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 – 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 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.**

Using an experimental approach, we show the negative impact of calcification on an important performance metric, 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. By decreasing the delay of the retrieved objects, it is possible to improve the Quality of Experience (QoE) of end-users. To achieve such goal, a common solution is to keep a large fraction of the data (objects) served by a website in main memory. In this context, Memcached (MC) [1] 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 companies, such as Wikipedia, Flickr, Digg, WordPress.com, Craigslist, and, with additional customizations, Facebook and Twitter.

In this work we focus on *memory partitioning*, which is the process by which MC assigns portions of the memory to store objects.

By design, MC 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 has to be stored, MC checks if there is available space in the appropriate class, and stores it. If there is no space, MC *evicts* a previously stored object in favor of the new one.

In MC, therefore, 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 referred to as *calcification* – a problem observed in some prominent operational setups [2]–[4].

The straightforward solution to calcification is to allow MC to reassign memory previously allocated to a given class; however, 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 revisit the *overall process of memory allocation*. Despite the clear consequences on hit rate, this problem has received little attention in the literature.

Our contributions: Our first goal is to understand the impact of calcification on MC performance. To do so, we use the well-known *hit rate* metric, defined as the number of requests that can be served directly from memory (the cache) over the total number of requests received by a Web server. Since the hit rate has a direct impact on the QoE, the evaluation of the hit rate under different working conditions is of fundamental importance.

Using an experimental testbed, we show and determine – for the first time – how calcification adversely impact the hit rate (Sec. III). In our experiments, we use the latest version of MC and Twemcache (TMC) – 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.* [4], which is based on production-traces of MC at Facebook. We then set off to design a new memory allocation scheme to replace that implemented in MC: 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

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

new memory allocation mechanism – which is computationally efficient and lightweight – and compare its performance to both MC and TMC (Sec. IV).

Our results indicate that the memory allocation mechanisms of both Memcached and Twemcache are far from being optimal. With our scheme, we obtain superior hit rates both in absence of and with calcification, which underlines the importance of memory allocation in general.

II. Memcached and Calcification

Memcached (MC) is a key-value store that keeps data in memory, i.e., data is not persistent. Clients communicate with MC through a simple set of APIs: Set, Add, Replace to store data, Get or Remove to retrieve or remove data. MC has been designed to simplify memory management [5] and to be extremely fast: since every operation requires memory locking (note that memory locking is required even in case of a Get, since access time statistics need to be updated), data structures must be simple and their access time should be kept as small as possible.

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

The total available memory to MC 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, MC 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), MC 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, MC 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 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 rate is 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 [3], 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 MC. 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.

Memcached Automove: a recent version of MC 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 Policies: Twemcache (TMC) [3] accommodates a set of eviction policies explicitly designed to solve the slab calcification problem. With the *Random* eviction policy, 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. With the *Least Recently Accessed Slab* eviction policy, instead of a random slab, the server chooses the least recently accessed slab. Both policies allow 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 that both policies may be too aggressive. Our experiments confirm such belief.

III. Experiments

We now study the performance of MC and TMC in terms of hit rate. Our experiments aim at showing the impact of calcification, as well as the effectiveness of current schemes available in the literature.

A. Experimental Setup

In scale-out Web application, a series of MC servers are commonly configured in a shared-nothing setup, whereby each server takes care of a subset of data objects using consistent hashing [4]. This means that each MC server receives requests for objects that have approximately the same statistical properties. Therefore, to study slab calcification, it is sufficient to measure the performance of a single MC server. As for the request arrival to the server, MC locks the memory at each operation: even if requests are managed by many threads (used to maintain open connections, process the requests and prepare the responses), from the memory viewpoint, these requests are processed in series; hence, generating the requests from a single or from multiple clients has little or no impact on memory management.

Following the above observations, in our experiments we deploy a simple, yet representative, Web architecture that is illustrated in Fig. 1. An application server is connected to a database and to a MC 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

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

object is in the cache; if MC 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 MC.

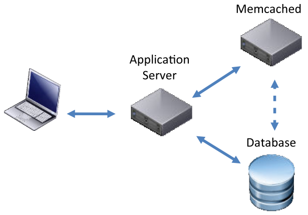


Fig. 1. An illustration of the testbed used in our experiments: this is a simple, yet representative, configuration.

The database is populated with two sets of objects. 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 [4]. 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 [2]–[4], 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 (not shown here for space constraints), obtaining always the same qualitative results presented in the next sections.

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. V we provide additional details about this.

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[3] 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 rate 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 MC 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 the following, we also report a number of internal statistics collected automatically by both MC and TMC. 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. The application server is instructed to take such snapshot every 20 Million requests.

As final consideration, we note that 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: a simple mathematical model to explain the sensitivity of the hit rate to the different system parameters remains elusive. As such, rather than the absolute value, it is interesting to study the relative performance of the different approaches we analyze: thus, in our experiments, we use the exact same sequence of requests for each scheme. Next, we present a set of *representative* results: we perform different runs for the same experiment using different seeds to generate the objects that populate the database and the requests; in all such cases, we consistently obtain similar results.

B. Results

We consider four system configurations: MC, MC with the reset policy, TMC with the random eviction policy, and TMC with the slab LRA policy[4]. Fig. 2 shows the results, in terms of hit rate over time. Note that we do not explicitly use time on the x-axis: request arrivals and the notion of time are intertwined, and we display the percentage of client requests that arrive to the application server.

Fig. 2(a) indicates that, for MC, after an initial period necessary to fill the cache, in the first phase the hit rate becomes stable at a value of roughly 84%. In the second and third phase, 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 a different distribution, the hit rate decreases progressively.

With the Reset policy, shown in Fig. 2(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 rate is affected by different object size distributions: this is clearly visible in the third "wave." However, once the transition is over, MC can restore the hit rate to a similar value to that of the first phase. Clearly, each reset action provokes a transitory period to fill the cache, which affects negatively the achieved hit rate.

Fig. 2(c) shows that the random eviction policy in TMC achieves a lower, and *extremely variable* hit rate, when compared to MC. As we anticipated in Sect.II, the eviction of randomly selected slabs may be too aggressive, because an individual slab may contain many popular items. As such, using TMC in conjunction with the random eviction policy has a negative impact on the hit rate overall. Our experiments show that also the slab LRA policy in TMC obtains a smaller hit rate than MC in the long run, albeit performing better than random eviction. The reasons underlying these result are elusive and require more experiments on TMC alone, a study that falls outside the scope of this work. In any case, the two TMC

[3] 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 [6]).

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

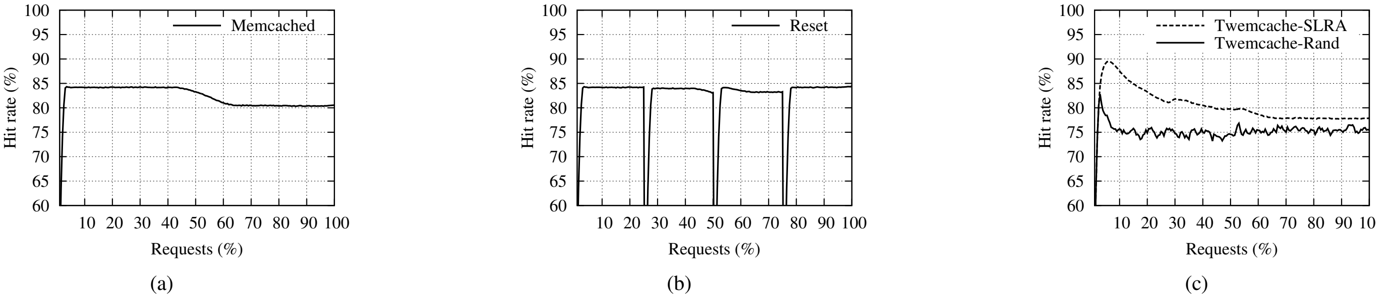


Fig. 2. Hit rate over time with calcification obtained by different schemes.

eviction variants under-perform MC (with calcification) and may be unstable.

Next, we discuss in more detail our baseline results by inspecting the internal state of Memcached. Fig. 3 illustrates the number, per class, of client requests, objects stored in the cache, and cache misses. Fig. 3(a) represents the internal state in the first phase of the experiment, while Fig. 3(b) considers the last phase – the number of requests across different classes has a different shape with respect to what shown in Fig. 3(a), because the object size distribution is different.

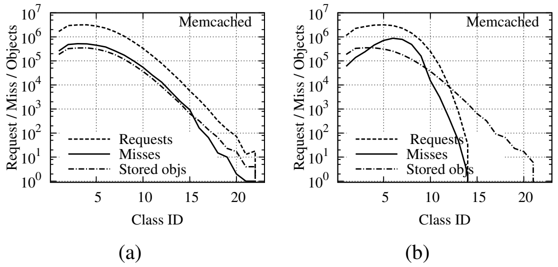


Fig. 3. Requests, Misses, and Objects distribution, in the first (a) and in the last phase (b).

In the first phase, the number of object and miss per class are approximately proportional to the number of client requests. In the last phase, instead, 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.

IV. A New Memory Partitioning Mechanism For Memcached

As observed previously, MC 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 slab calcification. Next, we describe our approach, called *Periodic Slab Allocation* (PSA).

We begin with an intuitive description of PSA: by design, MC partitions the memory into different classes. For example, let's consider Fig. 3 (a): classes whose ID is greater 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. 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 them, to decrease the number of misses. 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 might contribute to a higher hit rate overall.

Algorithm 1 illustrates the most important steps of PSA, which is driven by the number of misses incurred by MC. 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 MC 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: "moving" one slab from one class to another, 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 measurements have shown that the approach we propose is fairly accurate. If the class with the lowest risk has more than one slab, the slab reassignment follows a Least Recently Accessed (LRA) approach within the class.[5] Once slab allocation completes, LRU-based eviction within each class ensures an efficient memory utilization, until the next round.

Note that PSA considers the number of requests per class and per slab: PSA aims at finding a working point where a change in the memory partitioning does not increase the miss 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

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

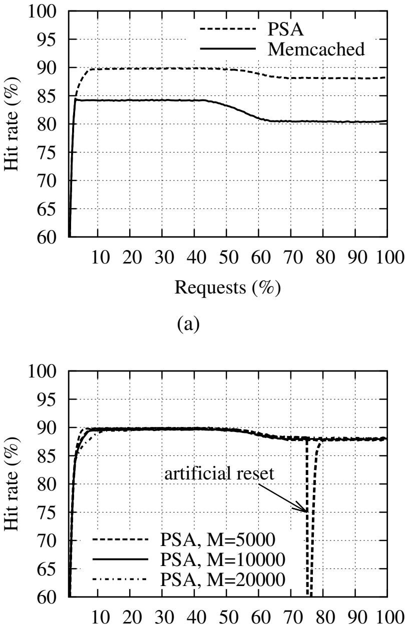
Algorithm 1 Periodic Slab Allocation (PSA)

```
Input: $\mathbf{s}$ // vector of slabs allocated to each class
Input: $\mathbf{r}$ // vector of requests in each class
Input: $\mathbf{m}$ // vector of misses in each class

Every $M$ misses do
    $id_{\text{take}} \leftarrow i : (r_i/s_i) < (r_j/s_j), \forall\ r_j, s_j \in \mathbf{r}, \mathbf{s}$;
    $id_{\text{give}} \leftarrow i : m_i > m_j, \forall\ m_j \in \mathbf{m}$;
    MoveOneSlab$(id_{\text{take}}, id_{\text{give}})$;
```

to the calcification problem. In terms of complexity, PSA is comparable to what is currently implemented in MC and TMC – the rate at which slabs are moved among classes is on par to that of TMC.

We have implemented and integrated PSA in MC and evaluate its performance following the same methodology described in Sect. III. Fig. 4(a) shows how the hit rate achieved by PSA compares to that of MC, in each experiment phase. With no calcification (first phase), PSA achieves a 7% increase over vanilla MC; in presence of calcification (third phase), the gain in favor of PSA reach 10%. Note 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 MC server: with the reset, we make sure that memory partitioning is "molded" according to the final object size distribution, following client requests. Fig. 4(b) shows that, after the reset, the hit rate converges to its previous value. Fig. 4(b) shows also the impact of the only parameter of the PSA mechanism, namely M. The figure indicates that the impact of M on PSA behavior is small.



(a)

(b)

Fig. 4. Hit rate over time obtained by PSA compared with vanilla MC (a) and by varying the parameter M (b).

V. 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 (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 [4].

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 [7].

VI. Related Work

The analysis of cache performance has been the subject of many past studies. In this paper we consider specifically MC, therefore we first focus on the literature about such system. Even if MC is widely used, the study of its performance has received only little attention. Atikoglu *et. al.* provide in [4] a set of measurement results from a production site – in our experiments we use these statistics to generate our workload. However, the paper does not analyze the different eviction policies, and it does not consider the impact of memory partitioning on the hit rate.

Gunther *et. al.* [8] highlight that MC has scalability issues, since threads access the same memory, and locks prevent the exploitation of the parallelism. For this reason, a number of works [5], [9] consider the throughput of MC, 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 rate as we do. Nishtala *et. al.* [10] study scalability problems, i.e., how to manage a multi-server architecture, but they do not study the eviction policies and memory partitioning.

Overall, the literature on caching mechanisms is vast: CPU [11], browser [12], Web [13], and DNS caches [14], as well as Content Delivery Networks [15] and P2P networks [16]–[20] 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

[21], which has a negative impact on performance. Similarly, in MC, 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 [21]–[23] are based on a common idea, in which the memory partitioning process tries to balance the number of misses among the processes.

Finally, we note that in many caching mechanisms (e.g., Web, DNS), memory partitioning is generally not considered to be problematic: previous works [12], [13], [24] mainly focus on the how eviction policies manage objects with different sizes. In MC, instead, eviction is done on a per-class basis, and objects within a class have the same size.

VII. Conclusion and Perspectives

In-memory key-value stores, such as Memcached (MC), are increasingly used by large-scale Web applications: they cater short response times and small delays, which are fundamental to achieve improved quality of experience for end-users. In this work, we studied an important aspect of MC, memory allocation, and measured its impact on a key performance metric, the cache hit rate. Using an experimental testbed, we have shown that MC suffers from a static memory partitioning, which is usually referred to as calcification. While calcification has been discussed and cited in technical blogs [2] and some papers [4], [10], we have shown, to the best of our knowledge for the first time, its impact on the hit rate. We have also studied TMC, a variant conceived at Twitter that includes eviction policies to address calcification, and showed that the price TMC 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 new mechanism, which we called PSA, that produced higher hit rates both in absence of and with calcification.

In concluding, we remark that in-memory caches also relieve back-ends from supporting the load to generate hot, popular items requested by clients. Thus, instead of a client perspective, which we have considered in this work, it would be interesting to examine an alternative metric accounting for the "pressure" on the back-end: the *byte hit rate* – the hit rate weighted by the size of the objects. To this end, we consider a promising research direction the general idea of assigning a "cost" to each object: for instance, some objects may be more costly to retrieve from the database than others due to their sizes. In the literature, there are a number of examples [12], [13], [24] which consider object cost to be related to the complexity of the database query to generate the object, and not their size. Currently, PSA does not support object cost: an extension in such a direction (e.g. building on the work by Cao *et. al.* in [13]) would require a number of modifications which we plan to address as part of our future work.

References

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

[2] (2013) Caching with twemcache and calcification. [Online]. Available: http://engineering.twitter.com/2012/07/caching-with-twemcache.html

[3] (2013) Twemcache. [Online]. Available: https://github.com/twitter/twemcache

[4] 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.

[5] 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.

[6] D. Carra and P. Michiardi, "Memory partitioning in memcached: An experimental performance analysis," Department of Computer Science, University of Verona, Tech. Rep., June 2013. [Online]. Available: http://profs.sci.univr.it/∼carra/downloads/TR-UNIVR-Car-2013-01.pdf

[7] (2013) Benchmarks for testing memcached memory management. [Online]. Available: http://profs.sci.univr.it/∼carra/mctools/

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

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] E. Biersack, D. Carra, R. L. Cigno, P. Rodriguez, and P. Felber, "Overlay architectures for file distribution: Fundamental performance analysis for homogeneous and heterogeneous cases," *Computer Networks*, vol. 51, no. 3, pp. 901–917, 2007.

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

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

[19] D. Carra, R. L. Cigno, and E. Biersack, "Stochastic graph processes for performance evaluation of content delivery applications in overlay networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 19, no. 2, pp. 247–261, 2008.

[20] P. Michiardi, D. Carra, F. Albanese, and A. Bestavros, "Peer-assisted content distribution on a budget," *Computer Networks*, vol. 55, no. 7, pp. 2038–2048, 2012.

[21] 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.

[22] 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.

[23] 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.

[24] 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.