Elastic Provisioning of Cloud Caches: a Cost-aware TTL Approach

Damiano Carra*, Giovanni Neglia† and Pietro Michiardi‡
*University of Verona, damiano.carra@univr.it
† Inria, Université Côte d'Azur, giovanni.neglia@inria.fr
§Eurecom, pietro.michiardi@eurecom.fr

***Abstract*—We consider elastic resource provisioning in the cloud, focusing on in-memory key-value stores used as caches. Our goal is to dynamically scale resources to the traffic pattern minimizing the overall cost, which includes not only the storage cost, but also the cost due to misses. In fact, a small variation of the cache miss ratio may have a significant impact on user perceived performance in modern web services, which in turn has an impact on the overall revenues for the content provider using such services.**

We propose and study a dynamic algorithm for TTL caches, which is able to obtain close-to-minimal costs. Since high-throughput caches require low complexity operations, we discuss a practical implementation of such a scheme requiring constant overhead per request independently from the cache size. We evaluate our solution with real-world traces collected from Akamai, and show that the TTL approach is able to track the optimal cache configuration and achieve significant cost savings specially in highly dynamic settings that are likely to require elastic cloud services.

I. Introduction

In-memory key-value stores used as caches are a fundamental building block for a variety of services, including web services and Content Delivery Networks (CDN). With the advent of cloud computing, these services, including the caches, have been offered as managed platforms with a pay-as-you-go model. Amazon's ElastiCache [1], Microsoft's Azure Redis Cache [2] and Google's Cloud Memorystore [3] are examples of caches that employ popular open source software such as Memcached [4] or Redis [5].

Elasticity, *i.e.*, the ability to adapt to workload changes, is a key characteristic of cloud computing: auto-scaling tools, configured by the users, determine the amount of cloud resources to deploy. The techniques used to drive the scaling process have been the subject of many studies in the past—see [6] and the references therein. These studies mainly focus on traditional services, such as computing, where the relation between the performance and the amount of deployed resources is almost linear.

When considering the caches, the relation between a key performance index, the hit ratio, and the resources deployed is not linear, *e.g.*, doubling the cache size does not correspond to doubling the hit ratio. Unfortunately, the analysis of dynamic adaptation of cloud caches has received little attention in the literature: existing studies have mainly focused on minimizing storage costs for a given target hit ratio, ignoring that misses may have different costs and disregarding the possibility to tune the hit ratio itself.

Several studies have highlighted the cost of delay for web services [7], *i.e.*, a direct connection between the response time (or web page load time) and economic losses, for example because the customer does not finalize a purchase. Notice that, even a small increase in the miss ratio (*e.g.*, 1%), often translates into a high variation in the average latency (*e.g.*, 25%) [8]. Misses can also translate to infrastructure costs because of the additional load on back-end databases or content servers [9], [10]. Beyond these specific examples, in this paper we assume that it is possible to quantify the *cost* due to misses. Then, when sizing cache resource allocation, these costs should be considered.

In this paper we study the dynamic assignment of resources to in-memory data stores used as caches. To this aim, we take into account the cost of the storage *and* the cost of the misses, and we adapt the amount of resources to the traffic pattern minimizing the total cost. We consider an approach based on Time-To-Live (TTL) caches [11], and we study a model in which the TTL is adapted through stochastic approximation iterations and dynamically converges to the best setting. We operate the system using a virtual TTL cache, whose virtual size informs the elastic deployment of fixed-size cache server instances to manage incoming requests.

High-throughput caches rely on low complexity operations: for instance, key lookup and update in LRU caches have $\mathcal{O}(1)$ complexity per request. This bound is considered a hard requirement for CDNs running on commodity hardware [12]. The auto-scaling tool, therefore, should not have higher complexity, otherwise it may represent a performance bottleneck. For this reason, we design a practical policy to automatically scale-out caches with $\mathcal{O}(1)$ complexity per request.

We evaluate such TTL-based solution with a testbed, using real-world traces collected from Akamai Content Delivery Network over 30 days. We show that our approach can achieve the same savings obtained by adapting previously proposed solutions based on Miss Ratio Curves (MRCs) [13], which are less scalable because they have a per-request computational overhead that grows logarithmically with the cache size.

Contributions: We make the following contributions.

- *TTL-based approach:* We propose and study a dynamic algorithm for TTL caches, which adapts the TTL value to both misses and storage costs minimizing the total operational expenditure under the Independent Reference Model for the request process.

- *Design and implementation of a horizontally scalable TTL-based solution:* We design and implement a system based on the TTL approach, which dynamically adds and removes fixed-size cache instances in order to maintain the total cost at minimum. We pay particular attention to system scalability, and provide a $\mathcal{O}(1)$ solution targeted at high-throughput caches.
- *Evaluation:* We evaluate the TTL-based solution in our testbed with real-world traces from Akamai, and show that it is able to track the optimal cache configuration and achieve significantly cost savings specially in highly dynamic settings.

Roadmap: We provide some background in Sect. II and define the problem in Sect. III. We present the general framework of a TTL-based solution in Sect. IV, and discuss its practical implementation in Sect. V. We evaluate our approach in Sect. VI. Sect. VII discusses the related works, and we conclude the paper in Sect. VIII.

II. Background

A. In-memory data stores

In-memory key-value stores represent a fundamental piece of web architectures. They are used to cache popular contents, so that the web application can access quickly the frequently requested data, while the back-end database contains the original copy of all the contents. For instance, Facebook heavily relies on caches based on in-memory data stores, and organize them hierarchically in order to store and access a complex set of contents [14].

Popular in-memory stores are Memcached [4] and Redis [5]. While Redis contains a richer set of APIs, when used as a cache, it shares with Memcached some basic commands, such as setting a key-value entry, or retrieving the value given a key. If the cache is full and a new content needs to be inserted, both systems employ slight variations of the Least Recently Used policy (LRU). In particular, Memcached organizes the content into classes of objects with similar sizes, and performs LRU within each class. Redis picks randomly 5 objects and evicts the one least recently accessed; if the available space is not sufficient, it repeats the process.

The amount of RAM assigned to Memcached or Redis instances is set when the instance is created, and it cannot be changed at runtime. In order to achieve vertical scalability—*i.e.*, changing the amount of memory at runtime—the only option is to create a new instance with the desired amount of memory and transfer the content from the old instance to the new one. Since this approach requires time and resources, vertical scalability is usually not considered practical.

On the other hand, horizontal scalability is easy to achieve. Instances can be added to (or removed from) a cluster of nodes, with a *load balancer* tool (such as `mcrouter` [15]) that manages all the aspects related to the distributed caches: data placement and request routing, possibly also data replication and instance failure management. Data placement and request routing may use consistent hashing to map keys and servers to points on the hash space, and key responsibility is assigned to the closest server in the hash space.

From the performance point of view, if there are W instances, key lookup (*i.e.*, finding the node responsible for a key) takes $\mathcal{O}(\log W)$. For this reason, alternative schemes use a two-step approach [16]: the hash space is divided into $B \gg W$ small intervals, and intervals are randomly assigned to the W instances (a map maintains the association between intervals and instances). For the key lookup, the system checks in which interval the key falls (hash of the key, modulo B), and it reads from the map which server is responsible for that interval. Overall, the key lookup takes $\mathcal{O}(1)$, therefore it is independent from the number of instances.

In this paper, we consider the basic scenario where the content is not replicated across instances and one load balancer is sufficient for managing the cluster. The results can be easily extended to any replication factor the user may decide to adopt.

B. Elastic on-demand services

Cloud computing enables services to be instantiated on demand, according to traffic volume. In the case of web architectures, for instance, it is possible to modify the number of web servers to accommodate the increasing traffic. Service providers have recently included, among the different services, in-memory data stores used as caches. Prominent examples are Amazon's ElastiCache [1], Microsoft's Azure Redis Cache [2] and Google's Cloud Memorystore [3].

These managed solutions take care of the details of the caches, such as software update and maintenance, and provide simple APIs to create and shut down instances, and manage the corresponding cluster of such instances.

The user can choose among a set of possible configurations for each instance. For example, Amazon's ElastiCache [17] allows the customer to choose among instances with different RAM size and number of cores (vCPUs). Different types of instance are also available, like regular, spot and burstable ones. The latter two types refer to instances whose capacity may be changed (reclaimed) by Amazon. In this work, we consider regular instances.

III. Problem definition

In this work, we focus on the caches, without considering the other elements of the specific service which exploits the caches, such as the web server, the back-end databases or the origin server if the cache is part of a CDN. Our aim is to adapt over time the total cache size to the content request pattern in order to minimize the total cost, that is the sum of the storage cost and the cost due to the misses.

The storage cost is immediate to evaluate, because it is determined by the pricing scheme of the cloud provider (we provide later specific examples for Amazon ElastiCache service). The provider offers different possible configurations with different costs. As a design choice, when scaling horizontally, we focus on homogenous instances with fixed size. Since the cost model of the service providers usually has a specific granularity (typically, one hour), we consider fixed intervals that we call epochs, and the choice of changing the number of instances is done at the end of each epoch. Let $I(k)$ be the number of instances selected during the k-th epoch and c^s be the cost of one instance. The storage costs over the first k epochs is then

$$C^s(1,k) = \sum_{h=1}^{k} c^s I(h)$$

Let us now consider the other contribution to the total cost. The cost of a miss can correspond to the additional delay experienced by the final user or to the additional load on the origin server, *e.g.* in terms of number of requests or bytes to serve. In this work, we assume that the service provider has the ability to quantify monetarily the miss cost. For example, there are several studies on the connection between delay and revenues [7]. We denote by m_o the miss cost for a generic object o, that we assume to be deterministic and constant over time.[1]

Let $r(n)$ and $e(n)$ denote respectively the object requested by the n-th miss, and the epoch during which this miss occurs. The total miss cost per time unit during the first k epochs is then

$$C^m(1,k) = \sum_{n | e(n) \in \{1,2,\dots k\}} m_{r(n)}.$$

Our goal is to select the number of instances $I(1)$, $I(2)$, … $I(k)$, in order to minimize the total cost. The tradeoff is evident. At any epoch, a larger number of instances would decrease the number of misses—and therefore the corresponding cost—but it would cause a higher cost due to storage. Conversely, a smaller number of instances would increase the cost due to misses, but it would decrease the cost of storage. Note that the relation between the storage and the number of hits (or misses) is not linear, *i.e.* doubling the storage does not double the number of hits (or halve the number of misses).

In what follows we present a policy that, at the end of each epoch, determines the number of instances to allocate such that the (estimated) total cost for the next epoch is minimal.

On the complexity of the solution. In order to deliver high throughput, caches require small processing overheads. For example, $\mathcal{O}(1)$ time complexity *per request* is considered a hard requirement for CDN caches [12]. At high request rates, more complex operations can pose an intolerable load on the CPU causing spurious misses [18], *i.e.* a requested content may not be served even if present in the cache. This is one of the reasons why eviction policies such as LRU and LFU are widely adopted: in fact their operations (key lookup, removal and insertion) have $\mathcal{O}(1)$ time complexity. On the contrary, more sophisticated eviction policies proposed in the literature, such as the Greedy Dual Size [19] and LRFU [20], may improve over LRU in terms of hit ratio, but have often $\mathcal{O}(\log M)$ time complexity per request (where M is the number of objects in the cache). Therefore, such schemes are unpractical, since they pose high burden on the CPU, as shown also in [21], which presents the popular Adaptive Replacement Cache policy (ARC).

Not only the eviction policy, but any operation related to the cache—including load balancing and cache resizing—needs to have small processing overheads. In the design of our policy, therefore, we will take care of the computational aspects of the proposed solution.

IV. Adaptive TTL based solutions

In this section, we begin with a key building block for the design of a horizontally scalable caching system. Our work draws inspiration from Time-To-Live (TTL) caches, *i.e.* caches that are managed by a TTL policy. There are two families of TTL policy: with and without renewal. In both cases, upon a miss, the content is stored locally and a timer with duration equal to T is activated and the content is evicted when the timer expires. The difference is that, in the case with renewal, the timer is reset by the following hits for the content, while it is not affected by them in the case without renewal. TTL caches are a natural model for DNS caches, but they have also been proposed as an approximate model to study the performance of existing replacement policies like LRU [22]. Moreover, different papers have suggested their practical use because of their higher configurability as well as amenability to analysis [11], [23], [24]. While a replacement policy maintains in the cache as many contents as the available space buffer allows (contents are evicted only if needed to make space), under a TTL policy the actual storage vary over time and is, in theory, potentially unbounded. A real implementation of a TTL cache will have a finite capacity and then it may need to evict some contents even if their timers have not expired yet. Some of these practical issues are discussed in [11]. In our solution a TTL cache with renewal is used as a **virtual cache**, storing only content metadata:[2] by computing its virtual size, our approach steers the addition or removal of cache server instances.

A. Dynamic adaptation

We present an adaptive mechanism based on stochastic approximation by which the timer value converges to the value that minimizes the total cost.

The theoretical results hold in the following scenario. We consider a finite catalogue with N contents and that requests for the different contents occur according to independent renewal processes. We denote by λ_i the request rate for content i. A case of particular interest in what follows is the case where these processes are Poisson ones. Then, a given request will be for content i with probability $\lambda_i / \sum_{j=1}^{N} \lambda_j$ independently from any previous request. This is (a continuous version of) the well known Independent Reference Model (IRM) [25].

In what follows, we consider an ideal TTL cache with renewal and assume that the cloud service charges the user only for the instantaneous storage occupancy. This differs from the more realistic scenario described above where the user needs to pay for the instances independently from their usage, but we will come back to the more realistic billing in

[1]Our theoretical results can easily be extended to the case when miss costs for each object are independent and identically distributed random variables.

[2]For some contents the metadata can have a size comparable with the content itself, but overall in our experiments the total storage required by the virtual cache was negligible.

Sect. V-A. Let s_i be the size of object i and c be the cost per unit time to store a unit of content ([26] shows that prices are almost linear also for real cloud services). Then, the total cost to store content i over a time window of duration τ is $cs_i\tau$. For simplicity, we denote $c_i = s_i c$. A miss for content i incurs a cost equal to m_i. We leave as future work the analysis of the case where the aggregate storage and miss costs are not linear functions (respectively of the cache size and of the number of misses).

Let $X_i(t)$ be the indicator function for the event "content i is stored in the cache at time t" and $M_i(t)$ the counting process of content i misses in the interval $[0, t]$. We can define the storage cost and the miss cost analogously to what is done in Sect. III. The total cost over the interval $[0, t]$ is then

$$\begin{aligned} C(0,t) &= C^s(0,t) + C^m(0,t) \\ &= \sum_{i=1}^{N} \int_0^t X_i(\tau) c_i \, \mathrm{d}\tau + M_i(t) m_i. \end{aligned} \tag{1}$$

If the caching policy uses a constant TTL value equal to T, then each process $X_i(t)$ is a renewal process whose regeneration points are the time instants at which misses of content i occur. The renewal reward theorem guarantees that, for each content, the time-average cost is equal to the expected cost over a renewal period divided by the expected duration of a renewal period, *i.e.*

$$\lim_{t\to\infty} \frac{\int_0^t X_i(t) c_i \, \mathrm{d}t + M_i(t) m_i}{t} = \frac{c_i \tau_{S,i} + m_i}{\tau_{M,i}},$$

where $\tau_{S,i}$ is the expected sojourn time of content i in the cache and $\tau_{M,i}$ is the expected time between two misses.

The asymptotic time average cost ($\mathcal{C}$) of the system as a function of T is then

$$\mathcal{C}(T) = \lim_{t\to\infty} \frac{C(0,t)}{t} = \sum_{i=1}^{N} \frac{c_i \tau_{S,i} + m_i}{\tau_{M,i}}. \tag{2}$$

We observe that $\tau_{S,i}/\tau_{M,i}$ is the asymptotic fraction of time content i spends in the cache or equivalently, the probability that content i is in the cache at a random time, that is often called the occupancy probability and will be denoted in this paper by o_i. The inverse of $\tau_{M,i}$ is the rate at which misses occur that we can also write as $\lambda_i(1 - h_i)$, where h_i is the hit ratio, *i.e.* the fraction of requests for content i that incurs a hit. Then we can rewrite (2) as

$$\mathcal{C}(T) = \sum_{i=1}^{N} c_i o_i + \lambda_i m_i (1 - h_i). \tag{3}$$

When arrivals follow a Poisson process, it holds $o_i = h_i$ because of PASTA property and moreover $h_i = 1 - e^{-\lambda_i T}$ [11]. Then, (3) becomes

$$\mathcal{C}(T) = \sum_{i=1}^{N} c_i + (\lambda_i m_i - c_i) e^{-\lambda_i T}. \tag{4}$$

We can check that if $T = 0$, *i.e.* no content is stored in the cache, the cost per time unit is equal to $\sum_{i=1}^{N} \lambda_i m_i$: we pay systematically for all the misses. Instead, if $T_i = \infty$, all the contents are stored indefinitely and the corresponding cost per time unit is $\sum_{i=1}^{N} c_i$.

We could look for the value T^* that minimizes the cost (4) by applying a gradient algorithm as follows:

$$\begin{aligned} T(n+1) &= T(n) - \epsilon(n) \frac{d\mathcal{C}}{dT}\Big|_{T(n)} \\ &= T(n) + \epsilon(n) \sum_{i=1}^{N} \lambda_i e^{-\lambda_i T(n)} (\lambda_i m_i - c_i), \end{aligned}$$

where the sequence $\epsilon(n)$ converges to zero as n diverges, but it is not summable, *i.e.* $\sum_{n\in\mathbb{N}} \epsilon(n) = \infty$. This approach is not viable because in a realistic scenario popularities are unknown, keep changing over time, and are not easy to estimate. The gradient algorithm suggests us a practical solution based on stochastic approximation [27]. We observe that $\lambda_i e^{-\lambda_i T} = \lambda_i(1 - h_i)$ is equal to the miss rate for content i. Upon a miss, this is for content i with probability proportional to $\lambda_i(1 - h_i)$. Let $r(n)$ be the object requested at the n-th miss and $\hat{\lambda}_i(n)$ be a random unbiased estimate of the arrival rate λ_i with finite variance. Consider the following update rule for the variable $T(n)$:

$$T(n+1) = T(n) + \epsilon(n) \left(\hat{\lambda}_{r(n)} m_{r(n)} - c_{r(n)} \right), \tag{5}$$

where the correction term $\hat{\lambda}_{r(n)} m_{r(n)} - c_{r(n)}$ is a random variable because *i)* content requests occur according to IRM and *ii)* the estimator itself is a random variable. The correction corresponds "in average" to the gradient $d\mathcal{C}/dT$ because, upon a miss, the fraction of requests for content i is proportional to $\lambda_i e^{-\lambda_i T(n)}$, and $\mathbb{E}[\hat{\lambda}_i m_i - c_i] = \lambda_i m_i - c_i$. We denote the correction term as $X(n, T(n))$, whose probability distribution depends on $T(n)$. The following proposition makes the result above formal.

Proposition 1. *Let $\{X(n, T(n))\}$ be a sequence of independent random variables such that $X(n, T(n))$ is equal to $\hat{\lambda}_i m_i - c_i$ with probability $\lambda_i e^{-\lambda_i T(n)} / (\sum_{j=1}^{N} \lambda_j e^{-\lambda_j T(n)})$. Let $\{\epsilon(n)\}$ be a non-negative sequence converging to 0, such that $\sum_{n\in\mathbb{N}} \epsilon(n) = \infty$ and $\sum_{n\in\mathbb{N}} \epsilon(n)^2 < \infty$. Consider the update rule*

$$T(n+1) = \prod\nolimits_{[0, T_{\max}]} (T(n) + \epsilon(n) X(n, T(n))),$$

where $\Pi_{[0,T_{\max}]}(x) = \min(\max(0, x), T_{\max})$ is the projection operator over the interval $[0, T_{\max}]$, then the sequence $T(n)$ converges with probability one to i) a stationary point of $\mathcal{C}(T)$ or ii) 0 or $T_{\max}$, if 0 and $T_{\max}$ are local minima of $\mathcal{C}(T)$.

Proof. We define $a(T) \triangleq \sum_{i=1}^{N} \lambda_i e^{-\lambda_i T} (\lambda_i m_i - c_i)$ and $b(T) = \sum_{i=1}^{N} \lambda_i e^{-\lambda_i T}$. Let $N_{[0,T_{\max}]}(\theta)$ denote the normal cone of the set $[0, T_{\max}]$ at θ. It holds

$$N_{[0,T_{\max}]}(\theta) = \begin{cases} \{0\}, & \text{if } \theta \in (0, T_{\max}) \\ \{x \in \mathbb{R}, x \le 0\}, & \text{if } \theta = 0 \\ \{x \in \mathbb{R}, x \ge 0\}, & \text{if } \theta = T_{\max}. \end{cases}$$

The result follows from Theorem 2.1 in [27, page 127]. All the hypotheses (A2.1) – (A2.5) are satisfied (details in

Appendix A), then the sequence $(T(n))_{n\in\mathbb{N}}$ converges to some limit set of the projected ordinary differential equation

$$\frac{d\theta(t)}{dt} = \frac{a(\theta(t))}{b(\theta(t))} + z(t), \quad z(t) \in N_{[0,T_{\max}]}(\theta(t)), \qquad (6)$$

where $z(t)$ is the projection or constraint term, i.e. the minimum force needed to keep $\theta(t)$ in $[0, T_{\max}]$.

We observe that (6) has only three possible types of limit points: 1) points in $(0, T_{\max})$ for which $a(\theta^*)/b(\theta^*) = 0$, 2) $\theta^* = 0$ iff $a(0)/b(0) \le 0$ and 3) $\theta^* = T_{\max}$ iff $a(T_{\max})/b(T_{\max}) \ge 0$.

As $b(\theta) > 0$ for any $\theta \in [0, T_{\max}]$, and $a(T) = -\frac{d\mathcal{C}(T)}{dT}$, the different types of limit points can be characterized as follows: 1) points in $(0, T_{\max})$ for which $\frac{d\mathcal{C}(T)}{dT}\big|_{T=\theta^*} = 0$, 2) $\theta^* = 0$ iff $\frac{d\mathcal{C}(T)}{dT}\big|_{T=\theta^*} \ge 0$, and 3) $\theta^* = T_{\max}$ iff $\frac{d\mathcal{C}(T)}{dT}\big|_{T=\theta^*} \le 0$. In the first case θ^* is a stationary point of $C(T)$, in the other two cases, it is a local minimum. □

If, instead of letting the weights $\epsilon(n)$ converge to zero, we keep them equal to a small constant value ϵ_0, then, in a stationary setting, $T(n)$ converges to a neighbourhood of the limits indicated in Proposition 1. Note that a constant weight makes it possible to track changes in the system, for example when popularities keep varying over time.

B. An optimal clairvoyant TTL Policy

In this section we present the optimal TTL policy (referred to as TTL-OPT), that minimizes the total cost when the sequence of future requests is known. The cost achieved by this clairvoyant policy is clearly a **lower-bound** for any feasible policy. Among the TTL policies, TTL-OPT plays the same role as Bélády's algorithm [28] for replacement policies. Indeed, Bélády's algorithm minimizes the miss ratio under knowledge of the future requests and uniform content sizes. Interestingly, the optimal clairvoyant TTL policy has polynomial complexity under heterogeneous content sizes and miss costs, while in such case finding an optimal replacement policy is an NP-hard problem [29] (and obviously Bélády's policy is no more optimal).

Algorithm 1: Optimal Clairvoyant TTL policy (TTL-OPT)

```
input : {$c_i$}, storage costs per unit of time
input : {$m_i$}, miss costs
input : request sequence
1 foreach request $r$ do
2     $j \leftarrow$ obj id of request $r$
3     $t_{j,\text{next}} \leftarrow$ time of the next request for obj $j$
4     $c_j^S \leftarrow c_j \times (t_{j,\text{next}} - t_{\text{now}})$
5     if ($c_j^S < m_j$) then
6         $T_j \leftarrow t_{j,\text{next}} - t_{\text{now}}$ // store $j$ until its next request
7     else
8         $T_j \leftarrow 0$ // do not store $j$
```

We allow the optimal policy TTL-OPT to select a different TTL value for each content and for each request. The policy is described in Algorithm 1 and is very simple: given a request for a content, say j, at time t_{now}, if the cost to store the content until its next request (at time $t_{j,\text{next}}$) is smaller than the cost of a miss for this object, then the content should be stored in the cache until the next request, *i.e.* the timer should be set equal to $t_{j,\text{next}} - t_{\text{now}}$. Otherwise, the object should be served but not stored. The formal proof of optimality for TTL-OPT follows.

Proposition 2. *The clairvoyant policy TTL-OPT in Algorithm 1 minimizes the sum of storage and miss costs.*

Proof. Let $C_i(0,t)$ denote the total cost paid during the interval $[0,t]$ for content i, i.e.

$$C_i(0,t) = \int_0^t X_i(t) c_i \, \mathrm{d}t + M_i(t) m_i.$$

The total cost $C(0,t)$ in (1) is then given by the sum of the costs for each content. The possibility to choose the timer value independently for each content reduces the minimization of the total cost $C(0,t)$ to separately minimize each term $C_i(0,t)$.

Let $\{t_{i,k}, k \in \mathbb{N}\}$ be the sequence of time instants of the requests for content i. A TTL policy needs to select a TTL value for each request, let us denote as $T_{i,k}$ the timer for the k-th request occurring at time $t_{i,k}$. We observe that we can restrict ourselves to consider $T_{i,k} \in \{0, t_{i,k+1} - t_{i,k}\}$. In fact, consider any sequence of timer values $\{\hat{T}_{i,k}, k \in \mathbb{N}\}$, and let $\hat{T}_{i,h}$ be a timer such that $\hat{T}_{i,h} < t_{i,h+1} - t_{i,h}$. If we replace $\hat{T}_{i,h}$ with $T_{i,h} = 0$, the cost $C_i(0,t)$ cannot increase. Similarly, we can replace any value $\hat{T}_{i,h}$ such that $\hat{T}_{i,h} > t_{i,h+1} - t_{i,h}$ with $T_{i,h} = t_{i,h+1} - t_{i,h}$, without increasing the cost $C_i(0,t)$. Let then $Z_{i,k}$ be an indicator function such that $Z_{i,k} = 1$ if $T_{i,k} = t_{i,k+1} - t_{i,k}$, and $Z_{i,k} = 0$ if $T_{i,k} = 0$. The total cost for content i can then be rewritten as follows:

$$C_i(0, t_{i,k}) = m_i + \sum_{h=0}^{k-1} \left(Z_{i,h} c_i (t_{i,h+1} - t_{i,h}) + (1 - Z_{i,h}) m_i \right), \qquad (7)$$

where the first term on the right hand side corresponds to the fact that the first request for content i generates always a miss. From (7) if follows that $C_i(0, t_{i,k})$ is minimized by choosing $Z_{i,h} = 1$ if $c_i(t_{i,h+1} - t_{i,h}) < m_i$ and $Z_{i,h} = 0$ otherwise. This corresponds to what TTL-OPT does. □

Clearly, the TTL-OPT policy cannot be used online. Nevertheless, given a trace, its cost can be computed (in polynomial time) and used as a reference.

V. Implementation

As introduced in Sect. III, any operation related to the cache should have $\mathcal{O}(1)$ time complexity *per request* [12]. In this section we present a practical, efficient implementation of a TTL cache with $\mathcal{O}(1)$ complexity. Then, we describe the operation of our elastic caching system, by focusing on the load balancer algorithm that determines the total cache size, and hence the storage cost.

A. Practical implementation of the TTL-based scheme

In what follows we progressively introduce some practical issues one needs to address to implement the TTL-policy.

When to estimate the request rates. A straight application of (5) would require to update the timer immediately upon a miss, and then popularity estimates should be available for contents that are not in the cache. Instead, we will start estimating content popularity immediately after the content is stored in the cache and we will then postpone the timer update to the moment when the estimate is available. The detailed description follows. Let $T(t)$ be the timer value at time t. If the timer is updated at t, then we denote as $T^-(t)$ the value immediately before the update. Updates are, as above, driven by misses, and we denote as t_n the time of the n-th miss and $r(n)$ the corresponding content. Upon a miss, content $r(n)$ is stored and its timer is set to the current value $T(t_n)$. Any new request for content $r(n)$ before the timer expiration will be a hit and will reset the timer to $T(t_n)$. The number of hits for content $r(n)$ during the interval $[t_n, t_n + T(t_n)]$ is recorded. Let us denote this number as $H_{r(n)}$. The ratio $H_{r(n)}/T(t_n)$ is an unbiased estimator of the rate $\lambda_{r(n)}$. Once this estimate is available at time $t_n + T(t_n)$, the timer is updated as follows:

$$T(t_n+T(t_n)) = T^-(t_n + T(t_n)) + \epsilon(n)\left(-c_{r(n)} + \frac{H_{r(n)}}{T_{r(n)}} m_{r(n)}\right). \qquad (8)$$

We observe that in general $T^-(t_n + T(t_n))$ is different from $T(t_n)$, because the timer may have been updated during $[t_n, t_n + T(t_n)]$ as effect of misses for contents other than $r(n)$.

When to update the timers. As a further refinement, we notice that the cache is driven by two main events: request arrival and object eviction. The updates of the timer should be done during these events, so that we do not need to create a specific event for each miss for updating the timer. This adds an additional small delay, as shown in Fig. 1: given a content, the TTL update is triggered by the hit after the first timer (case *a*), or, if no hit occurs after this time, by the content eviction (case *b*).

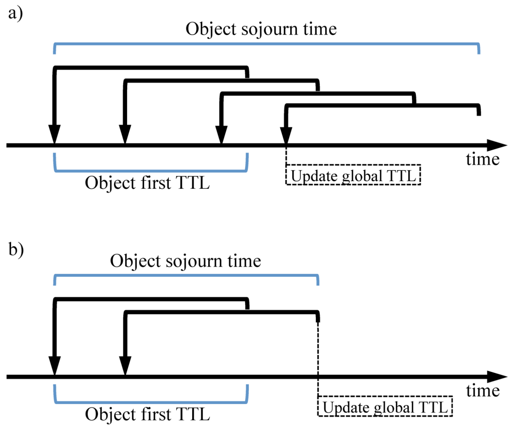

Fig. 1: Global TTL update.

The update rule in (8), together with the additional considerations above, leads to a feasible implementation. We observe that Proposition 1 does not hold for this new algorithm for two reasons. First, the different updates are not independent and identically distributed (conditioned on the current timer value). For example, upon a miss for content i, it is less likely that the following miss will also be for content i, because the content was stored in the cache after the first miss. Second, the update delays could in principle affect convergence. There are theoretical results for stochastic approximation algorithms when the correction terms are correlated and when updates are delayed, and we indeed think that the implementation described above may still converge, but we leave this study for further investigation. The timers are used as keys for organizing the metadata according to the approach proposed in [10], that manages a partially ordered data structure with $\mathcal{O}(1)$ complexity.

B. Horizontally scalable cache system

The TTL-based scheme discussed so far considers a single TTL cache, whose billing is based on its instantaneous storage. In other words, we have considered a perfect vertically-scalable system, where memory resources can be smoothly added and removed. In this section we discuss the design of a more practical horizontally-scalable system inspired by the TTL-based approach, where storage can only change at finite epochs and by some discrete amount.

In a horizontally-scalable solution, cache instances can be added or removed from the cluster, and all the instances have the same configuration. The first design choice to face is the configuration of a generic instance.

Cache instances. These are the physical caches storing the actual contents and have fixed size. They can be implemented using Memcached or Redis with a simple eviction policy like LRU.

Load balancer. The load balancer performs the ordinary operations, such as request routing, and content insertion, *i.e.*, in case of a miss, after retrieving the object from the origin or the back-end, it stores it in one of the cache instances. In addition, the load balancer maintains a virtual cache, with the references of the requested objects: this virtual cache is going to be managed as a TTL cache according to the description in Sect. V-A with $\mathcal{O}(1)$ computational cost per request. The size of the virtual cache depends on the timer value T, which in turn depends on the number of hits and misses, and on the corresponding costs for the storage and for the misses. Thus, the size of the virtual cache can be used to determine the number of actual instances to employ in the cluster.

Operation. Our scheme is described in Algorithm 2. At every request, we look for the object key in the virtual cache, update its position in case of a hit, or add it in case of a miss. Then, we start evicting objects from the virtual cache if they are expired. While inserting a new object or removing expired objects, we update the total size of the cache (the sum of the sizes of non-expired objects). Clearly, object sizes can be heterogeneous. At the end of the epoch, we look at the size of the virtual cache and we select the number of instances such that the sum of the sizes is the closest to the virtual cache size. At the end of the observation interval, if the number of instances has changed, the load balancer reassigns the responsibility of the hash space to the current instances.

Algorithm 2: TTL-based scaling

```
input : VC, Virtual Cache
input : $S_p$, Physical cache size
output: $I(k+1)$, # of the instances in $k+1$-th epoch
1 foreach request $r$ do
2     if ($r \in VC$) then
3         REMOVE($r$, VC)
4     $r.expire \leftarrow t_{\text{now}} + TTL_{\text{now}}$
5     INSERT($r$, VC)
6     EVICTEXPIRED(VC)
7 if (epoch $k$ ended) then
8     $I(k+1) \leftarrow$ ROUND(VC.size / $S_p$)
```

Additional considerations. We observe that objects stored in the physical caches may be different from the ones maintained by the virtual cache. When a physical cache needs to make space for new data, it may evict one content before the timer of the corresponding ghost at the virtual cache expires. Instead, the eviction of meta data associated to a content from the ghost does not cause the eviction of the actual content from the physical cache. Moreover, when cloud instances are added or removed, the object key space responsibility must be rearranged, which may lead to spurious misses due to route changes: the object is in a physical cache, but the request is routed to a different one. Overall, therefore, we will observe virtual misses at the virtual cache, and actual misses at the physical cache, and these two values may be different. We have experimentally observed that, since the number of requests within an epoch is usually very high, the effect of spurious misses due to the change of the number of instances is negligible.

VI. Experimental evaluation

A. Setup

Testbed. We evaluate our approach using a testbed that is representative of a typical web architecture. An application server is connected to a database and to a cluster of caches. The application receives the requests and checks if the content is stored in the cache. If the content is not in the cache, the application server retrieves the object from the database, serves the client and stores the object in the cache. For the operations related to the cache (*e.g.*, object lookup), the application server relies on a *load balancer*. We have implemented the scheme described in Sect. V-B in a custom tool similar to `mcrouter` [15]. The tool is able to add or remove the cache instances from a local cluster, but it can be easily extended to use the APIs of any cloud cache provider.

Trace. The requests sent to the application server are generated by reproducing two anonymized traces collected respectively for 30 and 5 days at two vantage points of the Akamai network. The traces we tested contain the timestamp of the request arrival, the anonymized object ID and the size of the requested object. We report the results for the 30-day trace, since we obtain similar qualitative results with the 5-day trace. In the 30-day trace, there are $2 \cdot 10^9$ requests for 110 millions contents, whose size varies from a few bytes to tens of MB. Figure 2 (left-hand side) shows the number of requests for each object, sorted by rank (in terms of popularity). The right-hand side shows the empirical Cumulative Distribution Function (CDF) for the size of the requested objects (without aggregating requests for the same object). Additional information about the traces can be found in [18, §4.2]. We could not carry on our experiments with other traces, like the ones collected in the public repository [30], because they refer to low level storage (block I/O), and they are not representative of a typical cloud-based application, such as a web service.

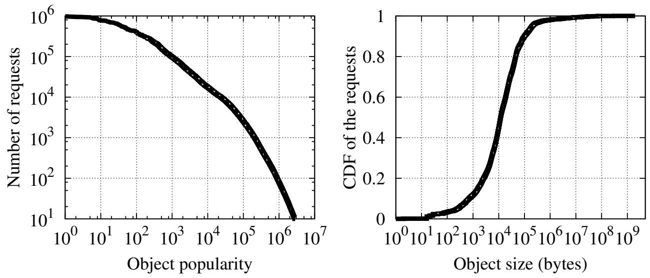

Fig. 2: Number of requests per object, ordered by rank (left), and cumulative fraction of the requests for objects up to a given size (right).

Settings. For the configurations and the costs, we refer to Amazon ElastiCache service [17]. For the duration of the epoch, we consider the minimum billing time, which is one hour. Among the different instances' options, we selected the *cache.t2.micro* with 0.555 GB RAM and one vCPU, which costs 0.017$/hour (Oct. 2017, US based). We use a small instance since it provides a fine granularity when we resize the cluster: for instance the experimental results in Fig. 4 shows that one small instance is sufficient during low traffic periods. Moreover, bigger instances (*e.g.*, with 3.22 GB or 6.05 GB) have just two vCPUs, which may limit the throughput of the cache. Replicating small instances each with a vCPU helps in maintaining the throughput while scaling the cluster. As for the cache, we tested both Redis and Memcached: even if they are both able to handle heterogeneous object sizes, we report the results for Redis, since Memcached provides slightly worse performance due to calcification [31], [32], [33].

In order to determine plausible miss costs, we reasoned as follows. The production server from which our trace was collected had an in-memory cache of 4 GB [12], *i.e.* roughly corresponding to eight *cache.t2.micro* instances. We assume that this system has been engineered so that storage and miss costs are equal, a reasonable rule of thumb to achieve a small total cost. The storage cost can be determined in our case considering the corresponding hourly cost of eight *cache.t2.micro* instances. By dividing this cost by the average number of misses observed during one hour in production, we obtain the cost per miss (in our case, 1.4676 $\times 10^{-7}$\$ per miss). Below we also evaluate the effect of different miss costs.

Baseline policies. Because of the considerations above, we consider as baseline a scenario with eight *cache.t2.micro* instances. In addition, as a reference, we consider the scenario with an ideal, vertically scalable, TTL cache, billed according to its instantaneous size. A practical TTL-based policy can approach the performance of this scheme only if *i)* billing periods become arbitrarily small, and *ii)* caches of any size can be rented. Finally, as a state-of-the-art-solution, we compare also our results with an elastic resource allocation scenario driven by the Miss Ratio Curves (MRCs) [13]. MRCs are a well-known tool for cache profiling: in a single graph it is possible to observe the relation between cache size and miss ratio, therefore one can compute the cost of the storage and estimate the cost of the misses for each point. Nevertheless, this approach has some issues, which we discuss in detail in the next section, before showing the overall results.

B. On the MRCs

While the MRCs seem the straightforward solution for deciding how to scale cloud caches, they have a main issue: their computational complexity. The seminal algorithm proposed by Mattson [34] takes $\mathcal{O}(M)$ operations per request, where M is the number of objects in the cache. A more efficient implementation, proposed by Olken [35], makes use of a tree data structure (*e.g.*, *counting B-Tree*) to keep track of the objects in the cache. This reduces time complexity to $\mathcal{O}(\log M)$ per request. In order to achieve the target $\mathcal{O}(1)$-complexity per request, many solutions have been proposed in the literature that compute approximate MRCs [13] [36] [37] [38]. Such solutions share a common characteristic: they have been designed considering objects with uniform sizes. On the contrary, the applications we are interested in exhibit contents with *heterogeneous sizes*.

We observe that it is possible to extend Olken's approach to MRC computation to the case of heterogeneous size contents maintaining $\mathcal{O}(\log M)$ complexity per request. To this aim, we suggest to use a special data structure, called *order statistics tree*, that has a method `rank(`x`)`, which returns the sum of the weights of the elements with keys less than or equal to x (the weights are the object size).[3] We can also modify analogously the algorithms to compute approximate MRCs to take into account heterogeneous sizes, but, as our following experiment shows, there is a significant loss of accuracy.

We consider in particular the method proposed in [37], [38] (but the others operate in a similar way): the request trace is sampled with rate R, a first MRC is computed on the subsampled trace – in case of heterogeneous size, we compute the MRC by including the size of the sampled objects – which is then scaled up opportunely (according to the samplig rate used) to obtain the approximate MRC for the whole traffic. A constant sampling rate would lead to $\mathcal{O}(\log RM)$ time complexity. By fixing the amount of memory dedicated to the observed objects, and adapting dynamically R according to such memory, the authors in [37] claim to achieve $\mathcal{O}(1)$ complexity. However, with the typical values used in the experiments in [37], the constant terms hidden in the $\mathcal{O}(1)$ complexity scheme are comparable to those of the $\mathcal{O}(\log RM)$ complexity scheme. The accuracy of the sampling methods is evaluated through the prediction error [37], which is computed as the mean of the absolute difference between the exact and the approximated MRCs.

We use the 30-day trace whose characteristics are reported in Fig. 2. For each request, besides the timestamp and the object identifier, we have the object size. First, we ignore the actual object size and assume it to be uniform. In this case, the method predicts the MRC with a prediction error smaller than $3 * 10^{-3}$ for all sampling rates between 0.1 to 0.001—see Fig. 3—, similarly to what observed in the original papers [37] [38]. We then consider the object sizes (therefore we consider the MRC computed using this information) and repeat the experiments: for a given sampling rate, the error increases by one order of magnitude! Moreover, in order to reach a given target error it may be necessary to increase by two orders of magnitude the sampling rate. Correspondingly, the dynamic sampling rate approach described in [37] would require a larger memory footprint and a larger number of operations for request.

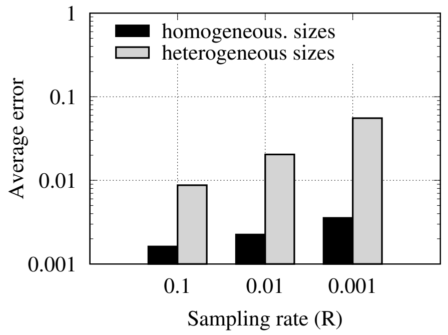

Fig. 3: Accuracy of the approximate MRC computation through sampling, with uniform and nonuniform object sizes.

This simple experiment shows that object sizes may have an unexpected impact on the accuracy of the approximated MRC computation. Note that we took particular care in adapting the approximate MRC computation in [37], [38] to the heterogenous case, but it seems that more sophisticated sampling methods are required. Other approximated techniques, designed to have $\mathcal{O}(1)$ time complexity per request, such as MIMIR [13] or AET [39], may be affected when considering a scenario with heterogeneous object size. In MIMIR, the authors state that their solution can be extended to consider non-uniform sizes, but they do not experimentally support their claim, and their solution is not available online. Similarly, the AET approach assumes uniform object size, and it is not clear if it can be easily extended to the heterogeneous case.

In summary, the approximate computation of the MRC with $\mathcal{O}(1)$ time complexity per request still needs to be studied in depth, especially when contents have different sizes. Thus,

[3]This is how we compute MRCs in all the experiments shown in this paper. We suspect that this approach may be known, but we were not able to find it described elsewhere.

the only option is to compute the MRCs exactly, which has $\mathcal{O}(\log M)$ complexity per request.

C. Results with traces

We present here the results for the trace described in Sect. VI-A, where the arrival pattern varies over time. In Appendix B we also show the result in case of IRM synthetic traffic. Our TTL approach continuously tracks such a variation: this is shown in Fig. 4 (left), where we plot the value of the TTL for an interval of four representative days: the evolution clearly follows a daily pattern. The fluctuation of the TTL is mirrored by the virtual cache size (Fig. 4, right), which varies from zero (the cost of the few misses does not justify the storage of the objects) to 3.5 GB.

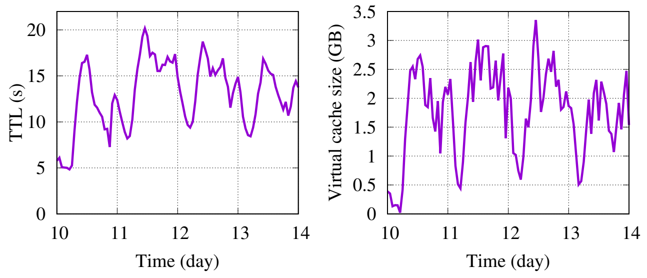

Fig. 4: Virtual cache: TTL over time (left), and cache size (right).

The virtual cache size translates into the number of instances used in the cluster. From this, it is possible to compute the total cost for storage and misses.

Figure 5 shows the per-hour cost of different cache scaling approaches over the 30 days, along with a zoom on two representative days. In particular, we compare our TTL-based system (labelled as "*TTL-practical*") with:

- *Fixed-size*: 8-instance cache, corresponding to our reference in-memory production cache;
- *MRC-based*: dynamic cache resizing in which, at every epoch, the MRCs are computed using the general approach discussed in Sect. VI-B, where we consider the (heterogeneous) object sizes;
- *TTL-ideal*: vertically scalable TTL cache where the cost per hour has been computed considering a charging scheme that takes into account the instantaneous occupancy;
- *TTL-OPT*: optimal clairvoyant policy described in Sect IV-B.

The results show that the TTL-based approach obtains similar cumulative costs as the MRC-based approach, but with a $\mathcal{O}(1)$ complexity instead of $\mathcal{O}(\log M)$ complexity.

In order to evaluate the overall economic impact of the adoption of a dynamic cache adaptation, we compute the cumulative cost over the 30 days, which is shown in Fig. 6. The figure also contains a zoom of the last two days to highlight the final costs at the end of the 30 days. Overall, with respect to the baseline fixed-size approach, the TTL-based approach is able to save 17% of the costs. The difference between the ideal and the practical TTL-based implementations is due to the

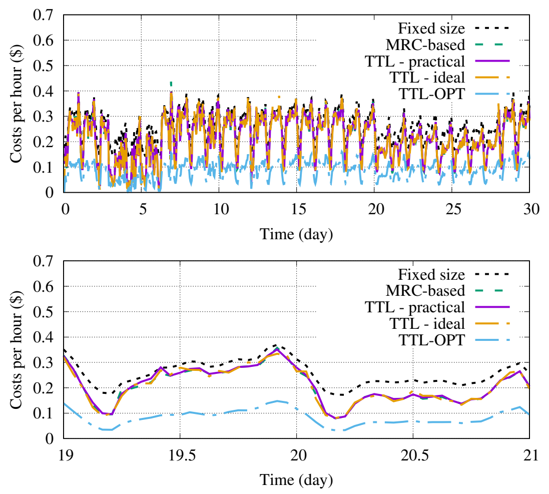

Fig. 5: Hour cost of TTL-based approach compared to fixed-size, MRC-based, and ideal pure TTL (on the bottom, zoom of two representative days).

discretization of cache sizes and billing periods. Nevertheless, such a difference causes only a 2% cost increase compared to the ideal TTL implementation. Interestingly, this result suggests that, at least for typical CDN applications, there is no need for finer-grained billing periods or cache sizes, but most of the potential improvement is already achievable with the current offer.

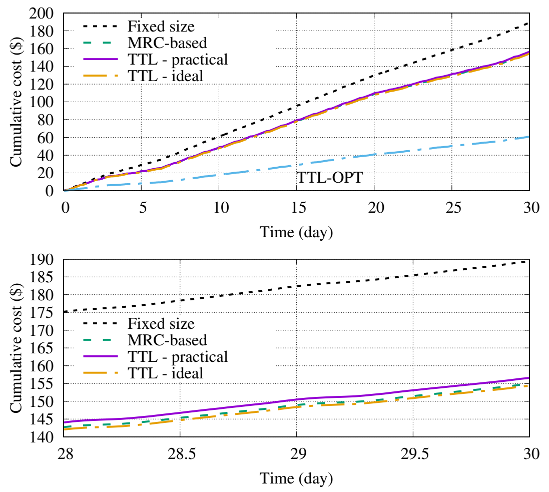

Fig. 6: Cumulative cost of TTL-based approach compared to fixed-size, MRC-based, and ideal pure TTL (on the bottom, zoom on the last two days).

Considering the optimal clairvoyant policy, we see that there is room for even more significant cost savings: TTL-OPT achieves a cost that is one third of the baseline. TTL-OPT assumes to know the sequence of future requests and is thus unpractical. Nevertheless, this result suggests that potential improvements can come from TTL policies that use different TTL values for different contents (as TTL-OPT does) selecting the timer value on the basis of a forecast for the next inter-arrival time. In the future we plan to investigate this possibility.

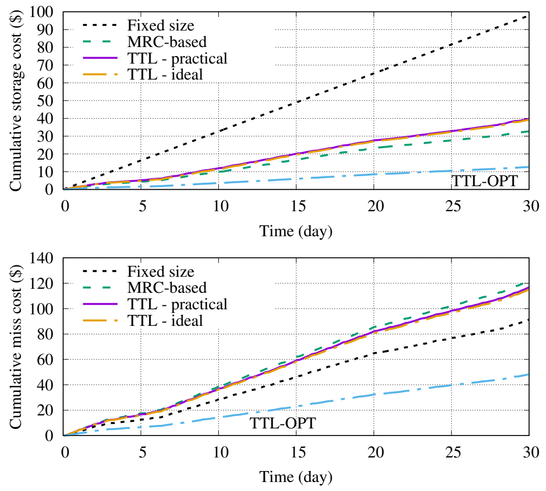

Fig. 7: Cumulative storage cost (top) and cumulative miss cost (bottom) for different approaches.

In Figure 7 we show the two cost components: the cumulative storage cost (left) and the cumulative misses costs (right). The MRC-based solution maintains a smaller number of instances, which translates into a slighltly higher cost due to misses. Nevertheless, their sum is similar to the one obtained by the TTL-based approach, suggesting that, when we are close to the minimum, different configuration options are available.

As discussed at the end of Sect. V-B, when we dynamically change the number of caches, there could be spurious misses due to the reorganization of the object key responsibility. Figure 8 shows the fraction of misses that are spurious. It is easy to see that, besides few peaks, less than 1% of the overall misses are due to spurious misses, and they have therefore little impact on the overall costs. Note that such a cost is paid by any dynamic cache resizing scheme, and it is not specific to the proposed TTL-approach.

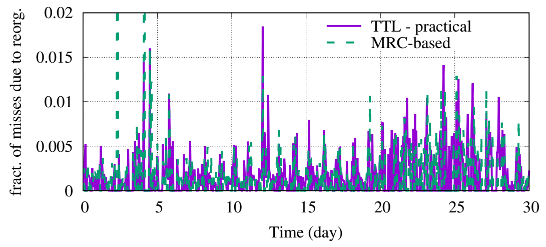

Fig. 8: Fraction of misses that are *spurious*, i.e., misses due to the reorganization of the object key responsibility.

In addition to the costs, in order to understand better the differences between our TTL-based solution and the MRC-based solution, we consider the burden imposed on the CPU of the two schemes, along with the baseline scenario, *i.e.* a fixed number of cache instances, where the load balancer simply routes requests to the correct instance using the hash of the key. In the first experiment, we replay the trace, *i.e.*, we generate the requests following the timestamps provided in the trace. Figure 9, left, shows the CPU load over time for two representative days. We see that the additional task to compute the MRC leads to almost double the CPU usage in comparison to the basic scenario, where the load balancer only distributes the requests among a fixed number of instances. On the contrary, the overhead of our TTL-approach is below 20%. While the hardware we used in the testbed was adequate to support also the computationally expensive MRC-approach, should the requests rate increase, a scheme with logarithmic complexity would not be able to cope with the processing leading to spurious misses [18]. Alternatively, the load balancer is likely to be scaled up, inducing additional fixed costs.

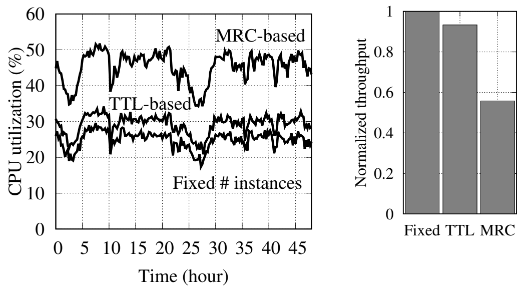

Fig. 9: Left: CPU load using a fixed number of instances routing scheme, our TTL-based solution (both with $\mathcal{O}(1)$ time complexity) and an MRC-based solution (with $\mathcal{O}(\log M)$ time complexity). Right: Throughput normalized to the fixed scheme case.

These findings are confirmed in the second experiment, which is the backlogged version of the first one: we ignored the trace timestamp, and generated a new request as soon as we received the reply from the load balancer for the previous request. This is indicative of the maximum throughput achievable by the different schemes. The results are shown in Fig. 9, right. For ease of representation, we normalize the throughput with respect to the basic scenario with a fixed number of instances. While our TTL solution experiences about 8% throughput reduction due to the additional data structure we maintain, the MRC solution almost halves the achievable throughput.

We conclude by observing that our results hold for a single-server trace collected by Akamai in a 30 days period. It is thus tempting to project our results to the current scale of a major CDN provider, for a yearly timeframe. According to the trend we measure in Fig. 6, our approach can potentially save millions of dollars, when compared to a best practice static configuration.

D. *Additional analyses*

Load analysis. Since we are dealing with a (dynamic) distributed cache, one may wonder if the assignment of object keys to cache instances is balanced. Note that Redis does not use consistent hashing, but a two-step scheme [16]. There are 16384 slots, and objects keys are hashed into one of the slots. Each slot is randomly assigned to a server. When a new server is added, some randomly selected slots are transferred to the new server. When a server is removed, its slots are transferred to the other randomly selected servers.

To understand if each server maintains the responsibility of approximately the same number of slots, we have considered, for each interval, the minimum and the maximum number of assigned slots to each server, and we have normalized them with the expected number of slots per server. When there is just one server, clearly the minimum and the maximum and the expected number of slots are the same. In Figure 10 (left), we can see that each server deviated from the expected number of slots by at most 2.5%.

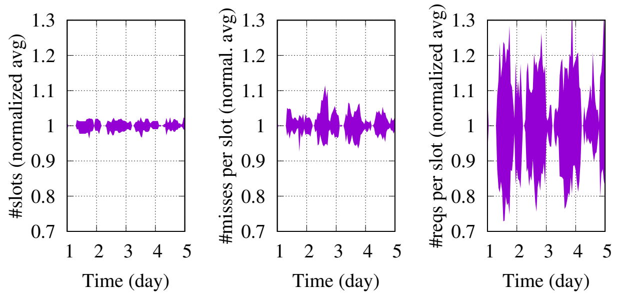

Fig. 10: Normalized mean number of slots per server, misses per slot and requests per slot.

Similarly, we computed the number of misses per server (minimum and maximum, normalized to the total number of misses divided by the number of servers). Here the distribution among server is more spread, with servers that sometimes get 10% more misses than the expected average. In addition, we have considered the number of requests per server. The load balancer tries to achieve distributed evenly the keys among the servers, but the actual number of requests depends on the popularity of a key. Figure 10 (right) shows that sometimes servers need to respond to 30% more traffic than the average. Rebalancing the server load can be done with known techniques that keeps track of the highly loaded slot and balance them among the servers, such as the ones presented in [40] and [41].

Sensitivity analysis. The update of the TTL is based on (8), which contains the weights sequence $\epsilon(n)$. As discussed in Sect. IV-A we keep them equal to a small constant value ϵ_0. Figure 11 (left) shows the total cost of the TTL-based system for different values of ϵ_0. The choice of this parameter is not critical: the cost change is limited to a few percent, while ϵ_0 varies by 4 orders of magnitudes. The default value used in all our experiments is $\epsilon_0 = 10^{-4}$. We also tested the sensitivity to the miss costs, by scaling them by a factor γ in comparison to the reference value computed in Sect. VI-A. Figure 11 (right) shows that the TTL-based approach consistently achieves almost the same cost of the more complex MRC-based approach, which represent the state-of-the-art solution for the determination of the best online number of instances.

Optimal fixed size. In Fig. 6 we compare the performance of different dynamic approaches as well as of a fixed size cluster with 8 instances, corresponding to a total memory of about 4GB, as the memory of the cache from which the traces

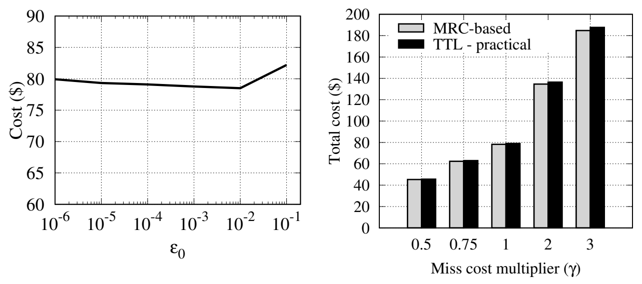

Fig. 11: Total costs after 15 days as a function of the parameter ϵ_0 (left) and the miss cost multiplier γ (right).

were collected. Figure 12 shows the total cost as well as its two components for a fixed cluster with different numbers of instances and for two different values of γ (*i.e.*, different miss costs). The corresponding cost of the TTL-based solution is also reported as a horizontal segment (as well as readable in Fig. 11 (right)). It appears that our solution is able to dynamically approach the best possible static configuration with costs equal to about \$80 and \$138 respectively for $\gamma = 1$ and $\gamma = 2$.

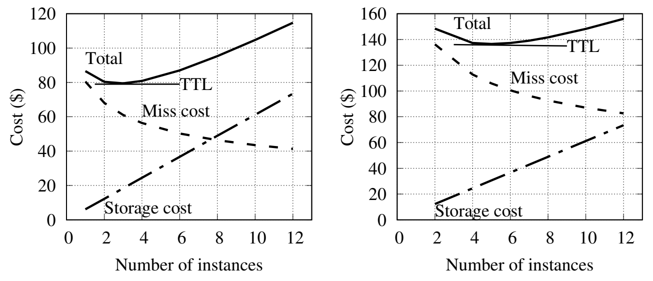

Fig. 12: Fixed number of instances: costs as a function of the number of instances (left: $\gamma = 1$; right: $\gamma = 2$).

A highly dynamic scenario: the Super Bowl case. Cloud-based services are particularly adapted to time-limited and highly dynamic settings for which the costs to deploy and manage an ad hoc infrastructure would not be justified. Our traces do not correspond to such a scenario. Akamai CDN has been indeed engineered to satisfy long-term service level agreements with a given number of content providers. Moreover, the traces have been collected directly at a cache that is located behind a load balancer that tries to keep the request rate at a given cache as uniform as possible. We were not able to find real traces representative of a highly dynamic request scenario. We decided then to gauge part of our traces to qualitatively reproduce the traffic variability at a large-scale event as the Super Bowl. To this aim, we consider the wireless data traffic generated by the attendees of the Super Bowl XLVII as described in [42]. In particular the authors of [42] mention that data traffic at the stadium increased by a factor of seven during an interval of about 8 hours (from a couple of hours before the beginning of the game until midnight). To reproduce such a scenario, we consider 8 hours of our 30-day trace, and we sample the traffic before and after this interval, such that the traffic during the 8-hour interval is seven time

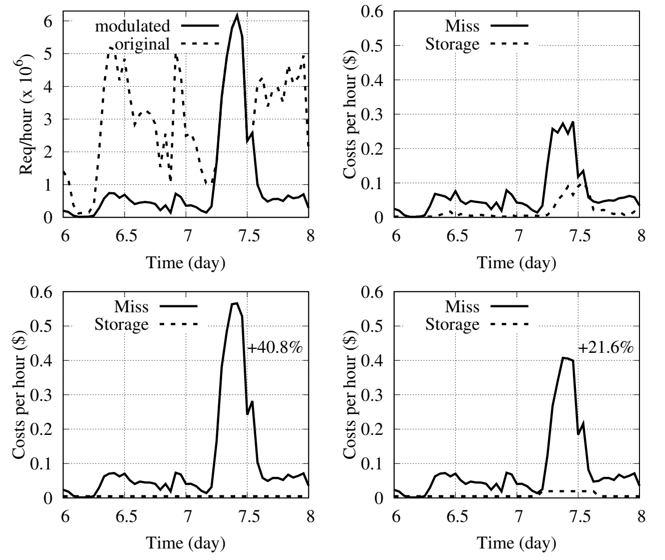

Fig. 13: Traffic increase at day 7 (increase factor: 7). Number of requests (top left). Storage and miss costs with TTL (top right), with fixed number instances when the event is not known in advance (bottom left) and when it is (bottom right).

larger than the average. Figure 13 (top-left) reports the original traffic pattern, and the one shaped to mymic the Super Bowl one.

We then compare our dynamic TTL-based configuration with fixed static configurations in two different scenarios. In the first scenario, the large-scale event is unexpected and then a static cache system is (optimally) sized on the basis of the usual traffic (seven times less than the Super Bowl peak). In the second scenario instead, the occurrence of the event is known and a cache is instantiated for its duration and sized on the basis of a traffic forecast that underestimate the peak traffic by 80%. We derive also this forecast error from [42]. In fact, the data traffic of Super Bowl XLVII exceeded that of the previous edition by 80% also due to a half-hour power outage that caused the game to be suspended, and then people to spend more time on their mobile (the wireless network was not affected by the outage).[4] The costs of our adaptive TTL solution are shown in Fig. 13 (top-right). The figure shows how the number of instances changes during the game period in order to amortize the cost due to misses. The corresponding plots for the two static configuration scenarios are in the bottom part of the figure. A static configuration incurs a total cost 40% larger than the TTL dynamic configuration when the event is not known in advance and 21% larger when the cache is sized on the basis of the previous edition of the Super Bowl. In summary, while a fixed number of instances can be engineered based on past traffic, unexpected events may put a burden on the caching system. Our dynamic TTL approach is able to adapt to these sudden changes, and it provides the minimum cost – we computed the cost also with a MRC-based approach, obtaining similar results – still maintaining the computational complexity low.

[4]The authors of [42] also highlight how the traffic was significantly affected by the football game itself, with most exciting/boring quarters causing a remarkable decrease/increase of traffic.

As for the spurious misses, Fig. 14 shows that, even in this highly dynamic scenario, they account only for a fraction of the overall misses – less than 0.4%.

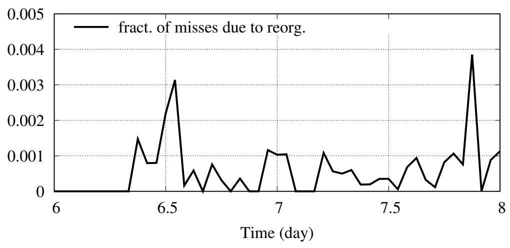

Fig. 14: Fraction of misses that are *spurious*, for the highly dynamic scenario.

We also observe that the TTL-policy has no a priori information about the ongoing event and we have kept the epoch duration to one hour, hence the number of instances is only updated 8 times during the whole event. If there is some a priori knowledge about the event like its duration or/and a traffic forecast, one could clearly think to exploit it, for example making the TTL-policy adapt faster during this event or set the initial number of instances to the optimal static value predicted on the basis of the traffic forecast. This would further reduce the cost of our TTL solution.

VII. Related work

Elastic resource provisioning of cloud services has been the subject of many studies. The authors in [6] provide a general overview of the techniques, such as control theory as used in [43]. Despite the broad set of results, they are computationally too expensive, and it is not clear if they can be applied in the context we consider, where the relation between the resource deployed and the key performance index (the hit ratio) is not linear. Moreover, none of them uses stochastic approximation for resource allocation as we do. Another prominent example of a general approach for auto-scaling is given by [44] but the proposed solution is based on methods (*e.g.*, time series prediction) that are too computationally intensive for the high-throughput scenario we consider.

Memory management is related to our problem but it rather aims to determine how the available memory should be shared among competing applications. Moreover, the proposed solutions, such as [8] [31] [32] [33], all require computations with higher complexity than our approach.

As for minimizing costs in a cloud computing environment, the authors in [45] and [26] explore the use of spot instances for different aims, such as content replication or decreasing the overall storage cost. Beside the computational complexity of the solution, the proposed schemes do not take into consideration the cost due to misses, as we do. The authors in [26] also consider a policy for modulating the allocation of on-demand instances to match the dynamic needs, but they do not describe it in detail.

A heuristic for horizontal scaling cloud caches is proposed in [46], but it does not explicitly take into account storage and miss costs as we do. Moreover, each cache implements a simple LRU policy. The optimal static content allocation in a network of elastic caches has been studied in [47] under the assumption that content request rates are known and stationary. Our policy, on the contrary, dynamically adapts to a time-varying request process.

Part of our TTL-based solution is based on the concept of a virtual cache, which maintains the metadata of cacheable objects, but not their content. These metadata are sometimes referred to as *ghosts*. This additional information is used in many caching schemes to decide how to manage the objects in the physical cache. For example in 2-LRU [48] the virtual cache is managed by LRU and a content is actually stored in the physical cache only if its metadata is already present in the virtual cache. As another example, ARC [21] uses two virtual caches to decide which contents should be evicted. Differently from the cases above, we use a virtual cache to size (multiple instances of) the physical one.

A recent work [24] also explores how to adapt the TTL value to the request pattern by using stochastic approximation. In particular, the authors focus on vertical scaling and aim to achieve a target hit ratio, possibly with a small cache size. On the contrary, our approach addresses horizontal scaling to minimize the total operational cost. Reference [23] aims to identify the optimal TTL value for each content in order to maximize a strictly convex utility function of hit rates, but their work considers the usual scenario with a fixed-size cache. Moreover, their approach cannot be easily extended to linear functions (see the discussion in [49]).

After our first results appeared in [50], two other papers, [51] and [52], looked at caching policies for elastic caches. In [51] the cost model is similar to ours but with equal miss costs for all requests. The authors consider policies with eviction based on a constant TTL and insertion upon the M-th request. They perform a competitive analysis of their policies and derive explicit average cost expressions and bounds under renewal request processes. In our paper, miss costs are heterogeneous and the TTL value adapts at run-time on the basis of the request process. The model in [52] is quite different: *i)* the request arrival rate is constant with one request every time-slot, *ii)* an arbitrary number of contents can be removed at each time-slot, paying an eviction cost for each of them, but there is no miss cost, *iii)* the storage cost can be an arbitrarily monotone function of the set of contents stored in the cache. [52] presents deterministic and randomized algorithms with competitive ratios' guarantees.

VIII. Conclusion

Dynamic sizing of cloud caches allows cloud users to adapt the cache size to the traffic pattern and minimize their total cost, which is given by storage and misses costs. We studied a TTL-based solution to dynamically track the required cache size. We provided a theoretical lower bound for the cost achievable by TTL solutions: in fact we characterize the optimal TTL policy (TTL-OPT) when the sequence of future requests is known. Moreover, we discussed a practical low-complexity implementation of a TTL solution, and evaluated it using real-world traces. Our experiments show that our solution is able to track the optimal cache configuration and achieve significant cost savings, especially in highly dynamic settings that are likely to require elastic cloud services. A key aspect of our solution is its low computational complexity, which is required for achieving high throughput. Our results also suggest that, at least for typical CDN applications, there is no need for finer-grained billing periods, but most of the potential improvement is already achievable with the current offer.

Encouraged by the experimental results related to a practical TTL cache implementation, we are exploring, from a theoretical point of view, the impact of the update delay on the convergence of TTL stochastic update rule. Moreover, our comparison with TTL-OPT suggests that there are possibilities for significant additional cost savings, if TTL values could be adapted on a per-content basis, as a function of the specific arrival pattern.

Acknowledgment

This work was partially supported by the Italian National Group for Scientific Computation (GNCS-INDAM) and by "Progetto di Eccellenza" of the Computer Science Dept., Univ. of Verona, Italy.

References

[1] AWS, "Amazon Web Service ElastiCache," 2018, accessed: Jan. 2018. [Online]. Available: https://aws.amazon.com/elasticache/

[2] Microsoft, "Azure Redis Cache," https://azure.microsoft.com/en-us/services/cache/, accessed: Jan. 2018.

[3] Google, "Cloud Memorystore," https://cloud.google.com/memorystore/, 2018, accessed: May 2018.

[4] Memcached, "Memcached," https://memcached.org/, 2018, accessed: Jan. 2018.

[5] Redis, "Redis," https://redis.io/, 2018, accessed: Jan. 2018.

[6] T. Lorido-Botran, J. Miguel-Alonso, and J. A. Lozano, "A review of auto-scaling techniques for elastic applications in cloud environments," *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[7] S. Work, "How Loading Time Affects Your Bottom Line," https://blog.kissmetrics.com/loading-time/, 2018, accessed: Jan. 2018.

[8] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching." in *HotStorage*, 2015.

[9] C. Li and A. L. Cox, "Gd-wheel: a cost-aware replacement policy for key-value stores," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 5.

[10] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic caching: Flexible caching for web applications," in *USENIX ATC*, 2017.

[11] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Performance evaluation of hierarchical TTL-based cache networks," *Computer Networks*, vol. 65, pp. 212 – 231, 2014.

[12] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network." in *NSDI*, 2017, pp. 483–498.

[13] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic performance profiling of cloud caches," in *Proceedings of the ACM Symposium on Cloud Computing*. ACM, 2014, pp. 1–14.

[14] 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.

[15] Facebook, "mcrouter," https://github.com/facebook/mcrouter, 2018, accessed: Jan. 2018.

[16] Redis, "Cluster Specification," https://redis.io/topics/cluster-spec, 2018, accessed: Jan. 2018.

[17] AWS, "Amazon Web Service ElastiCache Pricing," https://aws.amazon.com/elasticache/pricing/, 2018, accessed: Jan. 2018.

[18] G. Neglia, D. Carra, M. Feng, V. Janardhan, P. Michiardi, and D. Tsigkari, “Access-time-aware cache algorithms,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 2, no. 4, pp. 21:1–21:29, Nov. 2017. [Online]. Available: http://doi.acm.org/10.1145/3149001

[19] P. Cao and S. Irani, “Cost-aware www proxy caching algorithms.” in *Usenix symposium on internet technologies and systems*, vol. 12, no. 97, 1997, pp. 193–206.

[20] D. Lee, J. Choi, J.-H. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. S. Kim, “On the existence of a spectrum of policies that subsumes the least recently used (lru) and least frequently used (lfu) policies,” in *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 1. ACM, 1999, pp. 134–143.

[21] N. Megiddo and D. S. Modha, “Arc: A self-tuning, low overhead replacement cache.” in *FAST*, vol. 3, 2003, pp. 115–130.

[22] H. Che, Y. Tung, and Z. Wang, “Hierarchical Web caching systems: modeling, design and experimental results,” *Selected Areas in Communications, IEEE Journal on*, vol. 20, no. 7, pp. 1305–1314, Sep 2002.

[23] M. Dehghan, L. Massoulié, D. Towsley, D. Menasche, and Y. C. Tay, “A utility optimization approach to network cache design,” in *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, April 2016, pp. 1–9.

[24] S. Basu, A. Sundarrajan, J. Ghaderi, S. Shakkottai, and R. Sitaraman, “Adaptive ttl-based caching for content delivery,” in *Proceedings of the 2017 ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, 2017, pp. 45–46.

[25] E. G. Coffman and P. J. Denning, *Operating systems theory*. Prentice-Hall Englewood Cliffs, NJ, 1973, vol. 973.

[26] C. Wang, B. Urgaonkar, A. Gupta, G. Kesidis, and Q. Liang, “Exploiting spot and burstable instances for improving the cost-efficacy of in-memory caches on the public cloud,” in *Proceedings of the Twelfth European Conference on Computer Systems*. ACM, 2017, pp. 620–634.

[27] H. Kushner and G. Yin, *Stochastic Approximation and Recursive Algorithms and Applications*, ser. Stochastic Modelling and Applied Probability. Springer New York, 2003. [Online]. Available: https://books.google.fr/books?id=EC2w1SaPb7YC

[28] L. A. Belady, “A study of replacement algorithms for a virtual-storage computer,” *IBM Systems journal*, vol. 5, no. 2, pp. 78–101, 1966.

[29] S. Hosseini-Khayat, “On optimal replacement of nonuniform cache objects,” *IEEE Transactions on Computers*, vol. 49, no. 8, pp. 769–778, 2000.

[30] SNIA, “iotta repository block I/O traces,” http://iotta.snia.org/tracetypes/3, 2018, accessed: Jan. 2018.

[31] D. Carra and P. Michiardi, “Memory partitioning in memcached: An experimental performance analysis,” in *Communications (ICC), 2014 IEEE International Conference on*. IEEE, 2014, pp. 1154–1159.

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

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

[34] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger, “Evaluation techniques for storage hierarchies,” *IBM Syst. J.*, vol. 9, no. 2, pp. 78–117, Jun. 1970.

[35] Y. Zhong, X. Shen, and C. Ding, “Program locality analysis using reuse distance,” *ACM Trans. Program. Lang. Syst.*, vol. 31, no. 6, pp. 1–39, 2009.

[36] J. Wires, S. Ingram, Z. Drudi, N. J. Harvey, A. Warfield, and C. Data, “Characterizing storage workloads with counter stacks.” in *OSDI*, 2014, pp. 335–349.

[37] C. A. Waldspurger, N. Park, A. T. Garthwaite, and I. Ahmad, “Efficient mrc construction with shards.” in *FAST*, 2015, pp. 95–110.

[38] C. Waldspurger, T. Saemundsson, I. Ahmad, and N. Park, “Cache modeling and optimization using miniature simulations,” in *Proceedings of USENIX ATC*, 2017, pp. 487–498.

[39] X. Hu, X. Wang, L. Zhou, Y. Luo, C. Ding, and Z. Wang, “Kinetic modeling of data eviction in cache.” in *USENIX Annual Technical Conference*, 2016, pp. 351–364.

[40] Y.-J. Hong and M. Thottethodi, “Understanding and mitigating the impact of load imbalance in the memory caching tier,” in *Proceedings of the 4th annual Symposium on Cloud Computing*. ACM, 2013, p. 13.

[41] Y. Cheng, A. Gupta, and A. R. Butt, “An in-memory object caching framework with adaptive load balancing,” in *Proceedings of the Tenth European Conference on Computer Systems*, 2015, p. 4.

[42] J. Erman and K. K. Ramakrishnan, “Understanding the super-sized traffic of the super bowl,” in *Proceedings of the 2013 conference on Internet measurement conference*. ACM, 2013, pp. 353–360.

[43] H. C. Lim, S. Babu, and J. S. Chase, “Automated control for elastic storage,” in *Proceedings of the 7th international conference on Autonomic computing*. ACM, 2010, pp. 1–10.

[44] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, “Cloudscale: elastic resource scaling for multi-tenant cloud systems,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 5.

[45] Z. Xu, C. Stewart, N. Deng, and X. Wang, “Blending on-demand and spot instances to lower costs for in-memory storage,” in *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*. IEEE, 2016, pp. 1–9.

[46] C. X. Cai, G. Liang, and U. C. Kozat, “Load balancing and dynamic scaling of cache storage against zipfian workloads,” in *2014 IEEE International Conference on Communications (ICC)*, June 2014, pp. 4208–4214.

[47] P. Marchetta, J. Llorca, A. M. Tulino, and A. Pescapé, “Mc3: A cloud caching strategy for next generation virtual content distribution networks,” in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, May 2016, pp. 332–340.

[48] M. Garetto, E. Leonardi, and V. Martina, “A unified approach to the performance analysis of caching systems,” *ACM Trans. Model. Perform. Eval. Comput. Syst.*, vol. 1, no. 3, pp. 12:1–12:28, May 2016.

[49] G. Neglia, D. Carra, and P. Michiardi, “Cache Policies for Linear Utility Maximization,” *IEEE/ACM Transactions on Networking*, vol. 26, no. 1, pp. 302–313, 2018.

[50] D. Carra, G. Neglia, and P. Michiardi, “Elastic provisioning of cloud caches: a cost-aware TTL approach,” *CoRR*, vol. abs/1802.04696, 2018.

[51] N. Carlsson and D. Eager, “Worst-case bounds and optimized cache on mth request cache insertion policies under elastic conditions,” *Performance Evaluation*, vol. 127-128, pp. 70 – 92, 2018.

[52] A. Gupta, R. Krishnaswamy, A. Kumar, and D. Panigrahi, “Elastic caching,” in *Thirtieth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 2019.

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

Giovanni Neglia received the master’s degree in Electronic Engineering and the PhD degree in Telecommunications from the University of Palermo, Italy, in 2001 and 2005, respectively. He has been a researcher at Inria, Sophia Antipolis, France, since September 2008. In 2005, he was a research scholar with the University of Massachusetts, Amherst, visiting the Computer Networks Research Group. Before joining Inria, he was a post-doctorate with the University of Palermo and an external scientific advisor with the Maestro Team at Inria. His research is focused on modeling and performance evaluation of networks.

Pietro Michiardi received his M.S. in Computer Science from EURECOM and his M.S. in Electrical Engineering from Politecnico di Torino. Pietro received his Ph.D. in Computer Science from Telecom Paris, and his Habilitation from UNSA. Pietro is a Professor of Computer Science and head of the Data Science Department at EURECOM. In his work, Pietro blends theory and system research focusing on scalable machine learning algorithms. Pietro is interested in developing a theoretical understanding of the optimization process underlying many machine learning methods, in methodological aspects of computationally efficient Bayesian inference approaches, and their application in industrial domains such as the automotive and the IT infrastructure industries. In the past, Pietro worked on a wide range of research topics, including: computer networks and their performance evaluation, applied cryptography, applied game theory, distributed systems for content storage and distribution, and distributed data management systems.

Appendix

A. *Hypotheses of [27, Theorem 2.1]*

In this appendix we show that hypotheses (A2.1) – (A2.5) required in the first part of Theorem 2.1 in [27, p. 127] are satisfied for the specific problem.

We denote $X(n, T(n))$ simply as X_n to make the formulas more compact.

$$\begin{aligned}\mathbb{E}\big[X_n^2\big] &= \sum_{i=1}^{N} \frac{\lambda_i e^{-\lambda_i T(n)}}{\sum_{j=1}^{N} \lambda_j e^{-\lambda_j T(m)}}\, \mathbb{E}\Big[(\hat{\lambda}_i m_i - c_i)^2\Big] \\ &\le \sum_{i=1}^{N} \mathbb{E}\Big[(\hat{\lambda}_i m_i - c_i)^2\Big] \\ &= \sum_{i=1}^{N} (\lambda_i m_i - c_i)^2 + m_i^2 \mathrm{Var}(\hat{\lambda}_i) < \infty,\end{aligned}$$

where Var$(\cdot)$ denotes the variance of a random variable. Then, $\sup_n \mathbb{E}\big[X(n, T(n))^2\big] < \infty$ and (A2.1) holds.

$$\begin{aligned}&\mathbb{E}[X_n | T(0), X_1, \dots X_{n-1}] \\ &\quad= \sum_{i=1}^{N} \frac{\lambda_i e^{-\lambda_i T(n)}}{\sum_{j=1}^{N} \lambda_j e^{-\lambda_j T(n)}}\, \mathbb{E}\Big[\hat{\lambda}_i m_i - c_i\Big] \\ &\quad= \sum_{i=1}^{N} \frac{\lambda_i e^{-\lambda_i T(n)} (\lambda_i m_i - c_i)}{\sum_{j=1}^{N} \lambda_j e^{-\lambda_j T(n)}}. \\ &\quad\triangleq \bar{g}(T(n))\end{aligned}$$

The function $\bar{g}(\cdot)$ is continuous, then both (A2.2) and (A2.3) are satisfied with $\beta_n = 0$. As $\beta_n = 0$, (A2.5) ($\sum_n \epsilon(n) | \beta_n < \infty$) is also trivially satisfied.

Finally, assumption (A2.4) ($\sum_n \epsilon(n)^2 < \infty$) is explicitly required among the hypotheses of Proposition 1.

B. *Results with IRM*

In this section we investigate the performance of the update rule (8) under IRM traffic and in particular we show that it achieves minimal cost as indicated by Proposition 1. For this set of experiments, we consider a slightly different setting: we have a single, ideal TTL cache, i.e., a cache whose size depends on the value of the TTL, which can be arbitrarily large. We keep track of the instantaneous and average actual cache size (amount of space used by the stored objects), where the average is computed over an observation interval (e.g., one hour). We compute the cost due to storage as the cost of the average cache size. The input trace whose distribution has been showed in Fig.2 has been used to create an IRM trace, i.e., a trace where object popularity is the same as the original trace, but request arrival follows the IRM model.

Figure 15, left hand side, shows the value of the TTL over time. As the times goes by, the TTL converges to 1, that is the optimal value as it can be checked directly from (4). The right hand side of the figure shows the corresponding dynamics of the cache size.

Note that, once the TTL has reached a specific value, it remains stable, since the arrival process is stationary. Figure 16,

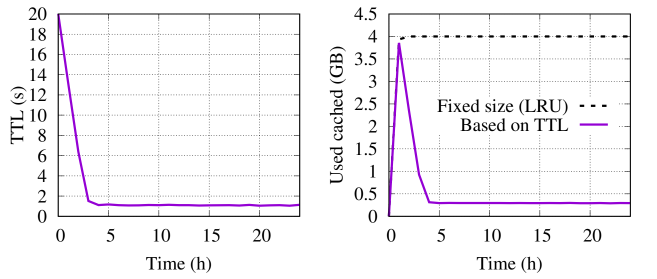

Fig. 15: Results with an IRM input trace: TTL over time (left), and average cache size (right).

left hand side, shows the total cost per hour of the TTL-based and fixed size caches. In order to verify that the TTL-based solution converges to the minimum cost, we have performed a set of experiments with fixed-size LRU caches, for different values of cache size, and record the cost per hour. Figure 16, right hand side, shows that the minimum cost is obtained with a cache with 0.3 GB, which is the value to which the TTL-based cache converges to.

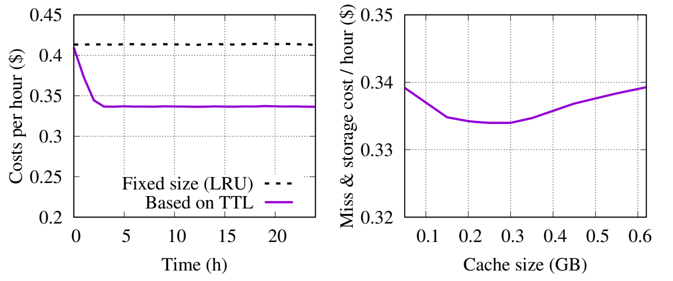

Fig. 16: Results with an IRM input trace: Cost per hour (miss and storage combined, left), and total cost for fixed-size LRU caches with different sizes (right).

In summary, the TTL-based solution represents a valid candidate for managing dynamic resizing of a cache.