analysis. In what follows, we focus on results obtained using real traces. We also have run experiments with synthetic traces, which we omit for clarity of the presentation, since we find similar quantitative behavior to what we obtain with real traces.

4.1 Experimental Setup

Typically, in scale-out Web applications, a series of Memcached servers are configured in a shared-nothing setup,

Table 1: Information about the traces.

Number of requests received	$9.67 \cdot 10^6$
Number of distinct objects	$5.62 \cdot 10^6$
Cumulative size of the requested objects	8.07 GiB

whereby each server takes care of a subset of data objects using consistent hashing [6] or variations thereof. This means that each Memcached server receives requests for objects that have approximately the same statistical properties. Therefore, to study memory management, it is sufficient to measure the performance of a single Memcached server. As for the request arrival to the server, Memcached locks the memory at each operation: even if requests are managed by many threads (used to maintain open connections, process the requests and prepare the responses), from the memory viewpoint, these requests are processed in series; hence, generating the requests from a single, or from multiple clients, has little or no impact on memory management.

Following the above observations, in our experiments we deploy a simple, yet representative, Web architecture: an application server is connected to a database and to a Memcached server (the cache size is set to 1 GB). A client issues requests for objects that are permanently stored in the database. The application server checks if the requested object is in the cache; if Memcached returns the object, the application server serves the client; otherwise, it retrieves the object from the database, serves the client and stores the object in Memcached.

The database is populated with objects extracted from real traces collected at a vantage point of a major CDN. The traces we use include the cost necessary to retrieve each object: for simplicity, we store such costs in an efficient data structure within the application server, so that each client request for an object is associated to its cost. Hence, the application server can use the new Memcached API we have developed and specify object costs along with requests[3]. The client issues object requests by replaying the real traces.

4.2 CDN Traces

In this section we describe the traces used in the experiments. The traces have been collected from one of the servers of a major CDN operator. As shown in Table 1, the traces contains almost 10 million requests for more than 5 million objects. Overall, the sizes of the objects sum up to approximately 8 GiB. The traces cover a period of time of roughly 48 hours of request traffic.

Next, we focus on *object popularity*. We compute the number of requests received by each object, sort objects according to this value and plot the empirical cumulative distribution function (CDF) of object popularity in Figure 3. It is interesting to note that, while the tail of the distribution follows a Zipf-like distribution, the popularity of the top 1'000 objects follows a different pattern. This, along with the heterogeneous object size, limits the applicability of theoretical models available in the literature to analyze the performance of the cache, and justifies the experimental approach we take in this work.

[3]In a real deployment, cost-information is usually provided by back-end services.

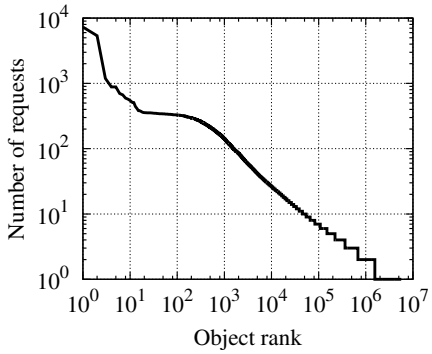

Figure 3: Number of requests for each object, ordered by objects rank, from the most popular to the least popular.

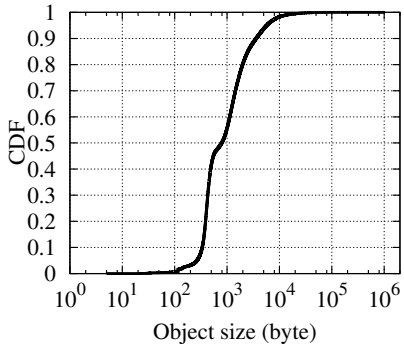

Figure 4: Empirical cumulative distribution function of the object sizes.

Next we focus on the *object size* distribution. The size of the objects spans from few bytes up to 1 MiB, with most of the objects having size between 100 bytes and 10 kiB. Figure 4 shows the empirical CDF of object sizes.

Along with each object, the traces report an additional parameter called *retrieval time*, which is the time need to fetch the object either from the original server, the cache hierarchy, the disk or memory, along with the necessary computation (e.g., unzipping or encoding the content). Considering the objects retrieved from the original server and the cache hierarchy, their retrieval times are an effective measure of the pressure on back-end servers each object impose, as computed by the CDN management system. Thus, in our experiments, we use the retrieval time as the cost associated to the object. Due to internal CDN operator confidentiality policies, this cost has been re-normalized to an integer between 1 and 10'000. Figure 5 shows the empirical CDF of the object costs.

It is important to note that the retrieval time is not necessarily correlated object sizes: Figure 6 shows the relation between the object size and its cost (each point represents an object). We have also computed the correlation coefficient between the size and the cost, obtaining a value equal to 0.013, which indicates no correlation.

The fact that objects may have different costs represents an important information that should be used when man-

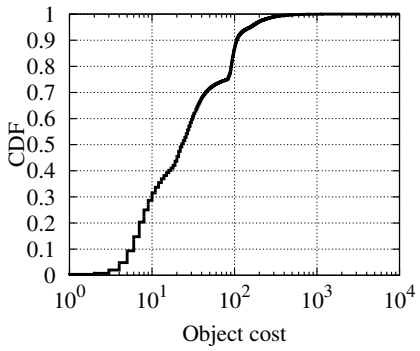

Figure 5: Empirical cumulative distribution function of the object costs.

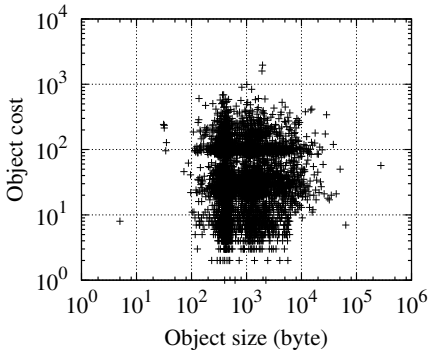

Figure 6: Relation between object size and cost. Each point represents an object.

aging the storage system. In the following, we study the performance of Memcached using a variety of cost metrics, as determined by different memory management policies.

4.3 Results

The main performance metric we consider in this work is the *cost hit ratio*, which is given by the sum of the costs of the hits divided by the sum of the costs of all the requests. This metric is computed from the application server that receives the requests from the client.

Our traces can be used to perform a variety of experiments, by changing the cost used to characterize the object. For instance, if we set all the costs equal to 1, we obtain the basic cache behavior, and the cost hit ratio becomes the traditional *hit ratio*, where each hit contributes equally. Alternatively, we can use the size of the objects as cost: in this case the performance metric indicates the so-called *byte hit ratio*. Finally, we can use the retrieval time to understand the impact of such costs on the performance.

In all the three cases outlined above, we use our variant of the Mattson stack algorithm, along with Algorithm 1 presented in Section 3, to compute – in an off-line manner – the *optimal* cost hit ratio. In the results we present, we take as a reference this optimal solution, and we show the cost hit ratio with respect to that optimum. For instance, a cost hit ratio of 90% means that the *on-line* version of a memory management scheme is able to obtain 90% of the hit ratio that the optimal off-line algorithm could obtain. Next, we present our results in terms of the cost hit ratio as a function of the number of requests received by the application server: hence, the x-axis of our figures is only loosely related to time.

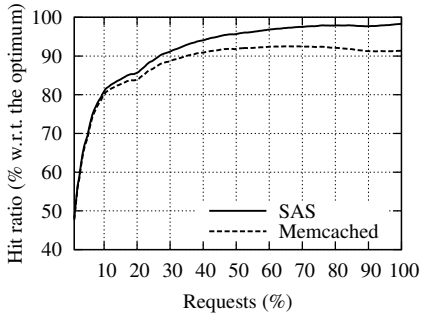

Figure 7: Hit ratio obtained when all the objects have the same cost.

Figure 7 shows the hit ratio when all the objects have the same cost. It is interesting to note that the basic Memcached policy is far from optimal: the slab assignment based on object request arrival is not able to exploit correctly the available memory. With SAS, instead, the hit ratio converges towards the optimum, as more and more requests are processed.

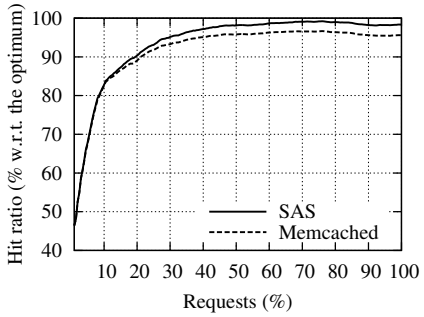

Figure 8: Byte hit ratio obtained when the objects have a cost equal to their sizes.

In Figure 8 we consider the case where the objects have cost equal to their sizes. The byte hit ratio obtained by Memcached is close to optimal: the vanilla Memcached slab assignment works well when focusing on the sizes of the objects. Nevertheless, the assignment is static, therefore any change in the statistical properties of the objects may lead to sub-optimal performance (see Section 5 for a brief discussion about a phenomenon called *calcification*). As for the SAS scheme, also in this case, its dynamic adaptation is able to slightly improve over Memcached.

In our final experiment, we assign object costs to be equal to their retrieval time, which constitutes a more representative cost value than object sizes. In this case, our scheme strives at reorganizing memory based on both user request patterns and the pressure on the back-end each request imposes. As the number of requests increases, SAS achieves near-optimal performance. Instead, the static memory management of vanilla Memcached, provides a sub-optimal cost hit ratio, which translates into eviction of objects that need

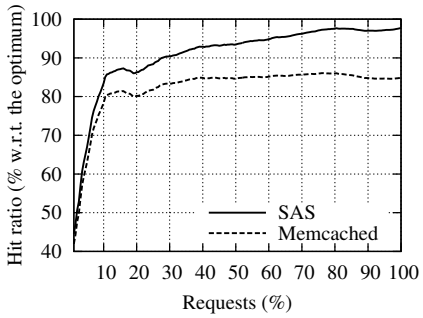

Figure 9: Cost hit ratio obtained with retrieval times as objects costs.

to be retrieved again from the back-end, with high costs.

To better illustrate the process of slab assignment made by SAS, we take a snapshot of the system approximately after half of the requests has been processed. This snapshot is shown in Figure 10, which describes how *slabs* are partitioned across object size classes. As a reference, we show the optimal slab assignment. The SAS scheme starts with an initial slab assignment similar to the one used by Memcached, since slabs are allocated on a per-request basis. As soon as all the available slabs are assigned, SAS periodically reorganizes memory allocation to decrease the cost due to the misses. The profile of the assignment gradually shifts from the one that characterizes Memcached, to the optimal one.

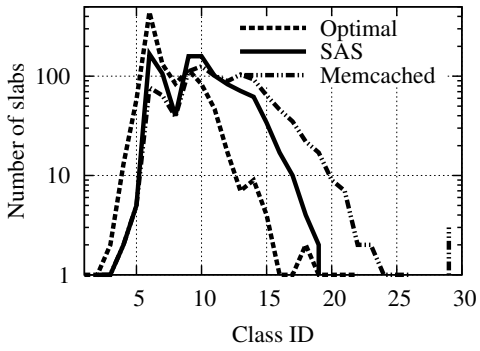

Figure 10: Slab assigned to the different classes. The snapshot has been taken after half of all the requests has been processed.

In summary, our proposed scheme is able to dynamically adapt to user requests and manage the memory according to both object sizes and costs.

5. DISCUSSION

We now discuss additional details of the SAS policy.

Parameter settings: The proposed scheme has a single parameter that needs to be set: how often the slab assignment should take place. If the interval is too short, then the variability of the statistics (sum of the cost of the requests and misses) may negatively impact the assignment. If the interval is too long, the scheme converges slowly to the optimum. We have experimentally checked that the sensitivity of the results to this parameter is actually not significant. We tested different cases with interval between 10'000 and 100'000 misses, and observed that any value within this range provide similar results.

Calcification: Another aspect that we have not considered is the variation of the statistical properties of the requested objects: what happens if the characteristics of the objects (average size, or cost) change over time, e.g., due to technological reasons? Examples are an update of the back-end architecture that translates into different costs, or a change in the resolution of the images stored on the web site that periodically increases. Memcached is not able to handle properly these situations [13], a problem known as *calcification* [1]. Our solution does not suffer from calcification since it continuously adapts the slab assignment to the current trace statistics.

Synthetic traces: The experiments we discussed in Section 4 consider a single trace. We have tested SAS on other internal traces, as well as on synthetic traces based on data reported by Atikoglu *et. al.* in [6]. In all the cases we have obtained similar results, not reported here for the sake of simplicity of the exposition.

Computation of misses due to eviction: In our scheme we distinguish between misses due to objects that have never been requested before, and misses due to objects previously evicted. This distinction is made possible thanks to the use of two Bloom filters. This technique is patented [20] and cannot be directly used in a open source software such as Memcached (note that we have used it for the sake of experiments, not in a production site). As a future work, we will investigate alternative techniques that can be used and released as open software (e.g., based on counting Bloom filters).

6. CONCLUSION AND PERSPECTIVES

Web-scale companies invest many resources and make considerable efforts to improve the performance of their web applications and the perceived Quality of Experience by end-users. Focusing on the hit ratio alone does not account for the *pressure* on the back-end correctly, since such a metric disregards the cost necessary to obtain the data. The cost hit ratio represents a metric that summarizes *both* the work done in the back-end and the performance perceived by the end users.

In this work, we proposed a cost-based memory management mechanism for Memcached – a widely adopted in-memory storage system used as a fundamental building block in many modern web architectures – that is able to dynamically provide a cost-based hit ratio close to optimal. Our scheme works on-line and is able to adapt to the characteristics of the objects that are requested, while other solutions statically allocate the memory to the different classes, thus obtaining sub-optimal performance.

As a future work, we intend to investigate alternative in-memory storage systems, such as Redis [4], that adopt a different approach for memory allocation than that of Memcached. In particular, we plan to analyze the performance of memory allocation schemes specifically designed to avoid fragmentation, such as `jemalloc` [16], developed by Jason

Evans for FreeBSD, and Google's `TCmalloc` [2]. With these allocators, the in-memory storage system does not need to take care of object classes, and can perform memory management with a single LRU queue. A comparison among such systems and Memcached may reveal interesting trade-offs and limits of modern memory management schemes.

7. ACKNOWLEDGMENTS

The authors would like to thank Marco Leovino for the fruitful initial discussion on the topic.

8. REFERENCES

[1] Caching with twemcache and calcification. https://blog.twitter.com/2012/caching-with-twemcache. Accessed: 2015-06-01.

[2] gperftools. https://code.google.com/p/gperftools/. Accessed: 2015-06-01.

[3] Memcached. http://memcached.org. Accessed: 2015-06-01.

[4] Redis. http://redis.io/. Accessed: 2015-06-01.

[5] Twemcache. https://github.com/twitter/twemcache. Accessed: 2015-06-01.

[6] B. Atikoglu, Y. Xu, E. Frachtenberg, S. Jiang, and M. Paleczny. Workload analysis of a large-scale key-value store. In *Proceedings of the 12th ACM SIGMETRICS/PERFORMANCE joint international conference on Measurement and Modeling of Computer Systems*, 2012.

[7] O. Bahat and A. Makowski. Optimal replacement policies for non-uniform cache objects with optional eviction. In *Proceedings of the Twenty-Second Annual Joint Conference of the IEEE Computer and Communications (INFOCOM)*, 2003.

[8] E. Biersack, D. Carra, R. L. Cigno, P. Rodriguez, and P. Felber. Overlay architectures for file distribution: Fundamental performance analysis for homogeneous and heterogeneous cases. *Computer Networks*, 51(3):901–917, 2007.

[9] G. Blelloch and P. Gibbons. Effectively sharing a cache among threads. In *Proceedings of the sixteenth annual ACM symposium on Parallelism in algorithms and architectures (SPAA)*, 2004.

[10] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Annual Technical Conference*, 1997.

[11] D. Carra, R. L. Cigno, and E. Biersack. Graph based modeling of p2p streaming systems. In *Proceedings of 6th International IFIP-TC6 Networking Conference on NETWORKING*, May 2007.

[12] D. Carra, R. L. Cigno, and E. Biersack. Stochastic graph processes for performance evaluation of content delivery applications in overlay networks. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):247–261, 2008.

[13] D. Carra and P. Michiardi. Memory partitioning in memcached: An experimental performance analysis. In *Proceedings of IEEE International Conference on Communications (ICC)*, June 2014.

[14] D. Carra, M. Steiner, and P. Michiardi. Adaptive load balancing in kad. In *Proceedings of International Conference on Peer-to-Peer Computing (P2P)*, Sept. 2011.

[15] R. L. Cigno, A. Russo, and D. Carra. On some fundamental properties of p2p push/pull protocols. In *Proceedings of 2nd International Conference on Communications and Electronics (HUT-ICCE)*, June 2008.

[16] J. Evans. A scalable concurrent malloc(3) implementation for freebsd. Unpublished, April 2006.

[17] B. Fan and D. Andersen. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[18] N. Gunther, S. Subramanyam, and S. Parvu. Hidden scalability gotchas in memcached and friends. In *VELOCITY Web Performance and Operations Conference*, 2010.

[19] J. Jung, E. Sit, H. Balakrishnan, and R. Morris. DNS performance and the effectiveness of caching. In *ACM SIGCOMM Workshop on Internet Measurement (IMW)*, 2001.

[20] A. Khakpour and R. J. Peters. Optimizing multi-hit caching for long tail content. US Patent n.US20130179529 A1, Jul 2013.

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

[22] P. Michiardi, D. Carra, F. Albanese, and A. Bestavros. Peer-assisted content distribution on a budget. *Computer Networks*, 55(7):2038–2048, 2012.

[23] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling memcache at facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2013.

[24] M. Qureshi and Y. Patt. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, 2006.

[25] S. Saroiu, K. P. Gummadi, R. J. Dunn, S. D. Gribble, and M. Levy. An analysis of Internet content delivery systems. *SIGOPS Operating System Review*, 36(SI):315–327, 2002.

[26] D. Starobinski and D. Tse. Probabilistic methods for web caching. *Performance Evaluation*, 46(2-3):125–137, 2001.

[27] G. Suh, L. Rudolph, and S. Devadas. Dynamic partitioning of shared cache memory. *The Journal of Supercomputing*, 28(1):7–26, April 2004.

[28] D. Thiebaut, H. Stone, and J. Wolf. Improving disk cache hit-ratios through cache partitioning. *IEEE Transactions on Computers*, 41(6):665 –676, jun 1992.

[29] A. Wiggins and J. Langston. Enhancing the Scalability of Memcached. In *Intel document, unpublished*, http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached, 2012.