Memory Partitioning and Management in Memcached

Damiano Carra, and Pietro Michiardi

Abstract—Memcached is a popular component of modern Web architectures, which allows fast response times – a fundamental performance index for measuring the Quality of Experience of end-users – for serving popular objects. In this work, we study how *memory partitioning* in Memcached works and how it affects system performance in terms of hit ratio.
We first present a cost-based memory partitioning and management mechanism for Memcached that is able to dynamically adapt to user requests and manage the memory according to both object sizes and costs. We 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. We show that our proposed memory management scheme achieves near-optimal performance, striking a good balance between the performance perceived by end-users and the pressure imposed on back-end servers.
We then consider the problem known as “calcification”: Memcached divides the memory into different classes proportionally to the percentage of requests for objects of different sizes. Once all the available memory has been allocated, reallocation is not possible or limited. Using synthetic traces, we show the negative impact of calcification on the hit ratio with Memcached, while our scheme, thanks to its adaptivity, is able to solve the calcification problem, achieving near-optimal performance.

Index Terms—Web architectures, performance evaluation.

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 [4] 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 [8], [12], [27] 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 insufficient 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 [12], 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?

The fact that the statistical characteristics of the objects, or the cost associated to them, may change, introduces another problem for Memcached: Once all the available memory has been allocated, memory reallocation is not

- *D. Carra is with University of Verona, Italy.*
- *P. Michiardi is with EURECOM, France.*

supported.[1] Such a strict approach to memory allocation raises a problem referred to as *calcification* – a problem observed in some prominent operational setups [1], [6], [7]. Despite the clear consequences on hit ratio, this problem has received little attention in the literature.

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 online scheme that takes into account the cost of the objects stored in 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.

We then study the effect of calcification on Memcached performance. Using an experimental approach, we show and determine how calcification adversely impacts the hit ratio. In our experiments, we use the latest version of Memcached and Twemcache – a custom version developed at Twitter that includes a series of policies to address the calcification problem. In addition, we generate object size distribution according to the model introduced by Atikoglu *et. al.* [7], which is based on production-traces of Memcached at Facebook. We show that our memory management scheme is able to avoid calcification, maintaining close to optimal performance.

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. We study the calcification problem in Section 5, and we provide additional observations and discussions in Section 6. We conclude in Section 7.

1. Starting from version 1.4.11, Memcached now provides a mechanism to reallocate the memory. However, the reallocation algorithm is extremely conservative, therefore reallocation is rare.

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 [31] and to be extremely fast: since every operation requires memory locking[2], 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[3]) 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.

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], [7], 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 [6] – 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. 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.

3. Throughout the paper we will use the terms "object" and "item" interchangeably.

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 [7] 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 [17], [31] 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.* [22] study scalability problems, i.e. how to manage a multi-server architecture, but they do not study the eviction policies and memory partitioning.

Recently, two works [19] [23] sharing similar objectives to ours have appeared in the literature: these studies have been developed independently and at the same time with our work. Hu *et. al.* [19] propose a dynamic programming approach for computing the best slab assignment. Their approach is computationally intensive, and the obtained results are similar to ours in terms of hit ratio. Nevertheless, they do not consider the impact of costs, and the corresponding cost hit ratio. The scheme proposed by Ou *et. al.* [23] has been tested on a key-value store simulator, while we have implemented our solution in Memcached, including the new interfaces necessary to handle the object costs.

Overall, the literature on caching mechanisms is vast: CPU [10], browser [27], Web [12], and DNS caches [20], as well as Content Delivery Networks [25], 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 [28], 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 [24], [28], [29] are based on a common idea, in which the memory partitioning process tries to balance the number of misses among the processes. In [14] 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 [8], [12], [27] 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 these situations are presented by Cao *et. al.* in [12]. 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 limitations, 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]. While more efficient algorithms have been proposed in the literature to compute the stack distance – also known as reuse distance, see [33] for an overview, or [32] [30] as examples for approximated solutions – in this section we present the basic version of the Mattson algorithm for simplicity of exposition. Any variant proposed in the recent literature can be used to improve the efficiency, but the algorithm output would be the same. For instance, the original algorithm maintains a stack where objects are stored, and an array to keep track of the position of the object in the stack in case of a hit. Both these data structures can be substituted with efficient and dynamic data structures such as a *counting B-Tree*, where elements can be inserted, removed and searched in $O(\log N)$, with N number of elements in the tree, and, with the same complexity, it is possible to obtain the current position of an element within the tree[4].

We report the pseudo-code of the Mattson algorithm in Algorithm 1. For each object in the trace, if it is not found in the stack (the function *findPosition()* returns -1), there is a special counter hit_∞ to increment, which is used to count the number of objects seen for the first time (usually referred to as *cold start misses* or *compulsory misses*). Every time an object is requested, such object is moved at the top of the stack. Once the trace has been fully read, the Miss Ratio Curve (MRC), for different dimensions of the cache, can be simply computed by looking at the array $\mathbf{hit}$ and the counter hit_∞.

The Mattson stack algorithm assumes that all the objects have the same cost, therefore, in the original algorithm, the value of $cost_i$ is equal to one for all objects (see lines 9 and

4. This is known as the Olken algorithm [33], which we actually used in our implementation.

12). To compute the miss-ratio curve in the general case where objects have different costs, we propose a variant: for each requests, we update the stack distance with the corresponding cost, instead of simply incrementing the stack by one. In this way, the array 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.

Algorithm 1 Mattson Algorithm with costs

1. **Input:** trace // trace to be analyzed, with object id, size and cost
2. **Data: stack** // data structure that contains the objects
3. **Data: hit** // array that keeps track the position of hits
4. **Data:** $hit_\infty = 0$ // counter for objects seen for the first time
5.
6. **for each** object i in trace **do**
7. pos = findPosition(i, **stack**);
8. **if** $pos \geq 0$ **then**
9. $hit[pos]$ += $cost_i$;
10. removeElementAtPosition(pos, **stack**);
11. **else**
12. hit_∞ += $cost_i$;
13. **end if**
14. push(i, **stack**);
15. **end for**
16. **for** $k \leftarrow 1$..length(**hit**) **do**
17. $\text{MRC}(k) = 1 - \frac{\sum_{j=0}^{k} hit[j]}{\sum_{j=0}^{k} hit[j] + hit_\infty}$;
18. **end for**

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.

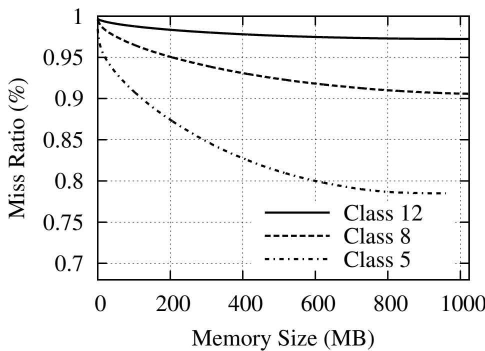

Fig. 1. Examples of the miss ratio curve for different classes. The miss ratio is computed considering in the denominator the total number of requests for all classes (this is why it is so high, on a per-class basis). The composition of all classes yields an overall miss ratio which goes, as the memory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

3.2 Memory partitioning: Offline Assignment

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 Mattson stack algorithm for all the classes, then we optimally assign portions of the memory to the different classes. For such a case, we need the MRC for the different classes. To this aim, we first need to compute the Hit Ratio Curve (HRC) for each class as shown below

$$\text{HRC}^w(k) = \frac{\sum_{j=0}^{k} hit^w[j]}{\sum_{m=0}^{\text{num classes}} \left(\sum_{j=0}^{k} hit^m[j] + hit^m_\infty \right)}. \quad (1)$$

Note that the denominator considers all the classes, so that HRC^w provides a measure of the fraction of the global hit ratio due to class w. Starting from the HRC^w, the hit ratio curve for class w, its MRC^w is given by

$$\text{MRC}^w(k) = 1 - \text{HRC}^w(k), \quad (2)$$

and the overall miss ratio curve is given by

$$\text{MRC}^{\text{tot}}(k) = 1 - \sum_m \text{HRC}^m(k); \quad (3)$$

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 an integer number of blocks. We assume also that the relative decrease of the miss-ratio, as the memory grows, is monotonically decreasing: in practice, for any class w, given a memory size k and the miss ratio $\text{MRC}^w(k)$ for that memory size, then

$$\text{MRC}^w(k) - \text{MRC}^w(k+1) \geq \text{MRC}^w(k+1) - \text{MRC}^w(k+2), \forall k.$$

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.

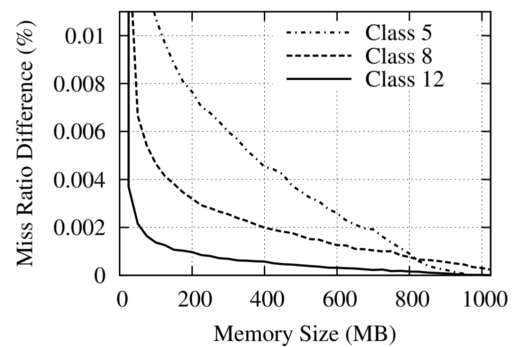

Fig. 2. Example of the miss ratio difference curve for different classes. The miss ratio difference is the decrease in the miss ratio when slabs are added to the class, one at a time.

Given the above assumptions, a simple heuristic can be used to compute the optimal assignment. Such a heuristic

has been inspired by [29], in which the authors look for the set of points that minimize the derivatives of the miss ratio curves. Here, we approximate such derivatives with the corresponding difference between two discrete values of memory sizes. Recently, different alternative approaches has been proposed to solve this assignment problem using dynamic programming [19] [11]. While these approaches have been developed independently and the same time with respect to our solution, they are computational intensive (time complexity is cubic) and obtain results similar to ours (if the MRCs are convex, as in our case).

In the heuristic we propose (see Algorithm 2), at each iteration, classes are sorted by the miss-ratio difference $\text{MRC}^w(k)$ - $\text{MRC}^w(k+1)$, where w is the class index and k is the number of blocks assigned to class w 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 2 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
```

Given the above assumptions, 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.

3.3 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 (compulsory misses), and misses for objects that have been evicted, usually referred to as *capacity misses*. This distinction is important, since we cannot avoid the compulsory misses, while the capacity misses are 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 data structure introduced in [26]: the Decaying Bloom Filter (DBF). The DBF is an extension of the Counting Bloom Filter designed to detect duplicates for data streams. Since detecting duplicates in an unbounded data stream is difficult to achieve, if not unfeasible, and we are interested in the case when the time between two different requests for the same object have short time scale, it is sufficient to consider an approximate solution that detects duplicates within a fixed time frame, i.e. over a sliding window. We have tested different sizes of the sliding window and observed that a size greater than 100’000 is sufficient to capture the vast majority of the popular objects: in other words, if the difference between two consecutive requests for the same object is greater than 100’000 requests, its contribution to the hit ratio is not significant.

Next, assume that all auxiliary information described above – the cost of the capacity misses per class $\mathbf{m}$, the cost of the requests per class $\mathbf{r}$, and the number of slabs allocated to each class $\mathbf{s}$ – is available. Our memory management scheme uses a *slab allocation algorithm* that we label SAS. The algorithm, outlined in Algorithm 3, evaluates a single slab movement, from a “rich” class, i.e. a class with many slabs and few misses, to a “needy” class, i.e. a class that would most benefit from additional memory.

Removing from a rich class: For each class we can compute the miss ratio $\frac{m_i}{r_i}$: a small miss ratio may indicate that a class can afford to lose some memory. Nevertheless, the miss ratio alone is not sufficient to provide a complete picture: we need to understand how efficiently a class is using the memory assigned to it. For instance, if we have two classes with the same miss ratio, but one class has two slabs assigned to it and the other has ten slabs, it is clear that the class with 2 slabs is using the memory more efficiently, and if we remove one slab from such a class, it will suffer, and the number of misses may increase significantly. On the other hand, if we remove a slab from the class with 10 slabs, the impact will be much more contained.
Therefore, we consider the miss ratio normalized to the number of slabs: $\frac{m_i/r_i}{s_i}$. This value is an indication of the increase in the miss ratio in case we decide to remove a slab from a class. While more sophisticated measures can be used to estimate the variation in the miss ratio when slabs are removed, our measurements have shown that the approach we propose is fairly accurate.

In summary, we define a rich class the class with the smallest value of $\frac{m_i/r_i}{s_i}$. Among the slabs of that class, we select the one according to the Least Recently Accessed (LRA) policy, i.e. we select the slab that has not been accessed for the longest time.

Giving to a needy class: The identification of the class that is suffering the most is much simpler. Since our aim is to decrease the overall number of misses, it is sufficient to look at the class that has registered the largest number of misses.

Algorithm 3 Slab Allocation Scheme (SAS)

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

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.

We observe that, when we remove a slab from a class, we need to evict all objects within such a slab. Since objects have different popularities, it may happen that we remove highly requested objects within that class, and the number of misses increase more than predicted. Clearly, once the rich class has been identified, it would be beneficial to reorganize the objects among the slabs, and put the least recently used objects in the slab to be evicted. This operation is computationally intensive and would lock the resources for a longer time with respect to our SAS scheme. On the other hand, as we will show in the results, our solution obtains nearly-optimal results and is computationally lightweight. Therefore, even if more complex policies would avoid evicting popular objects, the benefits that they could obtain may be marginal.

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. Due to the limited duration of the real traces (48 hours), we have not observed any significant change in the statistical properties of the requested objects. Therefore the results we show focus on the memory management scheme evaluation when the set of requested objects remains constant. The case where the set of objects changes, and their statistical properties change too (such as the distribution of the object size), introduces different problems (e.g., the calcification) that needs to be discussed separately: we consider this latter case, where we have run experiments with synthetic traces, in Section 5.

4.1 Experimental Setup

Typically, in scale-out Web applications, a series of Memcached servers are configured in a shared-nothing setup, whereby each server takes care of a subset of data objects using consistent hashing [7] 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[5].

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.

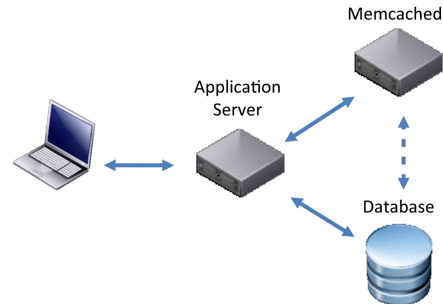

Fig. 3. An illustration of the testbed used in our experiments: this is a simple, yet representative, configuration.

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. In

5. Actually, with multiple threads, the overall throughput may increase, since memory lock occurs at the class level. Nevertheless, for the purpose of our experiments, the concurrent management of the classes has no impact on the main performance index we consider, i.e. the hit ratio.

TABLE 1
Information about the traces.

Length of the trace	48 hours
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

a real deployment, cost-information is usually provided by back-end services: we discuss in detail this aspect in Sect. 6. 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 resulting object popularity in Figure 4. 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. In particular, we refer to the Che's approximation [15], used to predict the hit ratio for a given cache size: the model assumes that all objects has the same size, the object popularity follows a Zipf-like distribution, and the arrivals follow a Poisson process. In our case, since the assumptions do not hold, we can not make use of such theoretical results, and we need to resort to the experimental approach.

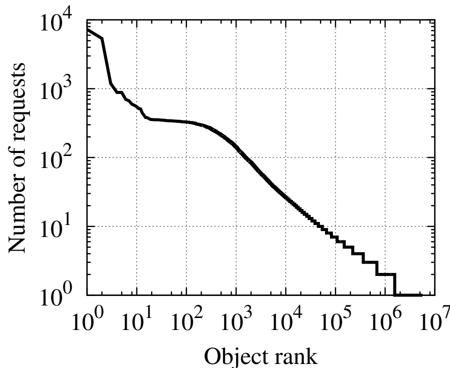

Fig. 4. Number of requests for each object, ordered by objects rank, from the most popular to the least popular.

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 5 shows the empirical CDF of object sizes.

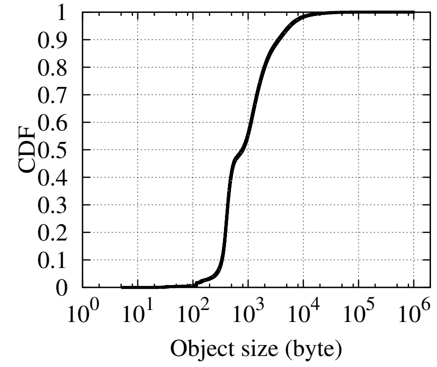

Fig. 5. Empirical cumulative distribution function of the object sizes.

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 6 shows the empirical CDF of the object costs.

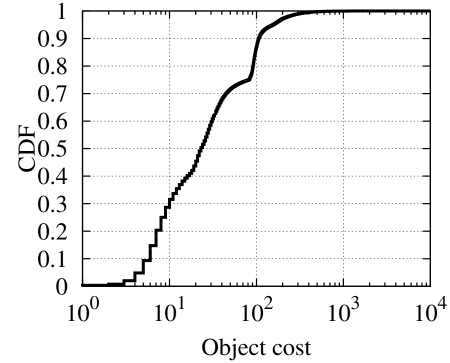

Fig. 6. Empirical cumulative distribution function of the object costs.

It is important to note that the retrieval time is not necessarily correlated to object sizes: Figure 7 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 managing 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.

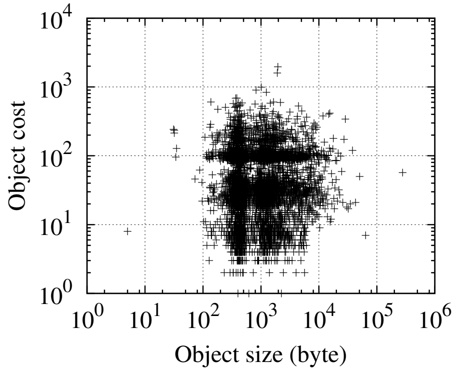

Fig. 7. Relation between object size and cost. Each point represents an object.

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 shown in Algorithm 1, along with Algorithm 2 presented in Section 3, to compute – in an off-line manner – the *optimal* cost hit ratio. 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. We show the optimal value with a horizontal line.

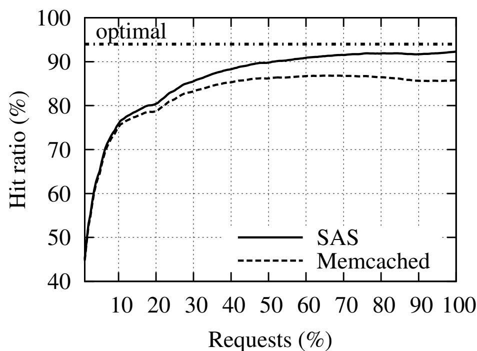

Fig. 8. Hit ratio obtained when all the objects have the same cost.

Figure 8 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.

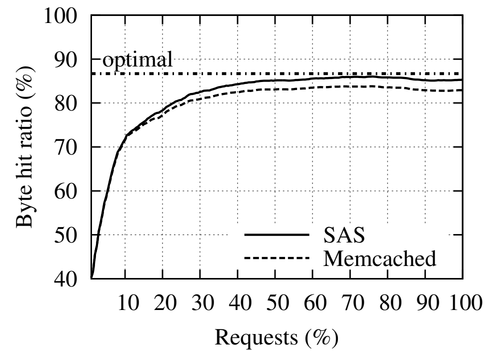

Fig. 9. Byte hit ratio obtained when the objects have a cost equal to their sizes.

In Figure 9 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. As for the SAS scheme, also in this case, its dynamic adaptation is able to slightly improve over Memcached.

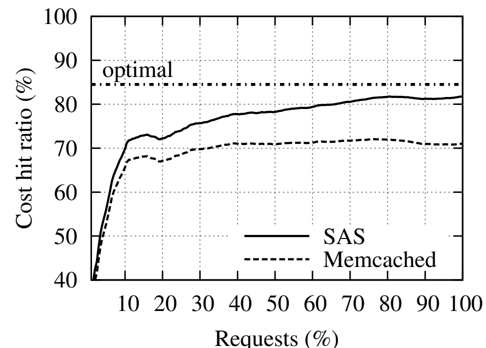

Fig. 10. Cost hit ratio obtained with retrieval times as objects costs.

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 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 11, 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 (the other snapshots are not shown here for space constraints), to the optimal one.

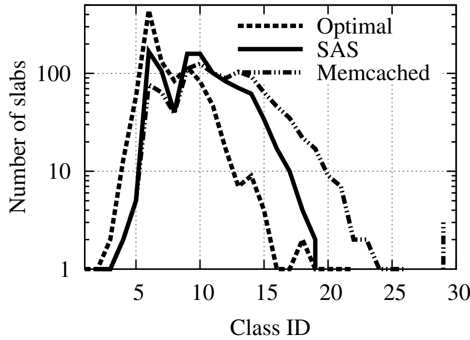

Fig. 11. Slab assigned to the different classes. The snapshot has been taken after half of all the requests has been processed.

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 The Calcification Problem

In Memcached, once an appropriate portion of memory has been assigned to a class, it will remain always associated to such class (unless the server is restarted). Clearly, if the statistical properties of the requested objects do not change over time, the hit ratio may be not affected by such a static slab allocation. Instead, when the statistical properties change (e.g., larger objects become more popular), the problem of *slab calcification* becomes tangible [6], and performance deteriorates. In the following, we summarize current attempts and known best practices to mitigate calcification:

Cache Reset: every T seconds all the objects are removed from the cache. This policy requires manual intervention, as it is not implemented in Memcached. Despite its simplicity, we note that the abrupt service interruption due to the reset is harmful in several aspects: *i)* client connections may result hanging; *ii)* several transitory periods may be required to fill the cache; and *iii)* the back-end servers and the database layer may suffer load spikes due to an empty cache. In a multi-server setting, the Reset has to be coordinated, so that the impact on the global hit ratio is limited.

Memcached Automove: a recent version of Memcached allows slab reassignment. Every 10 seconds, the systems collects the number of evictions in each class: if a class has the highest number of evictions three times in a row, it is granted a new slab. The new slab is taken from the class that had no eviction in the last three observations. As stated by the designers of this policy, the algorithm is conservative, i.e., the probability for a slab to be moved is extremely low (because it is rare to find a class with no eviction for 30 seconds).

Twitter random eviction: Twemcache [6] allows administrators to select a set of eviction policies explicitly designed to solve the slab calcification problem; with random eviction, for each `Set`, if there is no free chunk or free slab, instead of applying the class LRU policy, the server chooses a random slab (that can belong to any class), evicts all the objects in such a slab, reassigns the slab to the current class (by dividing the slab into chunks of appropriate size), and uses the first free chunk to store the new object – the remaining free chunks will be used for the next `Set` requests. This policy allows slabs to be reallocated among classes to follow request dynamics. However, since the eviction procedure is executed on a per-request basis and since slab eviction implies the eviction of all its stored objects, we believe the random eviction policy to be too aggressive. Our experiments confirm such claim.

Twitter slab LRA eviction: Twemcache provides also an alternative policy to overcome the limitation of the random policy. For each `Set`, if there is no free chunk or free slab, the server chooses the least recently accessed slab (that can belong to any class), evicts all the objects in such a slab, reassigns the slab to the current class, and uses the first free chunk to store the new object – the remaining free chunks will be used for the next `Set` requests. The access time of a slab is updated each time an object in such a slab is accessed. The policy aims at a dynamic slab-to-memory assignment, letting the slabs to be assigned dynamically to the classes, but the eviction of multiple items may have a negative impact on the hit ratio.

In the next section, we study with an experimental approach if these policies are able to efficiently address the calcification problem, and we compare the results with the ones obtained by our SAS scheme.

5.1 Experimental Setup

The experimental testbed is the same as explained in Section 4.1. In order to highlight the calcification problem, we have generated synthetic traces where the requested objects come from two sets with different statistical characteristics. The first set has $\mathcal{Q}_1 = 7$ Millions objects, whose size is randomly drawn from a Generalized Pareto distribution with location $\theta = 0$, scale $\varphi = 214.476$ and shape $k = 0.348238$ – these values have been reported by Atikoglu *et. al.* in [7]. The second set has $\mathcal{Q}_2 = 7$ Millions objects, whose size is randomly drawn from a Generalized Pareto distribution with different parameters: $\theta = 0$, $\varphi = 312.6175$ and $k = 0.05$. Even if calcification has been observed in some prominent operational setups [1], [6], [7], no detail has been given on the change of the statistical properties of the requested objects. Therefore our choice of the second set of parameters has been made to induce slab calcification. We have tested different distributions and parameters for the second set of objects, obtaining always the same qualitative results presented in the next sections.

Since Memcached and Twemcache does not support costs, for a fair comparison we set the cost of all the objects to one. To ensure a proper reproducibility of our results, we provide a set of traces that can be used by automatic scripts to populate a database, and to generate requests. In Sect. 6 we provide additional details about this.

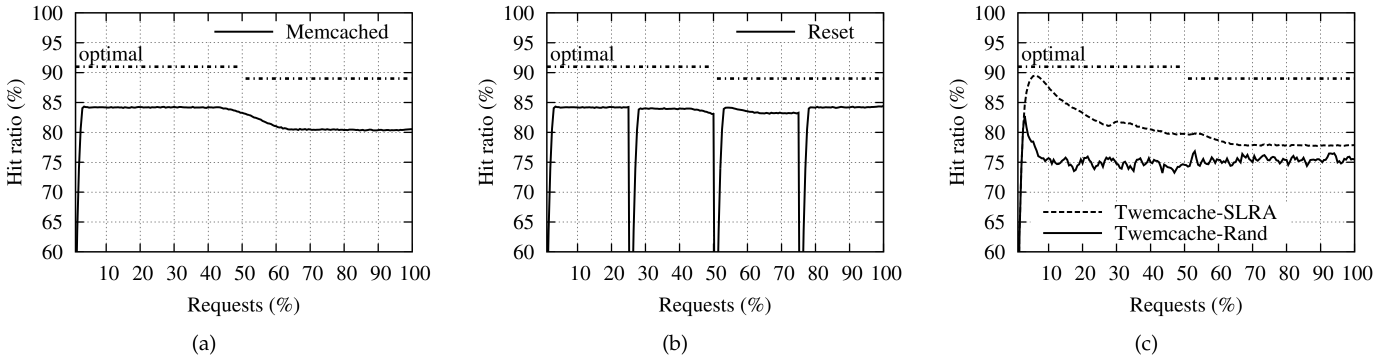

Fig. 12. Hit ratio over time with calcification obtained by different schemes.

Our experiments are built as follows. The client generates $\mathcal{R} = 200$ Millions requests, divided into three phases. In the first phase, the client selects random[6] objects from the first set; in the second phase, random objects of the second set are increasingly requested; in the third phase only random objects of the second set are considered. For each request, the application server registers a hit if the object is in the cache. To produce our results, we consider intervals of $R = 500'000$ requests and compute the aggregate hit ratio thereof.

Note that, while the first phase reproduces the usual behavior of the cache, the other phases have been designed to force the system to deal with a change in the statistical properties of the objects: this pinpoints the calcification problem in Memcached and allows to study the effectiveness of current countermeasures. Morevover, once the cache is full, it is not important how fast the statistical properties of the requested objects change. In other words, even if the change of the statistical properties appears over 400 Millions requests (instead of 200 Millions), the calcification will occur slowly, but it will appear in any case.

In order to compare with the optimal value, we have considered two portions of the traces, taken from the first and the last phase, and use these portions to feed the Mattson algorithm and the offline computation of the optimal slab assignment. Since the statistical properties of the requested objects change, the optimal value change, therefore we have two different values for the first and the last phase. Such values are shown as horizontal lines in the figures[7].

5.2 Results

We consider four system configurations: Memcached, Memcached with the reset policy, Twemcache with the random eviction policy, and Twemcache with the slab LRA policy[8]. Figure 12 shows the results, in terms of hit ratio over time.

Figure 12(a) indicates that, for Memcached, after an initial period necessary to fill the cache, in the first phase the hit ratio becomes stable at a value of roughly 84%. In the second and third phase, the impact of slab calcification on the hit ratio is evident, with a loss of 4%. As the client asks for more and more objects with sizes that have been drawn from a different distribution, the hit ratio decreases progressively. During all the trace it is clear that the hit ratio obtained by Memcached is far from optimal, as we already observed in the previous sections. Nevertheless, the calcification has a additional negative impact: while the optimal value changes from 91% to 89%, with Memcached the decrease is 4%.

With the Reset policy, shown in Figure 12(b), we impose a cache reset four times during the experiment. The resets mitigate the effects of slab calcification. During the transition among object sets, the hit ratio is affected by different object size distributions: this is clearly visible in the third "wave." However, once the transition is over, Memcached can restore the hit ratio to a similar value to that of the first phase, which is in any case far from the optimal value. Clearly, each reset action provokes a transitory period to fill the cache, which affects negatively the achieved hit ratio.

Figure 12(c) shows that the random eviction policy in Twemcache achieves a lower, and *extremely variable* hit ratio, when compared to Memcached. As we anticipated in Sect.2, the eviction of randomly selected slabs may be too aggressive, because an individual slab may contain many popular items. As such, using Twemcache in conjunction with the random eviction policy has a negative impact on the hit ratio overall. Our experiments show that also the slab LRA policy in Twemcache obtains a smaller hit ratio than Memcached in the long run, albeit performing better than random eviction. The reason why the slab LRA policy is close to optimal at the beginning is due to the initial slab assignment, which is biased to classes with small objects. In fact, the slab LRA policy choses the slab to be evicted based on the access time. Since classes with large objects contain less objects (and overall, they contribute less to the hit ratio), they are chosen more frequently at the beginning, but they have less impact

6. The probability distribution used to identify the object to request is a truncated Normal distribution that shifts over the object identifiers as the experiment progresses: in this way we emulate artificially the change in popularity of the objects. Note that, while popularity may have different distributions, the objects size are selected from the sets $\mathcal{Q}_1$ and $\mathcal{Q}_2$, therefore the choice of the popularity has no impact on the memory allocation (more details on this point can be found in [13]).

7. The optimal values can be used as a reference for the first and the last phase, while in the middle it is not possible to compute an optimal value due to the changing statistical characteristics of the objects.

8. We omit the Memcached Automove policy, since we have verified that, in our experimental setup, no slab has been moved.

on the hit ratio. This results in a memory assignment which provides good performances at the beginning. Nevertheless, in the long run, the evolution of whole slabs reassignment have a significant negative impact. Overall, the two Twemcache eviction variants under-perform Memcached (with calcification) and may be unstable.

Figure 13(a) shows how the hit ratio achieved by SAS compares to that of Memcached, in each experiment phase. With no calcification (first phase), SAS achieves a 7% increase over vanilla Memcached and almost reaches the optimal value; in presence of calcification (third phase), the gain in favor of SAS reach 10%, with a near-optimal value. Note that the hit ratio in the third phase is lower than that in the first phase: this is due to the particular object size distribution of the last phase, and should not be attributed to the consequences of calcification. This is confirmed by the fact that the optimal value is also diminished, and the results of SAS is close to such value, As additional test, we run an experiment where we impose an artificial reset to the SAS-based Memcached server: with the reset, we make sure that memory partitioning is "molded" according to the final object size distribution, following client requests. Figure 13(b) shows that, after the reset, the hit ratio converges to its previous value.

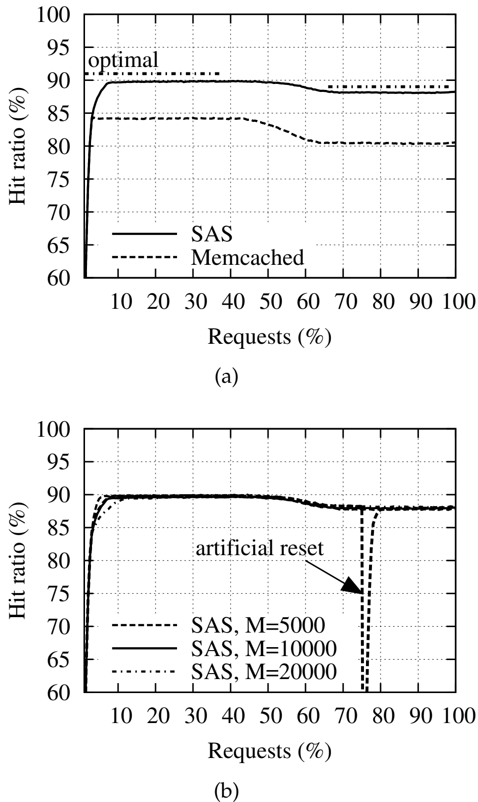

Fig. 13. Hit ratio over time obtained by SAS compared with vanilla Memcached (a) and by varying the parameter M (b).

Finally, we conclude by observing that SAS 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 5'000 and 20'000 misses, and observed that any value within this range provide similar results (see Figure 13(b)).

6 Discussion

We now discuss additional details about the SAS policy and the experiments we performed to validate it.

Overheads and complexity: The SAS policy requires storing, for each object, an additional cost information, which might take space. The overhead can be computed for each single slab (1 MB): in our experiments, considering the slab that contains the smallest objects, we have approximately 11'000 objects. If we use 4 bytes for representing the cost with a float, the overall additional size required is approximately 43 kB, i.e, 4% of the slab size. For slabs with bigger objects, this reduces to 1% or smaller overheads. Overall, even considering the additional data structures – such as the counters for each class – the overhead is limited.
As for the computational complexity, SAS is a lightweight algorithm, in line with what is currently implemented in Memcached and Twemcache. To operate, SAS locks the memory in two occasions. First, to compute, given the internal status of Memcached, the overall statistics $(\mathbf{r}, \mathbf{m}, \mathbf{s})$. This takes $O(C)$, where C denotes the total number of classes. Second, to move a single slab, consisting in removing all objects in the slab to evict: this takes $O(\text{items})$. SAS is executed every M misses, meaning that a round has no fixed duration. Compared with the frequency of slab movements performed in Twemcache, we observed that SAS does not impose a high toll on system resources in an operational setting. In order to confirm this, we have measured the throughput, i.e., the number of requests per seconds that are processed by Memcached, SAS and the two policies implemented in Twemcache (Figure 14). The values have been obtained with 5 different runs for each policy, using different seeds for the generation of the objects and arrival patterns. The small bars at the top of each box indicates the 95% confidence interval. Rather than the absolute values of the throughput (which, for instance, is influenced by the

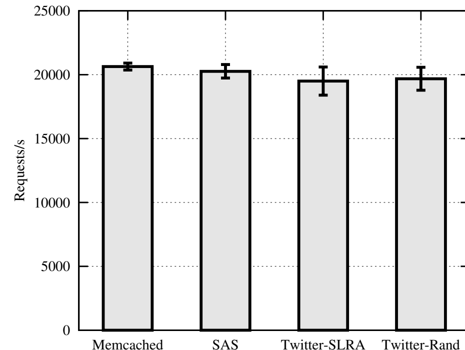

Fig. 14. Average throughput of the different schemes.

distribution of the object size), it is interesting to note that SAS has a limited impact on the performance in terms of throughput with respect to Memcached, and its burden is comparable to the one imposed by Twemcache. This is true also with different values of the parameter M, the length of the evaluation period.

Impact on the latency: While the throughput provides a measure of how fast the cache is, it would be interesting to understand the impact of caching on the overall response latency, which includes the time necessary to retrieve the content from the back end. It is worth noting that even modest improvements to the cache hit ratio has significant impact on the average latency. For instance, assuming an average read latency from the cache and from the backend of 100 μs and 10 ms respectively, increasing the hit rate by 6% (e.g., from 84% to 90%) would reduce the average read latency by over 35%. Clearly this result strongly depends on the overall architecture: if the backend provides a read latency of 2 ms, then increasing the hit rate from 84% to 90% would reduce the read latency by over 28%; if the backend needs to retrieve the content from an origin server far from the current cache, then the read delay would be tens of milliseconds. Therefore, the response time strongly depends on the architecture of the overall system, and it can be evaluated accurately only in a real deployment.

Rather than on the overall architecture, we may focus on the latency of the cache itself, and the impact of the different schemes on such metric. In the example above, we have assumed that the average read latency from the cache is the same before and after the increase of the hit rate. Nevertheless, if the scheme that helps in increasing the hit rate has an impact on the average read latency, the overall latency may be affected. To shows that this is not the case, we have measured the average latency of the `Get` requests and built the corresponding CDF – due to the high number of requests, each point in the CDF represents the average latency of 10'000 `Get` requests.

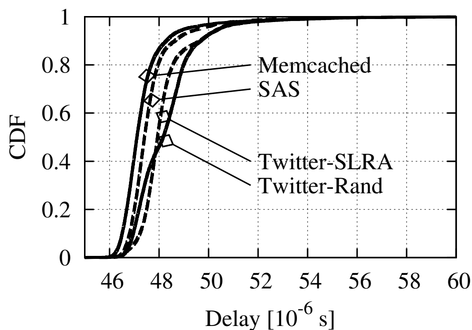

Fig. 15. CDF of the average latency of the `Get` requests.

Figure 15 shows that the different policies – SAS and the variants of Twemcache – have little impact on the `Get` request latency with respect to Memcached: for instance, for any percentile, the delay of SAS is at most 1 μs larger than Memcached. Overall, therefore, SAS is able to improve the hit ratio with negligible impact on the cache throughput and on the average cache latency.

Compulsory and capacity misses: In our work, we distinguish between compulsory misses (objects that have never been requested before) and capacity misses (objects previously evicted). This distinction is important since we cannot avoid compulsory misses, but we can direct the resources we have (the slabs) to decrease the capacity misses. Note that, in general, there is little correlation between compulsory and capacity misses, i.e. a class with high compulsory misses not necessarily has high capacity misses too. In other words, we can not take the total number of misses, without distinguish between compulsory and capacity misses, as an indication of the capacity misses. While designing our scheme, we have tested a solution that considers the total number of misses, obtaining a significant performance degradation with respect to the solution where the distinction between misses is made (not shown here for space constraints). On the other hand, the cost of the Decaying Bloom Filter used to detect duplicates in the data stream is negligible, and the benefits it provides in terms of performance significant.

Traces: The experimental evaluation of cache eviction policies or memory partition mechanisms, requires rather complex setups. First, it is necessary to populate a database system with millions of objects, defining minimum and maximum sizes, along with an appropriate definition of size distributions. Then, it is essential to define client requests for such objects (object popularity, and its dynamics). For experimental reproducibility, a specification of such parameters is key, in conjunction to measurement studies to inform the design of realistic distribution shapes – a methodology adopted in this work, building on the information discussed by Atikoglu *et. al.* in [7].

Nevertheless, the performance analysis of a caching system can be made smoother by building an appropriate set of software tools to accomplish the above in an automatic manner. With such tools, it is possible to reproduce exactly the same experimental conditions used to study system performance, making it possible to compare and benchmark a variety of existing and new memory management mechanisms.

Today, only a scattered set of pieces of software is available in the open-source domain to realize experiments: most of them, however, fall short in providing realistic setups and simplicity, due to the number of internal parameters they require. In our work, we attempt to address such problems by creating a set of traces that can be used by automatic scripts *(i)* to populate a database, and *(ii)* to generate requests. The format of these traces is extremely simple: those used to populate the database are a series of entries with $(id, size)$ of the objects; those used to generate client requests are a series of object identifiers. The interested reader can find details, scripts and the traces in [2].

Additional experiments: Due to the the limited availability of public traces, we could test our scheme on a few, yet representative, scenarios. Nevertheless, the basic version of our scheme, which did not include the cost, called PSA, has been implemented and tested in other works [19] [23]. Both works use a set of traces derived from the Facebook workload [7]. In addition, in [19], the authors performed a *stress-test*, i.e., they use a workload designed to test the throughput

of the server inspired by [9]. In all the experiments, SAS is able to obtain almost optimal performance in terms of hit ratio, confirming its effectiveness in adapting to different scenarios.

On the cost of the objects: When using the new API we introduced in Memcached, the operator needs to specify the cost of the object when storing it into the cache. One option could be to use the size of the object itself. Another option would be to use the delay to retrieve the content from the backend. Usually, such information is indeed readily available in current systems. Many production systems, in fact, deploy a monitoring infrastructure for measuring the performance of the system itself. For instance, in the case of the traces we used, the cost was actually recorded along with the traces. Such cost is the overall delay for obtaining the object, which include not only the time necessary to transfer the bytes, but also the time for additional operations, such as decompression. Note that not all objects are compressed, it depends on different factors (non only size) that are related to the store management policies adopted by the operator (which may be different from the cache management policies). In any case, any system such as a web application infrastructure, that needs to closely observe its performance, deploys such monitoring system, so that the information about the cost is available when retrieving the object.

7 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.

Such sub-optimal performance are present even when the statistical characteristics of the cached objects change over time. While calcification has been discussed and cited in technical blogs [1] and some papers [7], [22], we have shown, to the best of our knowledge for the first time, its impact on the hit ratio. We have also studied Twemcache, a variant conceived at Twitter that includes eviction policies to address calcification, and showed that the price Twemcache pays for adaptivity is a lower hit ratio. Our scheme, instead, is able defeat the calcification problem, yet maintaining close to optimal performance.

As a future work, we intend to investigate alternative in-memory storage systems, such as Redis [5], 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` [3]. 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.

References

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

[2] enchmarks for testing memcached memory management. http://profs.sci.univr.it/~carra/mctools/. Accessed: 2015-06-01.

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

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

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

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

[7] 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.

[8] 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.

[9] H. Bjornsson, G. Chockler, T. Saemundsson, and Y. Vigfusson. Dynamic performance profiling of cloud caches. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 59, 2013.

[10] 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.

[11] J. Brock, C. Ye, C. Ding, Y. Li, X. Wang, and Y. Luo. Optimal cache partition-sharing. In *Parallel Processing (ICPP), 2015 44th International Conference on*, pages 749–758. IEEE, 2015.

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

[13] D. Carra and P. Michiardi. Memory partitioning in memcached: An experimental performance analysis. Technical report, Department of Computer Science, University of Verona, June 2013.

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

[15] H. Che, Y. Tung, and Z. Wang. Hierarchical web caching systems: Modeling, design and experimental results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

[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] X. Hu, X. Wang, Y. Li, L. Zhou, Y. Luo, C. Ding, S. Jiang, and Z. Wang. Lama: Optimized locality-aware memory allocation for key-value cache. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, pages 57–69, 2015.

[20] 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.

[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] 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.

[23] J. Ou, M. Patton, M. D. Moore, Y. Xu, and S. Jiang. A penalty aware memory allocation scheme for key-value cache. In *Proceedings of International Conference on Parallel Processing (ICPP)*, pages 530–539, 2015.

[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] H. Shen and Y. Zhang. Improved approximate detection of duplicates for data streams over sliding windows. *Journal of Computer Science and Technology*, 23(6):973–987, 2008.

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

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

[29] 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.

[30] C. A. Waldspurger, N. Park, A. Garthwaite, and I. Ahmad. Efficient mrc construction with shards. In *13th USENIX Conference on File and Storage Technologies (FAST 15)*, pages 95–110, 2015.

[31] 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.

[32] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, and A. Warfield. Characterizing storage workloads with counter stacks. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 335–349, 2014.

[33] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 31(6):20, 2009.

Damiano Carra received his Laurea in Telecommunication Engineering from Politecnico di Milano, and his Ph.D. in Computer Science from University of Trento. He is currently an Assistant Professor in the Computer Science Department at University of Verona. His research interests include modeling and performance evaluation of peer-to-peer networks and distributed systems.

Pietro Michiardi received his M.S. in Computer Science from EURECOM and his M.S. in Electrical Engineering from Politecnico di Torino. Pietro received his Ph.D. in Computer Science from Telecom ParisTech (former ENST, Paris). Today, Pietro is a Professor of Computer Science at EURECOM. Pietro currently leads the Distributed System Group, which blends theory and system research focusing on large-scale distributed systems (including data processing and data storage), and scalable algorithm design to mine massive amounts of data. Additional research interests are on system, algorithmic, and performance evaluation aspects of computer networks and distributed systems.