Memory Partitioning in Memcached: An Experimental Performance Analysis

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

Pietro Michiardi
Eurecom
Sophia Antipolis, France
pietro.michiardi@eurecom.fr

***Abstract*—Memcached is a popular component of modern Web architectures, which allows fast response times 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 rate. 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, a problem called "calcification". Calcification constitutes a symptom indicating that current memory partitioning mechanisms require a more careful design.**

First, using an experimental approach, we evaluate how memory is assigned to the different classes, and quantify the impact of calcification on the hit rate. We then proceed to design and implement a new memory partitioning scheme, called PSA, which replaces that of vanilla Memcached. With PSA, Memcached achieves a higher hit rate than what is obtained with the default memory partitioning mechanism, even in the absence of calcification. Moreover, we show that PSA is capable of "adapting" to the dynamics of clients' requests and object size distributions, thus defeating the calcification problem.

I. Introduction

Modern Web architectures are designed to provide low latency response times to thousands of requests per second originated by a large number of clients trying to access, for example, a complex Web page. To achieve such goal, a common solution is to keep a large fraction of the data served by a website in main memory. In this context, Memcached [1] is a popular and well-integrated component of such Web architectures: it is a key-value store that exposes a simple API to store and serve data residing in the DRAM of a machine. Thanks to its simplicity and efficiency, Memcached has been adopted by many companies, such as Wikipedia, Flickr, Digg, WordPress.com, Craigslist, and, with additional customizations, Facebook and Twitter.

Recent studies [2], [3] have mainly focused on the throughput achieved by Memcached, i.e., how fast Memcached can serve data to satisfy a request. Every operation, in fact, requires to lock the memory, therefore even multi-threaded approaches may not be able to exploit efficiently the memory.

In this work, instead, we consider a different problem: we study how *memory allocation* performs in Memcached. 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, which takes into account the typical size distribution of web data objects. When a new object – that is, an object that was never requested before – 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 favor of the new one. In this context, a common metric to determine the performance of Memcached – as for any other caching mechanisms – is the *hit rate*, defined as the number of requests that can be served directly from memory over the total number of requests received by the Web server.

Essentially, in Memcached, memory allocation is the process of assigning portions of the memory to each class: memory is granted to a class based on the requests received for objects belonging to that class. Once all the available memory has been allocated, memory reallocation – that may be triggered by a change in the statistical properties of the requested objects – is not supported.[1] Such a strict approach to memory allocation raises a problem often referred to as *calcification* [4]. Clearly, if the statistical properties of the objects (i.e, the distribution of the object sizes) that are stored in Memcached does not change over time, then calcification does not affect performance, because the baseline eviction policy currently implemented in Memcached works well for storing "hot" objects. However, if the statistical properties of object sizes change over time, calcification has a direct impact on the hit rate achieved by Memcached – this problem has been reported by Twitter [5] and Facebook [6].

The straightforward solution to calcification is to allow Memcached to reassign memory previously allocated to a given class. Although memory reallocation may appear as a simple problem to solve at a first glance, we identify a number of challenges in doing so. How can the system detect whether calcification has occurred? What can be used as an indication that a class needs to be granted more memory than another class? How much memory should be reallocated from one class to another class? How often should the system evaluate memory allocation and proceed with reallocation?

The above questions suggest that, rather than focusing on calcification alone, it is reasonable to analyze the *overall process of memory allocation*: is the proportional allocation method implemented in Memcached the best approach to address memory assignment? Despite the clear consequences on hit rate, this problem has received little attention in the literature.

Our contributions: In this paper we analyze the memory

[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; see III-B for the details.

allocation mechanism of Memcached. We take an experimental approach and evaluate how memory is assigned to the different classes, and quantify the impact of calcification on the hit rate. 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 – and we generate the objects according to the size distribution provided by Atikoglu *et. al.* [6].
We then set off to design a new memory allocation scheme to replace that implemented in vanilla Memcached: the gist of our mechanism is to measure the absolute values of cache misses as an indication that a class needs more memory; the memory required by a problematic class is taken from the class that would suffer the least, would its memory be reallocated. We implemented our new memory allocation mechanism – which is computationally efficient and lightweight – and compare its performance to both vanilla Memcached and Twemcache, using the same experimental approach outlined above.

Our results indicate that the memory allocation mechanisms of both Memcached and Twemcache are far from being optimal. With our scheme, the hit rate in the absence of calcification is higher than that of both Memcached and Twemcache, which underlines the importance of memory allocation in general. In addition, we show that whereas the calcification problem has a non negligible effect on Memcached, our mechanism is capable of "adapting" memory allocation to the dynamics of clients' requests and object size distributions, making it practically oblivious to calcification.

The rest of the paper is organized as follows. In Sect. II we provide the necessary background to understand how memory allocation works in Memcached and we discuss related work. In Sect. III we overview a variety of countermeasures to the calcification problem currently available in the state of the art. In Sect. IV we describe our experimental setup and we provide the experimental results for vanilla Memcached and Twemcache. We describe our mechanism in Sect. V, and we show the experimental results obtained using a modified version of Memcached based on the new memory allocation. In Sect. VI we present our effort to produce and distribute a common benchmarking suite for the evaluation of memory management schemes for Memcached, and we conclude the paper in Sect. VII.

II. Background

A. Memcached

Memcached is a key-value store that keeps data in the DRAM, i.e., data is not persistent. Clients communicate with Memcached through a simple set of APIs: `Set`, `Add`, `Replace` are used to store data, `Get` or `Remove` are used to retrieve or remove data. Although it is well known that, in scale-out Web application, a series of Memcached servers can be configured in a shared-nothing setup, whereby each server takes care of a subset of data objects using consistent hashing [6], in this work we focus on a single instance of a Memcached server.

Memcached has been designed to be extremely fast: not only it stores the data in the DRAM, but it also organizes the memory dedicated to storing data objects to simplify its management [2]. Every operation, in fact, requires memory locking:[2] therefore, data structures must be simple and their access time should be kept as small as possible. Memory management in Memcached plays an important role: next, we focus on important technical details.

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. Note that chunks are statically assigned to a slab, i.e., one chunk (the physical memory) can not be moved from one slab to another.

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 – that is the whole available memory has been allocated – and a class receives a request for a new item currently not stored in Memcached (and there is no free chunks), the new object replaces (*evicts*) a previously stored object: the eviction policy used in Memcached is the Least Recently Used (LRU). The LRU policy is applied on a per-class basis: in fact, items in other classes are stored in chunks of memory with different sizes, and chunks can not be moved.

In summary, the memory organization used in Memcached (slabs, chunks, classes) has been designed to favor management efficiency (fixed number of classes, fixed number of chunks is a slab assigned to a specific class). An alternative approach to the one described above, which would allocate "any" available memory on a per-request basis, would incur in performance bottlenecks (e.g., the eviction needs to take into account the size of the item to be evicted, so that there will be enough room for the new object) and memory fragmentation.

B. 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 a 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

[2]Note that memory locking is required even in case of a `Get`, since access time statistics need to be updated.

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

production site – in our experiments we will use these statistics to generate our workload. Nevertheless, the paper does not analyze and compare the different eviction policies, and it does not consider the impact of memory partitioning on the hit rate.

Gunther *et. al.* [7] highlight that Memcached has scalability issues, since threads access the same memory, therefore locks prevent the exploitation of the parallelism. For this reason, a number of works [2], [3] consider the performance of Memcached in terms of response time and request throughput, proposing a set of mechanisms and data structures to decrease the overall latency. Again, these works do not consider explicitly the impact of the memory partitioning on the hit rate as we do. Nishtala *et. al.* [8] study scalability problems, i.e., how to manage a multi-server architecture, but they do not study the eviction policies and memory partitioning.

As for the general literature on caches (not focused on Memcached), there exists a vast amount of works: caches, in fact, have been used at different levels – CPU caches [9], browser caches [10], Web caches and proxies [11], DNS caches [12], and Content Delivery Networks [13] – and each level is characterized by different problems.

Among these works, CPU caches need to solve similar problems to ours. In a CPU cache, many processes share the memory space, and a single process may "pollute" the cache with its data [14], 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 rate. The solution adopted for CPU caches [14]–[16] are based on a common idea, in which the memory partitioning process tries to balance the number of misses among the processes. Our memory partitioning scheme has been inspired by these works, i.e., we have adapted these schemes to the specific context of Memcached.

In the other types of caches (e.g., Web, DNS) memory partitioning does not represent an issue: such works [10], [11], [17] are mainly focused on the impact of the eviction policies when managing objects with different sizes. In Memcached, instead, eviction is done on a per-class basis, and objects within a class have the same size.

III. Object Popularity and Slab Calcification

We now describe in detail the occurrence of the calcification problem, which is mainly due to object popularity dynamics (that is, variations in the statistical characteristics of the requested objects). As discussed earlier, calcification may decrease the cache hit rate, that is the number of requests that can be satisfied through the cache over the total number of requests. In addition, we present the countermeasures to the calcification problem currently available in the literature.

A. Popularity Dynamics

Given a finite set of objects, and given an *interval of time* ΔT, we define the popularity of an object as the ratio of the number of requests for such an object and all the requests received during ΔT. Clearly, object popularity varies across different objects: this is the reason to use a cache in the first place. In addition, the popularity of a given object may vary over time: this is due, for example, to time of the day effects, or exogenous events such as (a set of) objects being featured on an external and influential web site or blogging platform.

If the popularity of objects changes rapidly, the maximum achievable hit rate of the cache may be far from an ideal value. To explain this, we model popularity dynamics by considering the removal or the addition of items in the object space: a removed object is an object with popularity equal to zero, and a new object is an object whose popularity switches from zero to a finite value. If the number of requests during subsequent time intervals ΔT remains constant (R requests per ΔT), and N objects are added to the object space (i.e., they switch from zero popularity to a finite value because they are requested at least once during ΔT), then the maximum hit rate achievable is equal to $(1.0 - N/R) \cdot 100\%$: in fact, N out of R requests will be for new objects not included in the cache, therefore at most $R - N$ out of R requests can be found in the cache.

In Memcached, the allocation of the slabs to each class depends on the popularity of the first "wave" of objects that is sufficient to fill-up the cache, considering an empty cache at startup; in particular, the memory is assigned proportionally to the popularity of the different classes, where the popularity of a class is given by the sum of the popularity of the objects in the class. 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).

We note that, despite *object popularity* dynamics, the popularity of a given class may remain constant, therefore the allocation made by Memcached may not negatively impact the hit rate as object popularity evolves over time. Instead, problems arise when the *class popularity* changes over time: this happens when the statistical properties of the requested objects change (e.g., larger objects become more popular). Since slabs can not be reassigned to a different class to accommodate requests/popularity dynamics, the memory allocation computed by Memcached cannot "follow" this change, i.e., the fixed memory allocation is not proportional to the new class popularity. This is usually referred to as *slab calcification* [5].

Slab calcification has a negative impact on the hit rate. If the popularity of a class decreases, then the class has more space than necessary, and the improvement in the hit rate is marginal – its popularity is in any case decreasing. On the other hand, when the popularity of a class increases, the space in the cache for that class is limited (because before it was less popular), and the hit rate significantly decreases.

B. Current Solutions to Slab Calcification

Slab calcification has been observed in real world deployments. In the following, we summarize the current solutions or the best practices that aim at solving this issue.

Cache Reset: This is the simplest solution, i.e., every T seconds all the objects are removed from the cache. In a multi-server setting, the Reset has to be coordinated, so that the impact on the hit rate is limited. Currently there is no implementation of this policy within Memcached, i.e., the server should be restarted with an external intervention (automatic, such as a script, or manual). The abrupt interruption of the service may cause some problems, since open connections are lost. Moreover, once restarted, there is a transitory period before the cache becomes full again and, consequently, the hit rate momentarily decreases. Finally, such a technique may impose a strain on back-end servers and the database layer, which suffer a spike on the number of requests to serve.

Memcached Automove: Starting from version 1.4.11, Memcached introduced the possibility to move slabs among classes. 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 rare to find a class with no eviction for 30 seconds).

Twitter random eviction: Twemcache [5] 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 rate.

Next, we study the effectiveness of existing policies to mitigate the impact of calcification, through a series of experiments executed in a simple, yet representative testbed. In our results we omit the Memcached Automove policy, since we have verified that, in our experimental setup, no slab has been moved.

IV. Establishing Baseline Performance

We now study the performance of Memcached and Twemcache in terms of hit rate achieved by each system. Next, we describe the experimental setup we use to obtain our results, and we properly define object characteristics and popularity. In our experiments, we first proceed by defining a baseline scenario in which there is no calcification; then, we introduce popularity dynamics and quantify the impact of calcification as well as the effectiveness of current countermeasures available in the literature.

A. Experimental Setup

Testbed configuration: We use a simple, yet representative, Web architecture that is illustrated in Fig. 1. A single client issues requests for objects that are permanently stored in a database server. An application server receives the requests from the client; the server checks if the requested object is stored in the cache, i.e., it sends a `Get` to Memcached; if Memcached returns the object, the application server sends it to the client, otherwise it retrieves the object from the database and sends it to the client. When an object is retrieved from the database, it is stored in Memcached: this can be done either by the application server or the database itself – in our testbed the application server takes care of it. Note that in our experiments, we set the cache size to 1 GB.

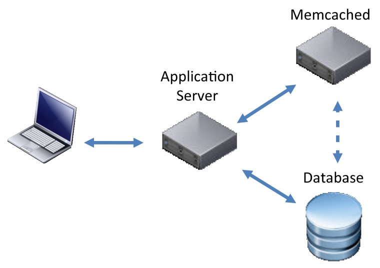


Fig. 1. An illustration of the testbed used in our experiments: this is a simple, yet representative, configuration involving a single application server, database and Memcached instance.

Object Characteristics: In our experiments, the database is populated with $\mathcal{Q} = 14$ Millions objects, whose size is randomly drawn from a Generalized Pareto distribution, as described by Atikoglu *et. al.* in [6]. Since our objective is to study the ill effects of calcification, we prepare two kinds of experiments.

In the first experiment, we use a single object size distribution for all objects in $\mathcal{Q}$: a (truncated) Generalized Pareto distribution with location $\theta = 0$, scale $\varphi = 214.476$ and shape $k = 0.348238$. We anticipate that in this experiment, slab calcification does not occur, despite object popularity dynamics.

In the second experiment, the object space $\mathcal{Q}$ is partitioned in two equal sets. The size of $\mathcal{Q}/2$ objects is generated according to a (truncated) Generalized Pareto distribution with the same parameters used in the first experiment. The size of the remaining $\mathcal{Q}/2$ objects follows a (truncated) Generalized Pareto distribution with different parameters: $\theta = 0$, $\varphi = 312.6175$ and $k = 0.05$. These latter parameters induce an object size distribution with the same mean but different variance with respect to the first half of the object space (we will explain in Sect. IV-D the reason for this choice). The particular manner in which we build the object space for the second experiment, in addition to popularity dynamics, induces slab calcification.

Finally, note that in all experiments, the minimum and the maximum object sizes are set according to the default values of Twemcache.

Object Popularity: In our experiments – as it commonly happens in real-life setups – the client issues requests for objects according to their popularity. Next, we describe how we model[4] object popularity dynamics.

[4]Object popularity dynamics is also influenced by time-of-day effects. In this work, we neglect such effects. An alternative approach to what we present here would be to use real-life traces of client requests, but we are not aware of any publicly available traces to do so.

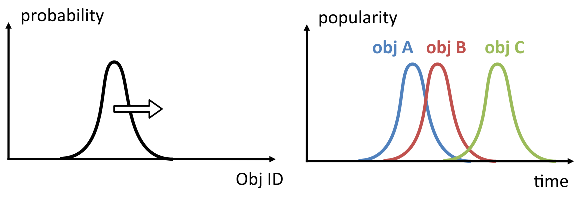


Fig. 2. Intuitive illustration explaining how the object selection process (left) and object popularity over time (right) are intertwined.

First, we neglect requests arrival times: we only consider the order of their arrivals. As such, time is slotted in discrete intervals, which essentially indicate the total number of requests received so far by the application server. This simplification is a direct consequence of how caching works.

Now, let $\mathcal{Q}$ indicate the size of the object space (as defined above), $\mathcal{R}$ the total number of requests, and ϵ a constant value between 0 and 0.5 (in our experiments, we set $\epsilon = 0.15$). We assume that the probability for a request to select a particular object follows a Normal distribution with mean μ_t and standard deviation σ (in our experiments, $\sigma = 625000$). Note that the mean μ_t is a function of time: namely, it depends on the t-th request (while the standard deviation remains constant). Essentially, the probability for a request to select an object to retrieve is characterized by a distribution whose mean shifts over time, from an initial position and with a certain speed (see Fig. 2, left). The initial position of the mean is equal to $\mathcal{Q}\epsilon$, while the last position is equal to $\mathcal{Q}(1-\epsilon)$. The "shifting speed" is constant, therefore we have the following relation:

$$\mu_t = \mathcal{Q}\left((1-2\epsilon)\frac{t}{\mathcal{R}-1}+\epsilon\right), \qquad t = 0, 1, 2, \ldots, \mathcal{R}-1 \quad (1)$$

We are now ready to define object popularity, and describe how it evolves over time. Let $\Phi(x)$ be the Cumulative Distribution Function (CDF) of the standard Normal distribution (mean ad standard deviation equal to zero and one respectively); then, the probability that an object i is selected by a client request is given by[5] the following expression:

$$p_i(\mu_t, \sigma) = \Phi\left(\frac{\lceil i-\mu_t \rceil}{\sigma}\right) - \Phi\left(\frac{\lfloor i-\mu_t \rfloor}{\sigma}\right) \qquad (2)$$

Expression 2 indicates that object popularity over time follows a (discretized and truncated) Normal distribution with its peak when the difference between i and μ_t is minimal, and standard deviation σ (see Fig. 2, right).

Experiment description and metrics: Our experiments are built as follows. The client generates $\mathcal{R} = 200$ Millions requests, selecting objects with a probability that depends on their popularity as described above. 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 requests ($R = 500'000$) and compute the aggregate hit rate in such intervals. With the settings described above, the maximum achievable hit rate is approximately 95% (cf. Sect. III-A).

In the following, we also report a number of internal statistics collected automatically by both Memcached and Twemcache. This is done by taking "snapshots" of the internal state of such systems, which report the number of hits, misses and objects stored in the cache, for each class. In our experiments, the application server is instructed to take such snapshot every 20 Million requests.

[5]For notation simplicity, we use here the CDF of the Normal distribution. Nevertheless, during our experiments, we have used the CDF of the *truncated* Normal distribution, since the support (the objects) is finite. Note also that the parameter ϵ – which is used to compute μ_t – makes the effects of truncation negligible: the initial position of the request distribution is different from zero.

B. *On the Relation Between the Hit Rate and the Experiment Parameters*

The hit rate is influenced by many factors: it depends, for instance, on the ratio between the cache size and the sum of the sizes of all the objects in each set, but this dependency is not linear. In general, it is not possible to come up with a proper mathematical model to explain the sensitivity of the hit rate to the different system parameters. As such, rather than the absolute value, it is interesting to study the relative performance of the different schemes. For this reason, we use the exact same sequence of requests for each scheme. The same is true for the calcification: it is not possible to understand the impact of the parameters on the loss of the hit rate due to calcification, but we can observe that there is indeed calcification and how the schemes react to it.

We would like to stress the fact that, in the following, we present a set of *representative* results. During our experimental campaign, we perform several runs for the same experiment using different seeds to generate (i) the objects that populate the database, (ii) the object popularity and (iii) the order of the requests; in all the cases we have obtained always similar qualitative results. Therefore the observations made in this work are valid for a wide rage of configurations, not shown here for space constraints.

C. *Baseline results, with no calcification*

In the first set of experiments, we use a single object size distribution for all objects stored in the database (cf. Sect. IV-A). As anticipated, although object popularity varies over time according to the model described above, the popularity of the classes allocated by both Memcached and Twemcache remains constant. Therefore, slab calcification does not occur. Hence, the results we present here indicate the performance achieved by each system, which we use as a baseline.

Next, we consider four system configurations: Memcached, Memcached with the reset policy, Twemcache with the random eviction policy, and Twemcache with the slab LRA policy. Fig. 3 shows the results, in terms of hit rate over time. Note that we do not explicitly use time on the x-axis: as discussed in Sect. IV-A, request arrivals and the notion of time are intertwined. Hence, we display the percentage of client requests that arrive to the application server.

Fig. 3 indicates that, for Memcached, after an initial period necessary to fill the cache, the hit rate becomes stable at a value of roughly 84%. The system behaves as expected, i.e., for a given configuration, the hit rate is not influenced by object popularity, as long as class popularity is constant.

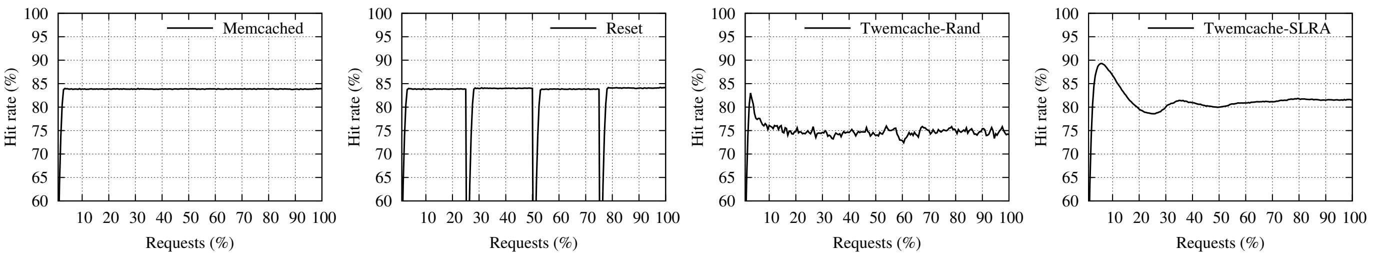


Fig. 3. Hit rate over time in the absence of calcification: the size of the requested objects has been drawn from a single distribution.

In case of the Reset policy, we impose a cache reset four times during the experiment. At each restart, there is a transitory period necessary to fill the cache, after which the hit rate reaches the stable value as for Memcached (with small variations of $\pm 0.25\%$). Note that each time we restart the cache, the arrival pattern of the requests is different from the previous period; the fact that the hit rate, after the transitory period, remains constant confirms that the hit rate is not affected by the request arrival pattern.

The random eviction policy in Twemcache achieves a lower, and *extremely variable* hit rate, when compared to vanilla Memcached. As we anticipated in Sect.III-B, 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 rate. Our experiments also show that the slab LRA policy in Twemcache, while it performs better than the random eviction, obtains a smaller hit rate than vanilla Memcached. Similarly to the random eviction policy, the slab LRA technique is counterproductive when slab calcification does not occur.

Next, we discuss in more detail our baseline results by inspecting the internal state of Memcached and Twemcache with the slab LRA policy. Fig. 4 illustrates the number, per class, of client requests, objects stored in the cache, and cache misses. Since the object size distribution per class grows exponentially, we use a logarithmic scale in our plots. Clearly, request distributions are equal for both Memcached and Twemcache. Note also that, in Memcached, both the number of object and miss per class are approximately proportional to the number of client requests.

We observe that Fig. 4 helps in understanding the ill behavior of Twemcache with the slab LRA policy. This policy, in fact, evicts the slabs based on their access times. Since slabs belonging to the higher classes have less objects (per slab), their access frequency will be smaller than slabs belonging to the lower classes, which store one or two order of magnitude more objects (per slab). Therefore slabs of the higher classes will be evicted more frequently than the other classes, and the overall effect is that the number of objects stored in each class is not proportional to the number of client requests for that class.

D. Results with calcification

We now turn our attention to the problem of slab calcification. As described in Sect. IV-A, the object space $\mathcal{Q}$ is partitioned in two equal sets, each with different parameters that determine the "shape" of object size distributions. Recall

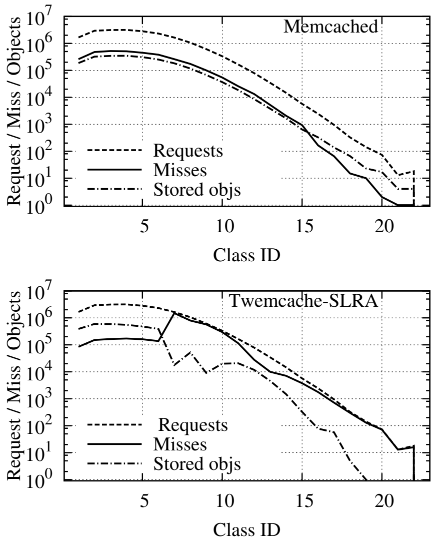


Fig. 4. Snapshot of the Requests, Misses, and Objects distribution as measured by Memcached, for single object size distribution.

that the two object sets have the same mean size: this has been done to establish a fair comparison of the performance achieved by client requests for the whole object space. As explained in Sect. IV-B, in fact, the hit rate is influenced by many factors: if we compare sets populated with objects of different mean sizes, then the hit rate would be affected by *both* the slab calcification problem and the different mean sizes. As such, our methodology isolates the problems induced by slab calcification from other "external" factors.

Our experiments are built as follows. We induce three phases in the request arrival pattern: in the first phase, the client sends requests for the first set of objects; in the second phase, objects of the second set are increasingly requested; in the third phase only objects of the second set are considered. Note that this methodology does not undermine the validity of Eqs. 1 and 2. Indeed, we chose the size of the first $\mathcal{R}/2$ objects as drawn from the first size distribution; the size of the remaining $\mathcal{R}/2$ objects is selected according to the second size distribution. The three phases discussed above are a direct consequence of the way we model object popularity dynamics (cf. Sect. IV-A). In summary, our experiments entail a variation in class popularity which, as we explained before, is the main culprit for the slab calcification problem.

Fig. 5 reports, similarly to what discussed in the baseline scenario without calcification, the variations of the cache hit

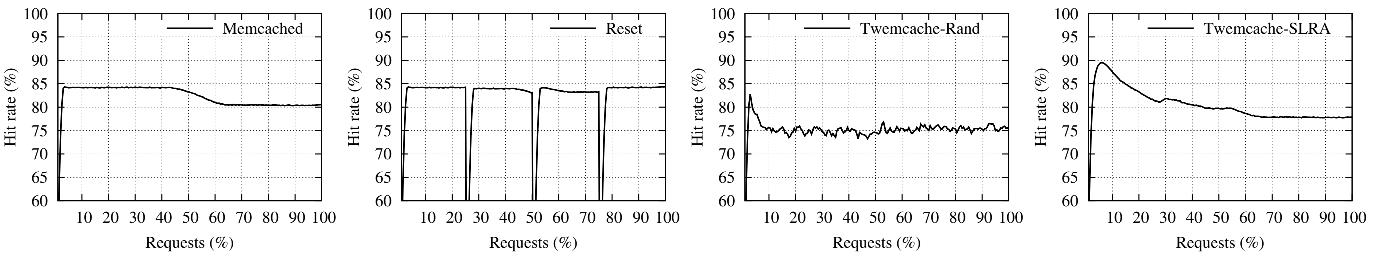


Fig. 5. Hit rate over time with calcification: the size of the requested objects has been drawn from two different distributions.

rate over time/requests, for the four system configurations described above.

With vanilla Memcached, the impact of slab calcification on the hit rate is evident, with a loss of 4%. As the client asks for more and more objects with sizes that have been drawn from different distributions, the hit rate decreases progressively. The graph represents exactly what would happen to the hit rate in case of a change of the object size distribution, and how much would be the loss.

The Reset policy mitigates the effects of slab calcification. During the transition among object sets (the three phases described above) the hit rate 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 rate to a similar value to that of the first phase. In practice, Memcached works as if there were no change in the statistical properties of the object size distributions, and adapt to the new request patterns. Clearly, each reset action provokes a transitory phase in which the cache is filled, which affects the achieved hit rate. From the practical standpoint, the Reset policy requires an automatic method to determine reset events, for otherwise a manual or periodic intervention would be required.

Our results indicate that Twemcache with random eviction is not affected by the slab calcification problem, since its behavior is equivalent to that observed in Fig. 3. As for Twemcache with slab LRA policy, our experiments highlight that calcification has a negative impact on the hit rate. The reasons underlying these result are elusive and require more experiments on Twemcache alone, a study that falls outside the scope of this work. In any case, the two Twemcache eviction variants under-perform vanilla Memcached (with calcification) and may be unstable.

We now inspect, similarly to the baseline case, the internal state of each system configuration we studied: Fig. 6 compare the number of requests, misses and stored objects for Memcached and Twemcache with the random eviction policy. Note that this graph represents the internal state of both systems in the last phase of the experiment.

First, we note that the number of requests across different classes has a different shape with respect to what shown in Fig. 4, because the object size distribution is different. Now, in vanilla Memcached the number of stored objects is not proportional to the requests, because the memory partitioning is that obtained during the first phase of the experiment and cannot adapt to any changes in object size distribution. With the random eviction policy in Twemcache, instead, and the number of stored objects is proportional to the requests, which

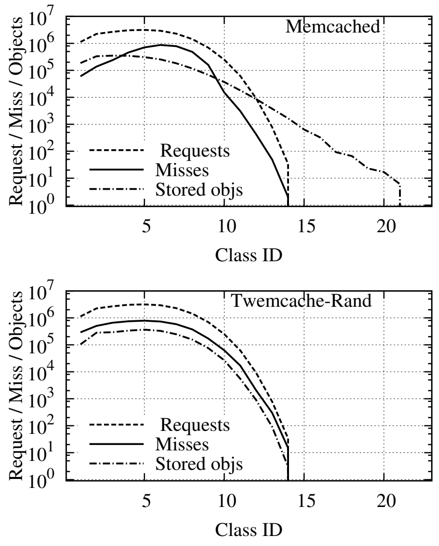


Fig. 6. Snapshot of the Requests, Misses, and Objects distribution as measured by Memcached, for two object size distributions.

essentially solves the calcification problem. However, the hit rate achieved in this case is low (i.e. the number of misses is high) because of the aggressive slab eviction mechanism.

V. A New Memory Partitioning Mechanism

Memcached partitions the memory proportionally to the number of requests to each class. Besides the calcification problem, we now study whether such a partitioning mechanism can be re-factored, to achieve both a higher hit rate in general, and to avoid the slab calcification problem. Next, we describe in detail an alternative memory partitioning scheme, that we call *Periodic Slab Allocation* (PSA). We have implemented and integrated PSA in Memcached: as such, we also show experimental results on the performance of PSA following the same methodology described in Sect. IV.

A. Periodic Slab Allocation (PSA) Policy

Illustrative example: We begin with an intuitive description of PSA, using an example. By design, Memcached partitions the memory into different classes. For example, let's consider Fig. 4 (top): classes whose ID is grater than or equal to 15 receive at most 10^4 requests, while classes whose ID is smaller than or equal to 8 receive at least 10^6 requests.

Clearly, the contribution to the aggregate hit rate of each class is different. If we continue the illustrative example above, evicting all objects from a slab assigned to a high-ID class incurs in an inflation in the number of misses for such objects that is bounded by the maximum number of requests, i.e., 10^4. On the other hand, the number of misses of low-ID classes is at least 10^5, therefore there are more chances, should the memory be reassigned, to decrease the number of misses.

Intuitively, a slab allocation mechanism that strives at increasing the hit rate in Memcached should operate as outlined above. The main problem to solve is to determine the candidate classes to reassign slabs: if the reduction in the number of misses for one class is higher than the increase in the number of misses for another class, then slab reallocation contributes to a higher hit rate overall.

PSA description: Algorithm 1 illustrates the most important steps of PSA, which is driven by the number of misses incurred by Memcached. Slab allocation is executed every time the cache collects M misses; we call the interval of time between two of such events a *round*. PSA runs in an individual thread and uses the internal statistics collected by Memcached to inform slab allocation: the total number of misses M, the number of misses per class $\mathbf{m}$, the number of requests per class $\mathbf{r}$, and the number of slabs allocated to each class $\mathbf{s}$. Note that $\mathbf{r}$ and $\mathbf{m}$ are recomputed every round; clearly, $\sum_i m_i = M$ holds. At each round, PSA "moves" a single slab 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 number of requests and the number of slabs allocated to the class, r_i/s_i. In other words, "moving" one slab from this class to another one increases the number of misses, as a first approximation, by a value equal to r_i/s_i. While more sophisticated measures can be used to estimate the variation in the number of misses when slabs are removed, our approach is computationally simple (with negligible burden on the system) and yet our measurements have shown that 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.[6]

The selected slab is assigned to the class that has the maximum number of misses. Computing such value requires some attention: indeed, new objects can be requested by clients, which contribute to unavoidable misses (cf. Sect. III-A). We estimate the number of new objects that enter the cache in a round as the minimum value of the ratio m_i/r_i, computed for each class i. In fact, a class with more slabs than required would only store new objects, therefore, among all classes, the minimum value of m_i/r_i represents an upper bound to the probability that new objects have been added to the object set during the round. The class with the maximum number of *net* misses (misses without the new objects) receives the slab. Once slab allocation completes, a LRU-based eviction policy within each class ensures an efficient memory utilization, until the next round.

Note that PSA considers the number of requests per class, not the size of the objects in such classes: PSA aims at finding a working point where a change in the memory partitioning does not increase the hit rate. In summary, PSA can be thought of as a mechanism that caters to a high hit rate by *adapting how memory is partitioned* to mirror both object popularity dynamics and variations in object size distribution. As a consequence, although not designed to explicitly address it, PSA is an effective countermeasure to the calcification problem we discuss in Sect. III. Our experimental results, that we show next, substantiate this claim.

[6]In Twemcache, the slab LRA policy is applied across classes, and thus is global, not local to a class as in our approach.

Algorithm 1 Periodic Slab Allocation (PSA)

```
1. Input: $\mathbf{s}$ // vector of slabs allocated to each class
2. Input: $\mathbf{r}$ // vector of requests in each class
3. Input: $\mathbf{m}$ // vector of misses in each class
4.
5. Every $M$ misses do
6.     $p_{\text{new}} \leftarrow \min(m_i/r_i)$;
7.     $id_{\text{take}} \leftarrow i : (r_i/s_i) < (r_j/s_j), \forall\ r_j, s_j \in \mathbf{r}, \mathbf{s}$;
8.     $id_{\text{give}} \leftarrow i : (m_i - p_{\text{new}} r_i) > (m_j - p_{\text{new}} r_j), \forall\ r_j, m_j \in \mathbf{r}, \mathbf{m}$;
9.     MoveOneSlab($id_{\text{take}}, id_{\text{give}}$);
```

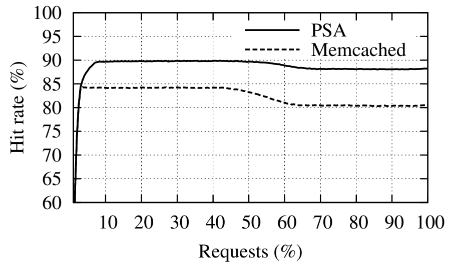
Complexity: PSA is a lightweight algorithm, in line with what is currently implemented in Memcached and Twemcache. To operate, PSA locks the memory in two occasions. First, to compute, given the internal status of Memcached, the statistics $(\mathbf{r}, \mathbf{m}, \mathbf{s})$. This takes $O(C)$, where 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 $O(\text{items})$.

Recall also that, PSA is executed every M misses, meaning that a round has no fixed duration: the number of client requests $R = M/\text{hit\_rate}$ determines when memory allocation has to be done. In Twemcache (with both S-LRA and random eviction policies), a slab is removed and assigned to a new class when there is no free chunks for that class, which happens approximately every c_i misses, where c_i is the number of chunks contained in a slab of class i – in our experimental setup, $c_i = 1, \ldots, 13796$. We conclude that, when M is in the range $[5'000; 20'000]$, PSA should not impose a high toll on system resources in an operational setting, and our experiments are in line with this expectation.

Additional considerations: In this work, we implicitly assume that each object contributes equally to the hit rate (as it is commonly supposed). In case of a miss, the cost for retrieving an object form the back-end is not usually considered. However, some objects may be more costly to retrieve from the database than other: such "cost" could influence eviction decisions. In the literature, there are a number of examples which consider object cost to be related to the complexity of the database query to generate the object. These works [10], [11], [17], albeit in the context of Web caches, describe approaches based this concept. Although, currently, Memcached does not support object cost, an extension in such a direction (e.g. building on the work by Cao *et. al.* in [11]) would be simple. If costs were available, PSA could be readily modified to gauge memory partitioning accordingly.

B. Experimental Results

Using our implementation of PSA for Memcached, we now present experimental results following the same methodology presented in Sect. IV. Specifically, we execute the second campaign of experiments, in which clients request for objects with two different size distributions (cf., Sect. IV-D). Recall that, in such experiments, we identify three phases: in the first phase, object sizes are drawn from the same (initial) distribution; in the second phase clients request objects with size drawn from both distributions; in the last phase, object sizes are drawn from the same (final) distribution. If not otherwise stated, we have set M equal to $10'000$.



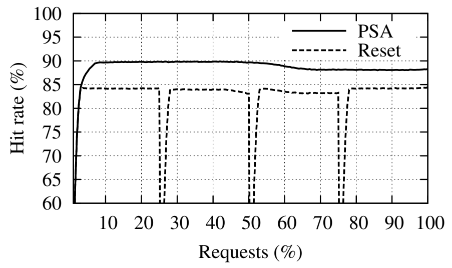
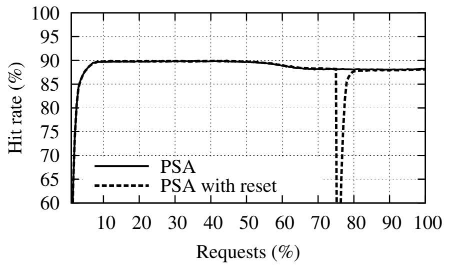


Fig. 7. Hit rate over time with calcification due to object popularity and size dynamics: comparison between PSA and other eviction policies.

Fig. 7 shows how the hit rate achieved by PSA compares to vanilla Memcached and Memcached with Reset. The PSA mechanism exhibit a higher hit rate in all the three phases of the experiment: the memory partitioning built by our scheme produces a larger number of hits, even in the absence of calcification. We also note that PSA adapts memory allocation according to request, popularity and size distribution dynamics. Note also that the hit rate 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. To verify this, we run an additional experiment where we impose an artificial reset to the PSA-based Memcached server: with the reset, we make sure that memory partitioning is "molded" according to the final object size distribution, following client requests. Fig. 8 (top) illustrates the evolution of the hit rate for PSA and PSA with reset: since both hit rates converge to the same value, we conclude that calcification is not the cause for a lower absolute value. In the last phase of the experiment, object size distribution is more compact than that used in the initial phase: the means are the same, but the variance is three times smaller, which may lead to lower hit rate. As we said in Sect. IV-B, the hit rate is sensitive to many factors, and coming up with a proper mathematical model to explain such sensitivity is outside the scope of this work.

Next, we study the impact of the only parameter of the PSA mechanism, namely M, which triggers memory allocation. Fig. 8 (bottom) indicated that such parameter has a rather small impact on PSA behavior: essentially, M determines the time it takes for the hit rate to reach its maximum value when the cache is empty.

We by discussing additional details on PSA: similarly to what we have done for the baseline experiments, we now show snapshots of Memcached internal state, including the number of client requests, misses and the stored objects for each class.



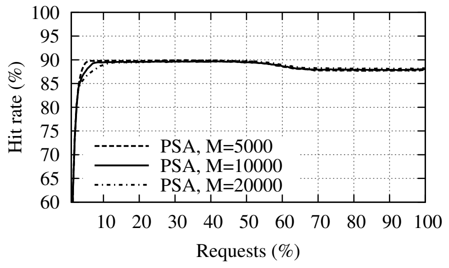


Fig. 8. Hit rate over time (dynamic popularity): PSA with reset (top) and with multiple values of M (bottom).

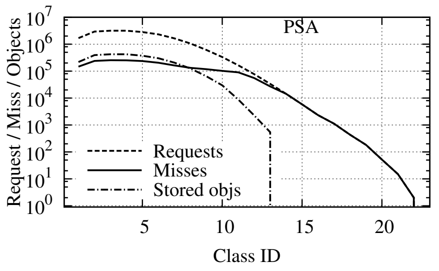


Fig. 9. Snapshot of the Requests, Misses, and Objects distribution for the first 10% of client requests.

Fig. 9 illustrates the internal state after 10% of client request have been issued, for the initial object size distribution. While Memcached allocates memory proportionally to the number of requests, the allocation of PSA aims at minimizing the number of misses. When clients request for objects with a different size distribution (the third phase of the experiment), PSA adapts how memory is partitioned to achieve as few misses as possible, as shown in Fig. 10.

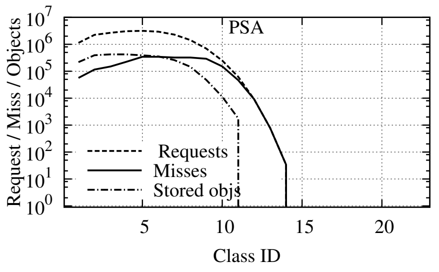


Fig. 10. Snapshot of the Requests, Misses, and Stored Objects distribution in the third phase of the experiment.

VI. Discussion

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: to do so, object popularity, and its dynamics, need to be appropriately crafted. For experimental reproducibility, a clear specification of such parameters is key, in conjunction to measurement studies to inform the design of realistic distribution shapes – a methodology we adopt in this work, building on the information discussed by Atikoglu *et. al.* in [6].

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: this is usually referred to as benchmarking suites. With such tools, it is possible to reproduce exactly the same experimental conditions used to study system performance with little effort, 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 [18].

VII. Conclusions

In-memory key-value stores are increasingly used by large-scale Web applications, as they relieve back-ends from supporting the load to generate hot, popular items requested by clients. In this paper we have considered Memcached, a popular in-memory key-value store that is used as a simple caching layer in many real-life deployments, including Facebook, Twitter and other large-scale setups. Despite its popularity, Memcached has received little attention from the academic community in the past. Only recently, for instance, realistic data concerning its usage has been disclosed, and issues that affect its performance have been discussed, albeit informally, in technical notes.

In this work, we focused on an important component of Memcached, which governs how memory is partitioned to accommodate object to be stored. Using an experimental testbed, we have shown that vanilla Memcached suffers from a static memory partitioning, which is usually referred to as calcification. While calcification has been discussed and cited in technical blogs [4] and some papers [6], [8], we have shown, to the best of our knowledge for the first time, its impact on the hit rate. 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 rate.

The analysis of the calcification problem has revealed the need for a new approach to memory partitioning altogether, aiming at achieving as high hit rates as possible, while adapting to dynamics in client requests, object popularity and characteristics. Our design materialized in a mechanism we called *periodic slab allocation* (PSA).

Using a realistic experimental setup, we showed that PSA outperforms existing mechanisms both in absolute terms – higher hit rate – and in that it eliminates the problem of calcification. Along the way, we also proposed a trace-based methodology for academics and practitioners to study and compare alternative memory partitioning mechanisms. Our future work will be devoted to refine the estimation of the number of misses as the memory assigned to a class varies, in order to investigate if it would be possible to further increase the performance of our scheme.

References

[1] (2013) Memcached. [Online]. Available: http://memcached.org/

[2] 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.

[3] 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.

[4] (2013) Caching with twemcache and calcification. [Online]. Available: http://engineering.twitter.com/2012/07/caching-with-twemcache.html

[5] (2013) Twemcache. [Online]. Available: https://github.com/twitter/twemcache

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

[8] 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.

[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] D. Starobinski and D. Tse, "Probabilistic methods for web caching," *Performance Evaluation*, vol. 46, no. 2-3, pp. 125–137, 2001.

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

[12] 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.

[13] 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*, vol. 36, no. SI, pp. 315–327, 2002.

[14] G. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *The Journal of Supercomputing*, vol. 28, no. 1, pp. 7–26, April 2004.

[15] D. Thiebaut, H. Stone, and J. Wolf, "Improving disk cache hit-ratios through cache partitioning," *IEEE Transactions on Computers*, vol. 41, no. 6, pp. 665 –676, jun 1992.

[16] 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.

[17] 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.

[18] (2013) Benchmarks for testing memcached memory management. [Online]. Available: http://profs.sci.univr.it/∼carra/mctools/