Adaptive TTL-Based Caching for Content Delivery

Soumya Basu, Aditya Sundarrajan, Javad Ghaderi, Sanjay Shakkottai, *Fellow, IEEE* and Ramesh Sitaraman

***Abstract*—Content Delivery Networks (CDNs) cache and serve a majority of the user-requested content on the Internet. Designing caching algorithms that *automatically* adapt to the heterogeneity, burstiness, and non-stationary nature of real-world content requests is a major challenge and is the focus of our work. While there is much work on caching algorithms for stationary request traffic, the work on non-stationary request traffic is very limited. Consequently, most prior models are inaccurate for non-stationary production CDN traffic. We propose two TTL-based caching algorithms that provide provable performance guarantees for request traffic that is bursty and non-stationary. The first algorithm called d-TTL dynamically adapts a TTL parameter using stochastic approximation. Given a feasible target hit rate, we show that d-TTL converges to its target value for a general class of bursty traffic that allows Markov dependence over time and non-stationary arrivals. The second algorithm called f-TTL uses two caches, each with its own TTL. The first-level cache adaptively filters out non-stationary traffic, while the second-level cache stores frequently-accessed stationary traffic. Given feasible targets for both the hit rate and the expected cache size, f-TTL asymptotically achieves both targets. We evaluate both d-TTL and f-TTL using an extensive trace containing more than 500 million requests from a production CDN server. We show that both d-TTL and f-TTL converge to their hit rate targets with an error of about 1.3%. But, f-TTL requires a significantly smaller cache size than d-TTL to achieve the same hit rate, since it effectively filters out non-stationary content.**

***Index Terms*—TTL caches, Content Delivery Network, Adaptive caching, Actor-Critic Algorithm**

I. Introduction

By caching and delivering content to millions of end users around the world, content delivery networks (CDNs) [2] are an integral part of the Internet infrastructure. A large CDN such as Akamai [3] serves several trillion user requests a day from 170,000+ servers located in 1500+ networks in 100+ countries around the world. The majority of today's Internet traffic is delivered by CDNs. CDNs are expected to deliver nearly two-thirds of the Internet traffic by 2020 [4].

The main function of a CDN server is to cache and serve content requested by users. The effectiveness of a caching algorithm is measured by its achieved hit rate in relation to its cache size. There are two primary ways of measuring the hit rate. The *object hit rate (OHR)* is the fraction of the requested objects that are served from cache and the *byte hit rate (BHR)* is the fraction of the requested content bytes that are served from cache. We devise algorithms capable of operating with both notions of hit rate in our work.

The major technical challenge in designing caching algorithms for a modern CDN is *adapting* to the sheer heterogeneity of the content that is accessed by users. The accessed content falls into multiple traffic classes that include web pages, videos, software downloads, interactive applications, and social networks. The classes differ widely in terms of the object size distributions and content access patterns. The popularity of the content also varies by several orders of magnitude with some objects accessed millions of times (e.g, an Apple iOS download), and other objects accessed once or twice (e.g, a photo in a Facebook gallery). In fact, as shown in Figure 2, 70% of the objects served by a CDN server are only requested once over a period of multiple days! Further, the requests served by a CDN server can change rapidly over time as different traffic mixes are routed to the server by the CDN's load balancer in response to Internet events.

Request statistics clearly play a key role in determining the hit rate of a CDN server. However, when request patterns vary rapidly across servers and time, a one-size-fits-all approach provides inferior hit rate performance in a production CDN setting. Further, manually tuning the caching algorithms for each individual server to account for the varying request statistics is prohibitively expensive. Thus, our goal is to devise self-tuning caching algorithms that can automatically learn and adapt to the request traffic and provably achieve any feasible hit rate and cache size, even when the request traffic is bursty and non-stationary.

Our work fulfills a long-standing deficiency in the current state-of-art in the modeling and analysis of caching algorithms. Even though real-world CDN traffic is known to be heterogeneous, with bursty, non-stationary and transient request statistics, there are no known caching algorithms that provide theoretical performance guarantees for such traffic.[1] In fact, much of the known formal models and analyses assume that the traffic follows the *Independent Reference Model* (IRM).[2] However, when it comes to production traces such models lose their relevance. The following example highlights the stark inaccuracy of one popular model corroborating similar observations in [5]–[7], among others.

Deficiency of current models and analyses. Time-to-live (TTL)-based caching algorithms [8]–[13] use a TTL parameter to determine how long an object may remain in cache. TTL caches have emerged as useful mathematical tools to analyze the performance of traditional capacity-based caching algorithms such as LRU, FIFO, etc. The cornerstone of such

A short version of this work has appeared as a two-page extended abstract [1] in the Proceedings of ACM Sigmetrics, Urbana, IL, June 2017.

S. Basu and S. Shakkottai are with the Department of Electrical and Computer Engineering, The University of Texas at Austin, TX 78712. (E-mail: basusoumya@utexas.edu and shakkott@austin.utexas.edu).

A. Sundarrajan and R. Sitaraman are with the College of Information and Computer Sciences, University of Massachusetts Amherst, MA 01003. (E-mail: asundar@cs.umass.edu and ramesh@cs.umass.edu)

J. Ghaderi is with the Department of Electrical Engineering, Columbia University, New York, NY 10027. (E-mail: jghaderi@ee.columbia.edu)

[1] We note that LRU cache has been previously studied under non-stationary models, e.g. box model [5], shot noise model [6].

[2] The inter arrival times are i.i.d. and the object request on each arrival are chosen independently from the same distribution.

analyses is the work by Fagin [14] that relates the cache hit rate with the expected cache size and characteristic time for IRM traffic, which is also popularly known as Che's approximation after the follow-up work [15]. Under this approximation, a LRU cache has the same expected size and hit rate as a TTL-cache with the TTL value equal to its characteristic time. Che's approximation is known to be accurate in cache simulations that use synthetic IRM traffic and is commonly used in the design of caching algorithms for that reason [12], [16]–[19].

However, we show that Che's approximation produces erroneous results for actual production CDN traffic that is neither stationary nor IRM across the requests.[3] We used an extensive 9-day request trace from a production server in Akamai's CDN and derived TTL values for multiple hit rate targets using Che's approximation. We then simulated a cache with those TTL values on the production traces to derive the *actual* hit rate that was achieved. For a target hit rate of 60%, we observed that a fixed-TTL algorithm that uses the TTL computed from Che's approximation achieved a hit rate of 68.23% whereas the dynamic TTL algorithms proposed in this work achieve a hit rate of 59.36% (see Section VI-E for a complete discussion). This difference between the target hit rate and that achieved by fixed-TTL highlights the inaccuracy of state-of-the-art theoretical modeling on production traffic.

A. Main Contributions

We propose two TTL-based algorithms: d-TTL (for "dynamic TTL") and f-TTL (for "filtering TTL") that provably achieve a target cache hit rate and cache size. Rather than statically deriving the required TTL values by inferring the request statistics, our algorithms *dynamically* adapt the TTLs to the request patterns. To more accurately model real traffic, we allow the request traffic to be non-independent and have non-stationary components. Further, we allow content to be classified into *types*, where each type has a target hit rate (OHR or BHR) and an average target cache size. In practice, a type can consist of all objects of a specific kind from a specific provider, e.g. CNN webpages, Facebook images, CNN video clips, etc. Our main contributions are as follows:

1) **d-TTL: A one-level TTL algorithm.** Algorithm d-TTL maintains a single TTL value for each type, and dynamically adapts this value upon each arrival (new request) of an object of this type. Given a hit rate that is "feasible" (i.e. there exists a static genie-settable TTL parameter that can achieve this hit rate), we show that d-TTL almost surely converges to this target hit rate. Our result holds for a general class of bursty traffic (allowing Markov dependence over time), and even in the presence of non-stationary arrivals. To the best of our knowledge, this is the first adaptive TTL algorithm that can provably achieve a target hit rate with such stochastic traffic.

However, our empirical results show that non-stationary and unpopular objects can contribute significantly to the cache size, while they contribute very little to the cache hit rate (heuristics that use Bloom filters to eliminate such traffic [20] support this observation).

2) **f-TTL: A two-level TTL algorithm.** The need to achieve both a target hit rate and a target cache size motivates the f-TTL algorithm. f-TTL comprises a pair of caches: a lower-level adaptive TTL cache that filters rare objects based on arrival history, and a higher-level adaptive TTL cache that stores filtered objects. We design an adaptation mechanism for a pair of TTL values (higher-level and lower-level) per type, and show that we can asymptotically achieve the desired hit rate (almost surely), under similar traffic conditions as with d-TTL. If the *stationary* part of the traffic is Poisson, we have the following stronger property. Given any feasible (hit rate, expected cache size) pair,[4] the f-TTL algorithm asymptotically achieves a corresponding pair that dominates the given target.[5] Importantly, with non-stationary traffic, the two-level adaptive TTL strictly outperforms the one-level TTL cache with respect to the expected cache size.

Our proofs use a two-level stochastic approximation technique (along with a latent observer idea inspired from actor-critic algorithms [21]), and provide the first theoretical justification for the deployment of two-level caches such as ARC [22] in production systems with non-stationary traffic.

3) **Implementation and empirical evaluation:** We implement both d-TTL and f-TTL and evaluate them using an extensive 9-day trace consisting of more than 500 million requests from a production Akamai CDN server. We observe that both d-TTL and f-TTL adapt well to the bursty and non-stationary nature of production CDN traffic. For a range of target object hit rates, both d-TTL and f-TTL converge to those targets with an error of about 1.3%. For a range of target byte hit rates, both d-TTL and f-TTL converge to those targets with an error that ranges from 0.3% to 2.3%. While the hit rate performance of both d-TTL and f-TTL are similar, f-TTL shows a distinct advantage in cache size due to its ability to filter out non-stationary traffic. In particular, f-TTL requires a cache that is 49% (resp., 39%) smaller than d-TTL to achieve the same object (resp., byte) hit rate. This renders f-TTL useful to CDN settings where large amounts of non-stationary traffic can be filtered out to conserve cache space while also achieving target hit rates.

Finally, from a practitioner's perspective, this work has the potential to enable new CDN pricing models. CDNs typically do not charge content providers on the basis of a guaranteed hit rate performance for their content, nor on the basis of the cache size that they use. Such pricing models have desirable properties, but do not commonly exist, in part, because current caching algorithms cannot provide such guarantees with low overhead. Our caching algorithms are the first to provide a theoretical guarantee on hit rate for each content provider, while controlling the cache space that they can use. Thus, our work removes a technical impediment to hit rate and cache space based CDN pricing.

[3] Under the assumption that traffic is IRM with memoryless arrival we compute the TTL/characteristic time that corresponds to the target hit rate. Other Che's approximation schemes, such as [6], can potentially increase the accuracy at a higher computation cost, thus leading to difficulties with large-scale deployments.

[4] Feasibility here is with respect to any static two-level TTL algorithm that achieves a (target hit rate, target expected cache size) pair.

[5] A pair dominates another pair if hit rate is at least equal to the latter and expected size is at most equal to the latter.

B. Notations

Some of the basic notations used in this paper are as follows. Bold font characters indicate vector variables and normal font characters indicate scalar variables. We note $(x)^+ = \max(0, x)$, $\mathbb{N} = \{1, 2, \dots\}$, and $[n] = \{1, 2, \dots, n\}$. The equality among two vectors means component-wise equality holds. Similarly, inequality among two vectors (denoted by $\preccurlyeq$) means the inequality holds for each component separately. We use the term 'w.p.' for '*with probability*', 'w.h.p.' for '*with high probability*', 'a.s.' for '*almost surely*', and 'a.a.s.' for '*asymptotically almost surely*'.

II. System Model and Definitions

Every CDN server implements a cache that stores objects requested by users. When a user's request arrives at a CDN server, the requested object is served from its cache, if that object is present. Otherwise, the CDN server fetches the object from a remote origin server that has the original content and then serves it to the user. In addition, the CDN server may place the newly-fetched object in its cache. In general, a caching algorithm decides which object to place in cache, how long objects need to be stored in cache, and which objects should be evicted from cache.

When the requested object is found in cache, it is a *cache hit*, otherwise it is a *cache miss*. A cache hit is desirable since the object can be retrieved locally from the proximal server and returned to the user with low latency. Additionally, it is often beneficial to maintain state (metadata such as the URL of the object or an object ID) about a recently evicted object for some period of time. Then, we experience a *cache virtual hit* if the requested object is not in cache but its metadata is in cache. Note that the metadata of an object takes much less cache space than the object itself.

Next, we formally describe the request arrival model, and the performance metrics: object (byte) hit rate and expected cache size, and formally state the objective of the paper.

A. Content Request Model

There are different types of content hosted on modern CDNs. A content type may represent a specific genre of content (videos, web pages, etc.) from a specific content provider (CNN, Facebook, etc.). A single server could be shared among dozens of content types. A salient feature of content hosted on CDNs is that the objects of one type can be very different from the objects of another type, in terms of their popularity characteristics, request patterns and object size distributions. Most content types exhibit a long tail of popularity where there is a smaller set of *recurring objects* that demonstrate a stationary behavior in their popularity and are requested frequently by users, and a larger set of *rare objects* that are unpopular and show a high degree of non-stationarity. Examples of rare objects include those that are requested infrequently or even just once, a.k.a. one-hit wonders [23]. Another example is an object that is rare in a temporal sense and is frequently accessed within a small time window, but is seldom accessed again. Such a bursty request pattern can occur during flash crowds [24]. In this section, we present a content request model that captures these characteristics.

1) Content Description:

We consider T different types of content where each type consists of both recurring objects and rare objects. The set of recurring objects of type t is denoted by $\mathcal{K}_t$ with $|\mathcal{K}_t| = K_t$ different objects, and the set of rare objects of type t is denoted by $\mathcal{R}_t$. The entire universe of objects is represented as $\mathcal{U} \equiv \cup_{t\in T}(\mathcal{K}_t \cup \mathcal{R}_t)$, and the set of recurring objects is represented by a finite set $\mathcal{K} \equiv \cup_{t\in T}\mathcal{K}_t$. Let $K \equiv |\mathcal{K}| = \sum_{t\in T} K_t$. In our model, the number of types T is finite. For each type $t \in [T]$ there are finitely many recurring objects, i.e. K_t is finite. However, the rare objects are allowed to be (potentially) countably infinite in number.

Each object $c \in \mathcal{U}$ is represented by a tuple, $c = (c_i, c_{typ}, c_m)$, and its meta-data is represented as $\tilde{c} = (c_i, c_{typ})$. Here, c_i is the unique *label* for the object (e.g., its URL), c_{typ} is the *type* that the object belongs to, and c_m is the actual body of the object c. If $c \in \mathcal{K}$, then w.l.o.g., we can index $c_i = k$ for some $k \in \{1, \dots K\}$. The object meta-data, $\tilde{c} = (c_i, c_{typ})$, is assumed to have negligible size, and the size of object c is denoted as $w_c = |c_m|$ (in bytes). Note that the object meta-data can be fully extracted from the incoming request. In our model, for all objects $c \in \mathcal{U}$, their sizes are uniformly bounded as $w_c \le w_{\max}$. Moreover, we assume, for each type $t \in [T]$, all rare objects of type t have equal size $\bar{w}_t$.[6]

2) General Content Request Model:

We denote the object requested on l-th arrival as

$$c(l) \equiv (\text{label} : c_i(l),\ \text{type} : c_{typ}(l),\ \text{size} : w(l)).$$

Further, let $A(l)$ be the arrival time of the l-th request, and $X(l)$ be the l-th inter-arrival time, i.e., $X(l) = A(l) - A(l-1)$. We define a random variable $Z(l)$ which specifies the label of the l-th request if the request is for a recurrent object, and specifies its type if the request is for a rare object (i.e. $Z(l) = c_i(l)$ if $c(l) \in \mathcal{K}$, and $Z(l) = c_{typ}(l)$ otherwise). We also require the following two definitions:

$$\begin{aligned} X_{pre}(l) &= \min\{A(l) - A(l') : l' < l, c(l') = c(l)\} \\ X_{suc}(l) &= \min\{A(l') - A(l) : l' > l, c(l') = c(l)\}, \end{aligned}$$

hence $X_{pre}(l)$ and $X_{suc}(l)$ represent the preceding and succeeding inter-arrival time for the object requested on l-th arrival, respectively. By convention, $\min\{\emptyset\} = \infty$.

For any constant $R > 0$, and $l \ge 1$, define the set of objects that arrived within R units of time from the l-th arrival, as

$$\mathcal{A}(l; R) = \{c(l') : l' \in \mathbb{N}, A(l') \le A(l) - R\}.$$

We also define, for all $R > 0$ and type $t \in [T]$, the *bursty arrival indicator* $\beta_t(l; R)$ as the indicator function of the event: (1) the l-th request is for some rare object c of type t, *and* (2) the previous request of the same rare object c happened (strictly) less than R units of time earlier. Specifically, $\beta_t(l; R) = \mathbb{1}(c(l) \in \mathcal{R}_t, X_{pre}(l) < R)$. Note that

[6]This could be relaxed to *average* size for type t rare objects, as long as the average size over a large enough time window has $o(1)$ difference from the average, w.p. 1.

$\beta_t(l;R)$ does not depend on a specific $c \in \mathcal{R}_t$, but accumulates over all rare objects of type t.

The general content request model is built on a Markov renewal process $(A(l), Z(l))_{l\in\mathbb{N}}$ [25] (to model the stationary components and potential Markovian dependence on the object requests), followed by rare object labeling to model non-stationary components. Formally, our general content request model, parameterized by constant $R > 0$, is as follows.

Assumption 1.1. General Content Request Model (R):

- **Markov renewal process** $(A(l), Z(l))_{l\in\mathbb{N}}$

(i) The inter-arrival times $X(l) = A(l) - A(l-1)$, $l \in \mathbb{N}$, are identically distributed, independently of each other and $Z(l)$. The inter-arrival time distribution follows a probability density function (p.d.f.), $f(x)$ which is *absolutely continuous* w.r.t a Lebesgue measure on $(\mathbb{R}, +)$ and has *simply connected support*, i.e. if $f(x) > 0, f(y) > 0$ then $f(z) > 0$ for all $z \in (x, y)$. The inter-arrival time has a *nonzero finite mean* denoted by $1/\lambda$.

(ii) The process $Z(l)$ is a Markov chain over $(K + T)$ states indexed by $1, \cdots, K + T$. The first K states represent the K recurring objects. The rare objects (possibly infinite in number) are grouped according to their types, thus producing the remaining T states, i.e, the states $K + 1, \cdots, K + T$ represent rare objects of types $1, \cdots, T$, respectively. The transition probability matrix of the Markov chain $Z(l)$ is given by P, where

$$P(c, c') := P(Z(l) = c' | Z(l-1) = c), \ \forall c, c' \in [K+T].$$

We assume that the diagonal entries $P(c, c) > 0$, hence the Markov chain is aperiodic. Also the Markov chain is assumed to be irreducible, thus it possesses a stationary distribution denoted by $\boldsymbol{\pi}$.

- **Object labeling process** $c(l)$

(i) *Recurrent objects:* On the l-th arrival, if the Markov chain $Z(l)$ moves to a state $k \in [K]$, the arrival is labeled by the recurrent object k, i.e. $c_i(l) = k$.

(ii) *Rare objects:* On the l-th arrival, if the Markov chain $Z(l)$ moves to a state $K+t$, $t \in [T]$, the arrival is labeled by a rare object of type t, chosen from $\mathcal{R}_t$ such that the label assignment has no stationary behavior in the time-scale of $O(1)$ arrivals and it shows a *rarity* behavior in large time-scales. Formally, for each type t, $\sum_{l'=1}^{l} \beta_t(l'; R)$ is maintained for $l \geq 1$ and an arbitrary constant $a_t > 0$ is fixed. On l-th arrival, given $Z(l) = K + t$,
  - if $\sum_{l'=1}^{l} \beta_t(l'; R) \leq a_t\sqrt{l}$: select any rare object of type t (arbitrarily), i.e., $c_i(l) \in \mathcal{R}_t$
  - else: select any rare object of type t that was not requested within R time units, i.e., $c_i(l) \in \mathcal{R}_t \backslash \mathcal{A}(l; R)$.

The above labeling of rare objects respects a more general *R-rarity condition* defined below (sufficient for our theoretical results):

Definition 1 (R-rarity condition)**.** For any type $t \in T$, and a finite $R > 0$,

$$\forall N_m^t = \omega(\sqrt{m}), \lim_{m\to\infty} \frac{1}{N_m^t} \sum_{l=m}^{m+N_m^t} \beta_t(l; R) = 0, \text{ w.p. 1.} \quad (1)$$

For any type t, let α_t be the aggregate fraction of total request arrivals for rare objects of type t in the long run. Note that by the Markov renewal construction, $\alpha_t = \pi_{(K+t)}$, where $\boldsymbol{\pi}$ is the stationary distribution of the process $Z(l)$. If $\alpha_t > 0$, then for the the R-rarity condition to hold, it is sufficient to have infinitely many rare objects of the same type t (over an infinite time horizon).

Remark 1 (Comment on the R-rarity condition). The "R-rarity condition" states that asymptotically (i.e, after m-th arrival, for large enough m) for each type t, requests for rare objects of that type can still arrive as bursty arrivals (i.e., request for any particular *rare object* is separated by less than R time units), as long as over large time windows (windows of size $N_m^t = \omega(\sqrt{m})$) the number of such bursty arrivals becomes infrequent (i.e. $o(N_m^t)$ w.p. 1). Note that the definition of bursty arrival and the associated "R-rarity condition" is not specified for a particular constant R, but is *parameterized by* R and we shall specify the specific value later. If R-rarity condition holds then R'-rarity condition also holds for any $R' \in [0, R)$, which easily follows from the definition.

Remark 2 (Relevance of the R-rarity condition). The condition (1) is fairly general, as at any point in time, no matter how large, it allows the existence of rare objects which may exhibit burstiness for small time windows. Trivially, if the inter-arrival time of each rare object is greater than R, then R-rarity condition is satisfied. More interestingly, the following real-world scenarios satisfy the (rare) object labeling process in Assumption 1.1.

- *One-hit wonders [23].* For each type t, a constant fraction α_t of total arrivals are for rare objects that are requested *only once*. As the indicator $\beta_t(l; R)$ is zero for the first and only time that an object is requested, $\sum_{l'=1}^{l} \beta_t(l'; R) = 0$, for all $l \geq 1$ and type $t \in [T]$.
- *Flash crowds [24].* Constant size bursts (i.e. a collection of $O(1)$ number of bursty arrivals) of requests for rare objects may occur over time, with $O(\sqrt{\tau})$ number of such bursts up to time τ. This allows for infinitely many such bursts. In this scenario, almost surely, for any type t, $\sum_{l'=1}^{l} \beta_t(l'; R) = O(\sqrt{l})$. Therefore, it is a special case of our model.

Remark 3 (Generalization of rare object labeling). In our proofs we only require that the R-rarity condition holds, for a certain value of R. Therefore, we can generalize our result to any rare object labeling process that satisfies the R-rarity condition (Definition 1), for that specific value of R. Further, it is possible to weaken the rarity condition by requiring the condition to hold with high probability instead of w.p. 1.

Remark 4 (Relevance of the content request model). Most of the popular inter-arrival time distributions, e.g., Exponential, Phase-type, Weibull, satisfy the inter-arrival model in Assumption 1.1. Moreover, it is easy to see that any i.i.d. distribution for content popularity, including Zipfian distribution, is a special case of our object labeling process. In fact, the labeling process is much more general in the sense that it can capture the influence of different objects on other objects, which may span across various types.

3) Special case: Poisson Arrival with Independent Labeling:

We next consider a specific model for the arrival process which is a well-studied special case of Assumption 1.1. We will later show that under this arrival process we can achieve stronger guarantees on the system performance.

Assumption 1.2. Poisson Arrival with Independent Labeling:

- The inter arrival times are i.i.d. and *exponentially* distributed with rate $\lambda > 0$.
- The labels for the recurring objects are determined independently. At each request arrival, the request is labeled a recurring object c with probability π_c, and is labeled a rare object of type t with probability α_t, following the same rare object labeling process for rare objects of type t, as in Assumption 1.1.
- For each recurrent object c, its size is given by w_c, which is non-decreasing w.r.t. probability π_c and at most $w_{\max}$. For each type $t \in [T]$, all rare objects of type t have size $\bar{w}_t$.

B. Object (Byte) Hit Rate and Normalized Size

There are two common measures of hit rate. The **object hit rate (OHR)** is the fraction of requests that experience a cache hit. The **byte hit rate (BHR)** is the fraction of requested bytes that experience a cache hit. BHR measures the traffic reduction between the origin and the cache severs. Both measures can be computed for a single object or a group of objects. Here, we consider all objects of the same type as being part of a group.

We formally define OHR and BHR as follows. Given a caching algorithm, define $Y(l) = 1$ if the l-th arrival experiences a cache hit and $Y(l) = 0$ otherwise. Also, let $\mathcal{C}(\tau)$ be the set of objects in the cache at time τ, for $\tau \geq 0$.

Definition 2. The OHR for each type $t \in T$ is defined as

$$h_t = \liminf_{\tau\to\infty} \frac{\sum_{l:A(l)\leq\tau} \mathbb{1}\left(c_{typ}(l) = t, Y(l) = 1\right)}{\sum_{l:A(l)\leq\tau} \mathbb{1}\left(c_{typ}(l) = t\right)}$$

Definition 3. The BHR for each type $t \in T$ is defined as

$$h_t = \liminf_{\tau\to\infty} \frac{\sum_{l:A(l)\leq\tau} w(l)\mathbb{1}\left(c_{typ}(l) = t, Y(l) = 1\right)}{\sum_{l:A(l)\leq\tau} w(l)\mathbb{1}\left(c_{typ}(l) = t\right)}$$

The performance of a caching algorithm is often measured using its hit rate curve (HRC) that relates the hit rate that it achieves to the cache size (in bytes) that it requires. In general, the hit rate depends on the request arrival rate which in turn affects the cache size requirement. We define a new metric called the ***normalized size*** which is defined as the ratio of the time-average cache size (in bytes) utilized by the object(s) over the time-average arrival rate (in bytes/sec) of the object(s). The normalized size is formally defined below.

Definition 4. For a caching algorithm, and each type $t \in T$, the ***normalized size*** for type t is defined as

$$s_t = \limsup_{\tau\to\infty} \frac{\int_{\tau'=0}^{\tau} \sum_{c\in\mathcal{C}(\tau')} w_c \mathbb{1}(c_{typ} = t) d\tau'}{\sum_{l:A(l)\leq\tau} w(l)\mathbb{1}\left(c_{typ}(l) = t\right)}$$

Remark 5. Dividing both the numerator and the denominator by τ gives the interpretation of the normalized size as the average cache size utilized by the objects of type t normalized by their aggregate arrival rate. For example, if a CDN operator wants to allocate an expected cache size of $100GB$ for type t and its arrival rate is known to be $10GB/sec$, then the corresponding normalized size is $\frac{100GB}{10GB/sec} = 10sec$.

C. Design Objective

The fundamental challenge in cache design is striking a balance between two conflicting objectives: minimizing the cache size requirement and maximizing the cache hit rate. In addition, it is desirable to allow different Quality of Service (QoS) guarantees for different types of objects, i.e., different *cache hit rates* and *size targets* for different types of objects. For example, a lower hit rate that results in a higher response time may be tolerable for a software download that happens in the background. But, a higher hit rate that results in a faster response time is desirable for a web page that is delivered in real-time to the user.

In this work, **our objective** is to tune the TTL parameters to asymptotically achieve a *target hit rate vector* $\mathbf{h}^*$ and a (feasible) *target normalized size vector* $\boldsymbol{s}^*$, without the prior knowledge of the content request process. The t-th components of $\mathbf{h}^*$ and $\boldsymbol{s}^*$, i.e., h_t^* and s_t^* respectively, denote the target hit rate and the target normalized size for objects of type $t \in [T]$.

A CDN operator can group objects into types in an arbitrary way. If the objective is to achieve an overall hit rate and cache size, all objects can be grouped into a single type. It should also be noted that the algorithms proposed in this work *do not* try to achieve the target hit rate with the smallest cache size; this is a non-convex optimization problem that is not the focus of this work. Instead, we only try to achieve a *given* target hit rate and target normalized size.

III. Adaptive TTL-based Algorithms

A TTL-based caching algorithm works as follows. When a new object is requested, the object is placed in cache and is associated with a time-to-live (TTL) value. If no new requests are received for that object, the TTL value is decremented in real-time and the object is evicted when the TTL becomes zero.[7] If a cached object is requested, the TTL is reset to its original value. In a TTL cache, the TTL helps balance the cache size and hit rate objectives. When the TTL increases, the object stays in cache for a longer period of time, increasing the cache hit rate, at the expense of a larger cache size. The opposite happens when TTL decreases.

We propose two adaptive TTL algorithms. First, we present a dynamic TTL algorithm (d-TTL) that adapts its TTL to achieve a target hit rate $\mathbf{h}^*$. While d-TTL does a good job of achieving the target hit rate, it does this at the expense of caching rare and unpopular recurring content for an extended period of time, thus causing an increase in cache size without any significant contribution towards the cache hit rate. We present a second adaptive TTL algorithm called filtering TTL (f-TTL) that filters out rare content to achieve the target hit rate with a smaller cache size. To the best of our knowledge, both

[7]The TTL-based cache presented here is known as reset-TTL.

d-TTL and f-TTL are the first adaptive TTL-based caching algorithms that are able to achieve a target hit rate $\mathbf{h}^*$ and a feasible target normalized size $\mathbf{s}^*$ for non-stationary traffic.

A. Dynamic TTL (d-TTL) Algorithm

We propose a dynamic TTL algorithm, d-TTL, that adapts a TTL parameter on each arrival to achieve a target hit rate $\mathbf{h}^*$.

1) Structure: The d-TTL algorithm consists of a single TTL cache $\mathcal{C}$. It also maintains a *TTL vector* $\boldsymbol{\theta}(l) \in \mathbb{R}_+^T$, at the time of the l-th arrival, where $\theta_t(\cdot)$ represents the TTL value for type t. Every object c present in the cache $\mathcal{C}$, has a timer ψ_c^0 that encodes its remaining TTL and is decremented in real time. On the l-th arrival, if the requested object c of type t is present in cache, $\theta_t(l)$ is decremented, and if the requested object c of type t is not present in cache, object c is fetched from the origin, cached in the server and $\theta_t(l)$ is incremented. In both cases, ψ_c^0 is set to the updated timer $\theta_t(l+1)$ until the object is re-requested or evicted. As previously discussed, object c is evicted from the cache when $\psi_c^0 = 0$.

2) Key Insights: To better understand the dynamic TTL updates, we consider a simple scenario where we have unit sized objects of a single type and a target hit rate h^*.

Adaptation based on stochastic approximation. Consider a TTL parameter θ. Upon a cache miss, θ is incremented by ηh^* and upon a cache hit, θ is decremented by $\eta(1-h^*)$, where $\eta > 0$ is some positive step size. More concisely, θ is changed by $\eta(h^* - Y(l)))$, where $Y(l) = 1$ upon a cache hit and $Y(l) = 0$ upon a cache miss. If the expected hit rate under a fixed TTL value θ is h, then the expected change in the value of θ is given by $\eta((1-h)h^* - h(1-h^*))$. It is easy to see that this expected change approaches 0, as h approaches h^*. In a dynamic setting, $Y(l)$ provides a noisy estimate of h. However, by choosing *decaying step size*, i.e. on l-th arrival $\eta = \eta(l) = \frac{1}{l^\alpha}$, for $\alpha \in (0.5, 1]$, we can still ensure convergence, by using results from stochastic approximation theory [26].

Truncation in presence of rare objects. In some scenarios, the target hit rate h^* may be unattainable due to the presence of rare objects. Indeed, in the 9-day trace used in our paper, around 4% of the requests are for one-hit wonders. Clearly, in this scenario, a hit rate of over 96% is unachievable. Whenever h^* is unattainable, θ diverges with the above adaptation. Therefore, under *unknown amount of rare traffic* it becomes necessary to **truncate** θ with a large but finite value L to make the algorithm robust.

3) Adapting $\boldsymbol{\theta}(l)$: Following the above discussion, we restrict the TTL value $\boldsymbol{\theta}(l)$ to $\boldsymbol{\theta}(l) \preceq \boldsymbol{L}$.[8] Here $\mathbf{L}$ is the truncation parameter of the algorithm and an increase in $\mathbf{L}$ increases the achievable hit rate (see Section IV for details). For notational similarity with f-TTL, we introduce a latent variable $\boldsymbol{\vartheta}(l) \in \mathbb{R}^T$ where $\boldsymbol{\vartheta}(l) \in [0,1]$. Without loss of generality, instead of adapting $\boldsymbol{\theta}(l)$, we dynamically adapt each component of $\boldsymbol{\vartheta}(l)$ and set $\theta_t(l) = L_t \vartheta_t(l)$, where $\vartheta_t(\cdot)$ is the latent variable for objects of type t. The d-TTL algorithm is presented in Algorithm 1, where the value of $\boldsymbol{\theta}(l)$ dynamically changes according to Equation (2).

[8]This gives an upper bound, typically a large one, over the size of the cache. Further it can control the staleness of objects.

Algorithm 1 Dynamic TTL (d-TTL)

Input:
Target hit rate $\mathbf{h}^*$, TTL upper bound $\mathbf{L}$.
For l-th request, $l \in \mathbb{N}$, object $c(l)$, size $w(l)$ & type $t(l)$.
Output: Cache requested object using dynamic TTL, $\boldsymbol{\theta}$.

Initialize: Latent variable $\boldsymbol{\vartheta}(0) = \mathbf{0}$.
for all $l \in \mathbb{N}$ **do**
  if Cache hit, $c(l) \in \mathcal{C}$ **then**
    $Y(l) = 1$
  else Cache miss
    $Y(l) = 0$
  Update TTL $\theta_{t(l)}(l)$:

$$\begin{aligned} \vartheta_{t(l)}(l+1) &= \mathcal{P}_{[0,1]}\left(\vartheta_{t(l)}(l) + \eta(l)\widehat{w}(l)\left(h^*_{t(l)} - Y(l)\right)\right) \\ \theta_{t(l)}(l+1) &= L_{t(l)}\vartheta_{t(l)}(l+1) \end{aligned} \tag{2}$$

  where,
    $\eta(l) = \frac{\eta_0}{l^\alpha}$ is a decaying step size for $\alpha \in (1/2, 1)$,
    $\mathcal{P}_{[0,1]}(x) = \min\{1, \max\{0, x\}\}$,
    $\widehat{w}(l) = 1$ for OHR and $w(l)$ for BHR.

  Cache c with TTL $\psi^0_{c(l)} = \theta_{t(l)}(l+1)$ in $\mathcal{C}$.

B. Filtering TTL (f-TTL) Algorithm

Although the d-TTL algorithm achieves the target hit rate, it could lead to cache sizes that are excessively large. d-TTL could still cache rare and unpopular content that contribute to a non-negligible portion of the cache size (for example one-hit wonders still enter the cache while not providing any cache hit). We propose a two-level filtering TTL algorithm (f-TTL) that efficiently filters non-stationary content to achieve the target hit rate along with a target normalized size (Def. 4).

1) Structure: The two-level f-TTL algorithm maintains two caches: a higher-level (or deep) cache $\mathcal{C}$ and a lower-level cache $\mathcal{C}_s$. The higher-level cache (deep) cache $\mathcal{C}$ behaves similar to the single-level cache in d-TTL (Algorithm 1), whereas the lower-level cache $\mathcal{C}_s$ ensures that cache $\mathcal{C}$ stores mostly stationary content. Cache $\mathcal{C}_s$ does so by filtering out rare and unpopular objects, while suitably retaining bursty objects. To facilitate such filtering, it uses additional sub-level caches: *shadow cache* and *shallow cache*, each with their own dynamically adapted TTLs. The TTL value associated with the *shadow cache* is equal to the TTL value of deep cache $\mathcal{C}$, whereas the TTL associated with the *shallow cache* is smaller.

TTL timers for f-TTL. The complete algorithm for f-TTL is given in Algorithm 2. f-TTL maintains a time varying TTL-value $\boldsymbol{\theta}^s(l)$ for the *shallow cache*, along with a TTL value $\boldsymbol{\theta}(l)$ for both the *deep and shadow caches*. Every object c present in f-TTL has an *exclusive* TTL tuple $(\psi_c^0, \psi_c^1, \psi_c^2)$ indicating remaining TTL for that specific object: ψ_c^0 for deep cache $\mathcal{C}$, ψ_c^1 for the shallow cache of $\mathcal{C}_s$, and ψ_c^2 for the shadow cache of $\mathcal{C}_s$. Object c is evicted from $\mathcal{C}$ (resp., $\mathcal{C}_s$) when ψ_c^0 (resp., ψ_c^1) becomes 0. Further, the metadata $\tilde{c}$ is evicted from $\mathcal{C}_s$ when ψ_c^2 equals 0.

Suppose on the l-th arrival, the request is for object $c(l)$ (of type $t(l)$ and size $w(l)$). Let $c(l) = c$ and $t(l) = t$. The algorithm first updates the two TTL values to $\boldsymbol{\theta}(l+1)$ and

$\boldsymbol{\theta}^s(l+1)$, according to the update rules which will be described shortly. Then, it performs one of the operations below.

Cache hit: If a cache hit occurs, i.e., c is either in the deep cache $\mathcal{C}$ or in the shallow cache of $\mathcal{C}_s$, f-TTL caches object c in the deep cache $\mathcal{C}$ with TTL $\theta_t(l+1)$, thus setting the TTL tuple to $(\theta_t(l+1), 0, 0)$. Further, if c was in the shallow cache at the time of the cache hit, the object c and its metadata $\tilde{c}$ are removed from the shallow cache and the shadow cache of $\mathcal{C}_s$, respectively. [lines 12-15 in Algorithm 2].

Cache miss: If both object c and its meta-data $\tilde{c}$ are absent from $\mathcal{C}$ and $\mathcal{C}_s$ resp., we have a cache miss. In this event, f-TTL caches object c in the shallow cache of $\mathcal{C}_s$ with TTL $\theta^s_t(l+1)$ and its meta data $\tilde{c}$ in the shadow cache of $\mathcal{C}_s$ with TTL $\theta_t(l+1)$; i.e. the TTL tuple is set to $(0, \theta^s_t(l+1), \theta_t(l+1))$ [lines 19-20 in Algorithm 2].

Cache virtual hit: Finally, if $\tilde{c}$ is in the shadow cache of $\mathcal{C}_s$ but object c is not in the shallow cache of $\mathcal{C}_s$, we have a cache virtual hit. In this event, f-TTL caches c in the deep cache $\mathcal{C}$ with TTL tuple $(\theta_t(l+1), 0, 0)$, and evicts $\tilde{c}$ from $\mathcal{C}_s$ [lines 16-18 in Algorithm 2].

2) Key Insights: We pause here to provide the essential insights behind the structure and adaptation rules in f-TTL.

Normalized size of the f-TTL algorithm. We begin by characterizing the normalized size of the different content types in the f-TTL algorithm. For the l-th request arrival, define $\hat{s}(l)$ to be the time that the requested object will spend in the cache until either it is evicted or the same object is requested again, whichever happens first. We call $\hat{s}(l)$ the normalized size of the l-th arrival. Therefore, the contribution of the l-th request toward the cache size is $w(l)\hat{s}(l)$, where $\hat{s}(l) = \min\{X_{suc}(l), \theta^s_{t(l)}(l+1)\}$ for cache miss and $\hat{s}(l) = \min\{X_{suc}(l), \theta_{t(l)}(l+1)\}$ for cache hit/virtual hit. Then the normalized size, defined in Def. 4, can be equivalently characterized as

$$s_t = \limsup_{\tau\to\infty} \frac{\sum_{l:A(l)<\tau} w(l)\hat{s}(l)\mathbb{1}(c_{typ}(l)=t)}{\sum_{l:A(l)<\tau} w(l)\mathbb{1}(c_{typ}(l)=t)}, \forall t \in [T]. \tag{4}$$

To explain the key insights, we consider a simple scenario: single content type, unit sized objects, hit rate target h^* and normalized size target s^*.

Shadow cache for filtering rare objects. The shadow cache and shallow cache in $\mathcal{C}_s$ play complementary roles in efficiently filtering out rare and unpopular objects. By storing the meta-data (with negligible size) with TTL θ upon a new arrival, the shadow cache simulates the deep cache but with negligible storage size. Specifically, on the second arrival of the same object, the presence of its meta-data implies that it is likely to result in a cache hit if stored in $\mathcal{C}$ with TTL θ. This approach is akin to ideas the use Bloom filters [23] and 2Q [27].

Shallow cache for recurring bursty objects. While the shadow cache filters out rare objects (e.g. one-hit wonders), it has an undesirable impact on the cache hit rate as the first two arrivals of any object always result in cache misses. This can lead to a larger TTL θ (for the deep cache), for a given target hit rate, compared to d-TTL. This problem is even more pronounced when one considers correlated requests (e.g. Markovian labeling in our model), where requests for an object typically follow an on-off pattern—few request arrivals in a short time-period followed by a long time-period with no requests.[9] Inspired from multi-level caches such as LRU-K [28], we use a shallow cache to counter this problem. By caching new arrivals with a smaller TTL θ^s in the shallow cache, f-TTL ensures that, on the one hand, rare and unpopular objects are quickly evicted; while on the other hand, a cache miss for correlated requests on the second arrival is avoided.

Two-level adaptation. In f-TTL, the TTL θ is used to achieve the target hit rate h^* and is adapted in the same way as in d-TTL. The TTL of the shallow cache, θ^s, is however adapted to achieve the normalized size target s^*. Therefore, θ^s must depend on the normalized size $\hat{s}(l)$.

Consider the following adaptation strategy: Compute an online unbiased estimate of the normalized size, denoted by

Algorithm 2 Filtering TTL (f-TTL)

Input:
Target hit rate $\mathbf{h}^*$, target normalized size $\mathbf{s}^*$, TTL bound $\mathbf{L}$.
For l-th request, $l \in \mathbb{N}$, object $c(l)$, size $w(l)$ & type $t(l)$.

Output: Cache requested object using dynamic TTLs, $\boldsymbol{\theta}$ and $\boldsymbol{\theta}^s$.

1: **Intialize:** Latent variables, $\boldsymbol{\vartheta}(0) = \boldsymbol{\vartheta}^s(0) = \mathbf{0}$.
2: **for all** $l \in \mathbb{N}$ **do**
3: **if** Cache hit, $c(l) \in \mathcal{C} \cup \mathcal{C}_s$ **then**
4: $Y(l) = 1$,
5: $s(l) = \begin{cases} \theta_{t(l)}(l) - \psi^0_{c(l)}, & \text{if } c \in \mathcal{C} \\ \theta_{t(l)}(l) - \psi^1_{c(l)}, & \text{if } c \in \mathcal{C}_s. \end{cases}$
6: **else if** Virtual hit, $c(l) \notin \mathcal{C} \cup \mathcal{C}_s$ and $\tilde{c}(l) \in \mathcal{C}_s$ **then**
7: $Y(l) = 0$, $s(l) = \theta_{t(l)}(l)$.
8: **else** Cache miss
9: $Y(l) = 0$, $s(l) = \theta^s_{t(l)}(l)$.
10: **Update TTL** $\theta_{t(l)}(l)$:

$$\vartheta_{t(l)}(l+1) = \mathcal{P}_{[0,1]}\left(\vartheta_{t(l)}(l) + \eta(l)\widehat{w}(l)\left(h^*_{t(l)} - Y(l)\right)\right)$$
$$\theta_{t(l)}(l+1) = L_{t(l)}\vartheta_{t(l)}(l+1),$$

where,
$\eta(l) = \frac{\eta_0}{l^\alpha}$ is a decaying step size for $\alpha \in (1/2, 1)$,
$\mathcal{P}_{[0,1]}(x) = \min\{1, \max\{0, x\}\}$,
$\widehat{w}(l) = 1$ for OHR and $w(l)$ for BHR.

11: **Update TTL** $\theta^s_{t(l)}(l)$:

$$\vartheta^s_{t(l)}(l+1) = \mathcal{P}_{[0,1]}\left(\vartheta^s_{t(l)}(l) + \eta_s(l)w(l)(s^*_{t(l)} - s(l))\right)$$
$$\theta^s_{t(l)}(l+1) = L_{t(l)}\vartheta_{t(l)}(l+1)\Gamma\left(\vartheta_{t(l)}(l+1), \vartheta^s_{t(l)}(l+1); \epsilon\right) \tag{3}$$

where,
$\eta_s(l) = \frac{\eta_0}{l}$ and ϵ is a parameter of the algorithm,
$\Gamma(\cdot,\cdot;\epsilon)$ is a *threshold function*.

12: **if** Cache hit, $c(l) \in \mathcal{C} \cup \mathcal{C}_s$ **then**
13: **if** $c(l) \in \mathcal{C}_s$ **then**
14: Evict $\tilde{c}(l)$ from $\mathcal{C}_s$ and move $c(l)$ from $\mathcal{C}_s$ to $\mathcal{C}$.
15: Set TTL tuple to $(\theta_{t(l)}(l+1), 0, 0)$.
16: **else if** Virtual hit, $c(l) \notin \mathcal{C} \cup \mathcal{C}_s$ and $\tilde{c}(l) \in \mathcal{C}_s$ **then**
17: Evict $\tilde{c}(l)$ from $\mathcal{C}_s$,
18: Cache $c(l)$ in $\mathcal{C}$ and set TTL tuple to $(\theta_{t(l)}(l+1), 0, 0)$.
19: **else** Cache Miss
20: Cache $c(l)$ and $\tilde{c}(l)$ in $\mathcal{C}_s$ and
set TTL tuple to $(0, \theta^s_{t(l)}(l+1), \theta_{t(l)}(l+1))$.

[9] Under our model, a lazy labelling Markov chain with K states where the transitions are $i \to i$ w.p. 0.5 and $i \to (i+1) \mod K$ w.p. 0.5., for all $i \in [K]$, is such an example.

$s(l)$ for the l-th arrival, and then update θ^s to $\theta^s \leftarrow \min\{(\theta^s + \eta_s(s^* - s(l)))^+, \theta\}$ for some decaying step size η_s. Clearly, as the expected normalized size $s = \mathbb{E}[s(l)]$ approaches s^* and the expected hit rate $h = \mathbb{E}[Y(l)]$ approaches h^*, the expected change in TTL pair (θ, θ^s) approaches $(0, 0)$.[10]

Two time-scale approach for convergence. Due to the noisy estimates of the expected hit rate and the expected normalized size, $Y(l)$ and $s(l)$ resp., we use decaying step sizes $\eta(l)$ and $\eta_s(l)$. However, if $\eta(l)$ and $\eta_s(l)$ are of the same order, convergence is no longer guaranteed as adaptation noise for θ and θ^s are of the same order. For example, if for multiple (θ_i, θ_i^s), the same target hit rate and normalized size can be attained, then the TTL pair may oscillate between these points. We avoid this by using $\eta(l)$ and $\eta_s(l)$ of different orders: on l-th arrival, $\theta \leftarrow \min\{(\theta + (h^* - Y(l))/l^\alpha)^+, L\}$ for $\eta(l) = 1/l^\alpha$, $\alpha \in (0.5, 1)$ and $\theta^s \leftarrow \min\{(\theta^s + (s^* - s(l))/l)^+, \theta\}$ for $\eta_s(l) = 1/l$. By varying θ^s much slower than θ, the adaptation behaves as if θ^s is fixed and it changes θ to attain the hit rate h^*. On the other hand, θ^s varies slowly to attain the normalized size while h^* is maintained through faster dynamics.

Mode collapse in f-TTL with truncation. Recall that in the presence of rare objects, TTL θ is truncated by a large but finite L. Consider a scenario where f-TTL attains hit rate target h^* if and only if both $\theta > 0$ and $\theta^s > 0$. Now, let s^* be set in such a way that it is too small to attain h^*. Under this scenario the TTL value θ^s constantly decreases and collapses to 0, and the TTL value θ constantly increases and collapses to L. Mode collapse $(\theta, \theta^s) = (L, 0)$ occurs while failing to achieve the achievable hit rate h^*. In order to avoid such mode collapse, it is necessary to intervene in the natural adaptation of θ^s and increase it whenever θ is close to L. But due to this intervention, the value of θ^s may change even if the expected normalized size estimate equals the target s^*, which presents a paradox!

Two time-scale actor-critic adaptation. To solve the mode collapse problem, we rely on the principle of separating *critics* (the parameters that evaluate performance of the algorithm and serve as memory of the system), and *actors* (the parameters that are functions of the critics and govern the algorithm). This is a key idea introduced in the Actor-critic algorithms [21]. Specifically, we maintain two critic parameters ϑ and ϑ^s, whereas the parameters θ and θ^s play the role of actors.[11] The critics are updated as discussed above but constrained in $[0, 1]$, i.e on l-th arrival $\vartheta \leftarrow \min\{(\vartheta + (h^* - Y(l))/l^\alpha)^+, 1\}$, for $\alpha \in (0.5, 1)$ and $\vartheta^s \leftarrow \min\{(\vartheta^s + (s^* - s(l))/l)^+, 1\}$. The actors are updated as, $\theta = L\vartheta$, and for some small $\epsilon > 0$, (i) $\theta^s = L\vartheta^s$ if $\vartheta < 1 - 3\epsilon/2$, (ii) $\theta^s = L\vartheta$ if $\vartheta > 1 - \epsilon/2$, and (iii) smooth interpolation in between. With this dynamics ϑ^s stops changing if the expected normalized size estimate equals s^*, which in turn fixes θ^s despite the external intervention.

3) Estimating the normalized size: The update rule for θ^s depends on the normalized size $\hat{s}(l)$ which is not known upon the arrival of l-th request. Therefore, we need to estimate $\hat{s}(l)$. However, as $\hat{s}(l)$ depends on updated TTL values, and future arrivals, its online estimation is non-trivial. The term $s(l)$, defined in lines 5, 7, and 9 in Algorithm 2, serves as an online estimate of $\hat{s}(l)$.[12] First, we construct an approximate upper bound for $\hat{s}(l)$ as $\theta^s_{t(l)}(l)$ for cache miss and $\theta_{t(l)}(l)$ otherwise. Additionally, if it is a deep (resp. shallow) cache hit with remaining timer value $\psi^0_{c(l)}$ (resp. $\psi^1_{c(l)}$), we update the estimate to $(\theta_{t(l)}(l) - \psi^0_{c(l)})$ (resp. $(\theta_{t(l)}(l) - \psi^1_{c(l)})$), to correct for the past overestimation. Due to decaying step sizes, and bounded TTLs and object sizes, replacing $\hat{s}(l)$ by $s(l)$ in Eq. (4) keeps s_t unchanged $\forall t \in [T]$. We postpone the details to Appendix A.

4) Adapting $\boldsymbol{\theta}^s(l)$ and $\boldsymbol{\theta}(l)$: The adaptation of the parameters $\boldsymbol{\theta}(l)$ and $\boldsymbol{\theta}^s(l)$ is done following the above actor-critic mechanism, where $\boldsymbol{\vartheta}(l)$ and $\boldsymbol{\vartheta}^s(l)$ are the two critic parameters lying in $[0, 1]^T$. Like d-TTL, the f-TTL algorithm adaptively decreases $\boldsymbol{\vartheta}(l)$ during cache hits and increases $\boldsymbol{\vartheta}(l)$ during cache misses. Additionally, f-TTL also increases $\boldsymbol{\vartheta}(l)$ during cache virtual hits. Finally, for each type t and on each arrival l, the TTL $\theta_t(l) = L_t\vartheta_t(l)$ [line 10 in Algorithm 2].

The external intervention is implemented through a *threshold function*, $\Gamma(x, y; \epsilon) : [0, 1]^2 \to [0, 1]$. Specifically, the parameter $\boldsymbol{\theta}^s(l)$ is defined in Equation 3 as

$$\theta^s_t(l) = L_t\vartheta_t(l)\Gamma\left(\vartheta_t(l), \vartheta^s_t(l); \epsilon\right) \ \forall t \in [T].$$

Here, the threshold function $\Gamma(x, y; \epsilon)$ takes value 1 for $x \geq 1 - \epsilon/2$ and value y for $x \leq 1 - 3\epsilon/2$, and the partial derivative w.r.t. x is bounded by $4/\epsilon$. Additionally, it is twice differentiable and non-decreasing w.r.t. both x and y.

This definition maintains the invariant $\theta^s_t(l) \leq \theta_t(l)\ \forall t, l$. Note that, in the extreme case when $\boldsymbol{\vartheta}^s(l) = 0$, we only cache the metadata of the requested object on first access, but not the object itself. We call this the *full filtering TTL*.

One such threshold function can be given as follows with the convention $0/0 = 1$,

$$\Gamma(x, y; \epsilon) = \left(y + \frac{(1 - y)((x - 1 + \frac{3\epsilon}{2})^+)^4}{((x - 1 + \frac{3\epsilon}{2})^+)^4 + ((1 - \frac{\epsilon}{2} - x)^+)^4}\right). \tag{5}$$

If the estimate $s(l) > s^*_{t(l)}$, intuition suggests more aggressive filtering is required. To enhance filtering we decrease $\vartheta^s_{t(l)}(l)$ and consequently $\theta^s_{t(l)}(l)$. The opposite occurs when $s(l) < s^*_{t(l)}$ [line 11 in Algorithm 2].

IV. Analysis of Adaptive TTL-Based Algorithms

In this section we present our main theoretical results. We consider a setting where the TTL parameters live in a compact space. Indeed, if the TTL values become unbounded, then objects never leave the cache after entering it. This setting is captured through the definition of $\mathbf{L}$ feasibility, presented below.

Definition 5. For an arrival process $\mathcal{A}$ and the d-TTL algorithm, object (byte) hit rate $\mathbf{h}$ is '$\mathbf{L}$-feasible' if there exists a $\boldsymbol{\theta} \preccurlyeq \mathbf{L}$ such that the d-TTL algorithm with fixed TTL $\boldsymbol{\theta}$ achieves $\mathbf{h}$ asymptotically almost surely under $\mathcal{A}$.

[10] This is not the only mode of convergence for θ^s. Detailed discussion on the convergence of our algorithm will follow shortly.

[11] It is possible to work with θ alone, without introducing ϑ. However, having ϑ is convenient for defining the threshold function in (5).

[12] With slight abuse of notation, we use 's' in $s(l)$ and $\hat{s}(l)$ to denote 'normalized size'; whereas in $\mathcal{C}_s$, $\theta^s(l)$, $\vartheta^s(l)$, and $\eta_s(l)$'s' denotes 'secondary cache'.

Definition 6. For an arrival process $\mathcal{A}$ and the f-TTL caching algorithm, the object (byte) hit rate and normalized size tuple $(\boldsymbol{h}, \boldsymbol{s})$ is '$\mathbf{L}$-feasible' if there exist $\boldsymbol{\theta} \preccurlyeq \mathbf{L}$ and $\boldsymbol{\theta}^s \preccurlyeq \boldsymbol{\theta}$, such that f-TTL algorithm with fixed TTL pair $(\boldsymbol{\theta}, \boldsymbol{\theta}^s)$ achieves $(\boldsymbol{h}, \boldsymbol{s})$ asymptotically almost surely under $\mathcal{A}$.

A hit rate $\mathbf{h}^*$ or a tuple $(\mathbf{h}^*, \mathbf{s}^*)$ are '$\mathbf{L}'$-feasible' if they are '$\mathbf{L}$-feasible' for all $\mathbf{L}' \succ \mathbf{L}$, where $\mathbf{L}'$ feasibility is more strict than $\mathbf{L}$ feasibility. To avoid trivial cases (hit rate being 0 or 1), we consider *typical* hit rates as defined below.

Definition 7. A hit rate $\mathbf{h}$ is 'typical' if $h_t \in (0,1),\ \forall t \in [T]$.

A. Main Results

We now show that both d-TTL and f-TTL asymptotically almost surely (a.a.s.) achieve any 'feasible' *object (byte) hit rate*, $\mathbf{h}^*$ for the arrival process in Assumption 1.1, using stochastic approximation techniques. Further, we prove a.a.s that f-TTL converges to a target $(\boldsymbol{h}^*, \boldsymbol{s}^*)$ tuple for *object (byte) hit rate* and *normalized size*.

Theorem 1. *Under Assumption 1.1 with $\|\boldsymbol{L}\|_\infty$-rarity condition (i.e. $R = \|\boldsymbol{L}\|_\infty$):*

***d-TTL:** if the hit rate target $\boldsymbol{h}^*$ is both $\boldsymbol{L}$-feasible and 'typical', then the d-TTL algorithm with parameter $\boldsymbol{L}$ converges to a TTL $\boldsymbol{\theta}^*$ a.a.s. Further, the average hit rate converges to $\boldsymbol{h}^*$ a.a.s.*

***f-TTL:** if the target tuple of hit rate and normalized size, $(\boldsymbol{h}^*, \boldsymbol{s}^*)$, is $(1-2\epsilon)\boldsymbol{L}$-feasible, with $\epsilon > 0$, and $\boldsymbol{h}^*$ is 'typical', then the f-TTL algorithm with parameter $\boldsymbol{L}$ and ϵ converges to a TTL pair $(\boldsymbol{\theta}^*, \boldsymbol{\theta}^{s^*})$ a.a.s. Further the average hit rate converges to $\boldsymbol{h^*}$ a.a.s., while the average normalized size converges to some $\hat{\boldsymbol{s}}$ a.a.s.. Additionally, $\hat{\boldsymbol{s}}$ for each type t, satisfies one of the following three conditions:*

1) *The average normalized size converges to $\hat{s}_t = s_t^*$ a.a.s.*
2) *The average normalized size converges to $\hat{s}_t > s_t^*$ a.a.s. and $\theta_t^{s^*} = 0$ a.a.s.*
3) *The average normalized size converges to $\hat{s}_t < s_t^*$ a.a.s. and $\theta_t^{s^*} = \theta_t^*$ a.a.s.*

As stated in Theorem 1, the f-TTL algorithm converges to one of three scenarios. We refer to the second scenario as *collapse to full-filtering TTL*, because in this case, the lower-level cache contains only labels of objects instead of caching the objects themselves. We refer to the third scenario as *collapse to d-TTL*, because in this case, cached objects have equal TTL values in the deep, shadow and shallow caches.

The f-TTL algorithm ensures that under Assumption 1.1, with $\|\boldsymbol{L}\|_\infty$-rarity condition, the rate at which the rare objects enter the deep cache $\mathcal{C}$ is a.a.s. zero (details deferred to Appendix A), thus limiting the normalized size contribution of the rare objects to those residing in the shallow cache of $\mathcal{C}_s$. Theorem 1 states that f-TTL converges to a filtration level which is within two extremes: *full-filtering* f-TTL where rare objects are completely filtered (scenario 2) and d-TTL where no filtration occurs (scenario 3).

We note that in f-TTL, scenario 1 and scenario 3 have 'good' properties. Specifically, in each of these two scenario, the f-TTL algorithm converges to an average normalized size which is smaller than or equal to the target normalized size. However, in scenario 2, the average normalized size converges to a normalized size larger than the given target under general arrivals in Assumption 1.1. Further, under Assumption 1.2, we show in Corollary 1 that scenario 2 cannot occur.

Corollary 1. *Assume the target tuple of hit rate and normalized size, $(\boldsymbol{h}^*, \boldsymbol{s}^*)$, is $(1-2\epsilon)\boldsymbol{L}$-feasible with $\epsilon > 0$ and additionally, $\boldsymbol{h}^*$ is 'typical'. Under Assumption 1.2 with $\|\boldsymbol{L}\|_\infty$-rarity condition, an f-TTL algorithm with parameters $\boldsymbol{L}$, ϵ, achieves asymptotically almost surely a tuple $(\boldsymbol{h}^*, \boldsymbol{s})$ with normalized size $\boldsymbol{s} \preccurlyeq \boldsymbol{s}^*$.*

B. Proof Sketch of Main Results

Here we present a proof sketch of Theorem 1, and Corollary 1. The complete proof can be found in Appendix.[13]

The proof of Theorem 1 consists of two parts. The first part deals with the 'static analysis' of the caching process, where parameters $\boldsymbol{\vartheta}$ and $\boldsymbol{\vartheta}^s$ both take fixed values in $[0,1]$ (i.e., no adaptation of parameters). In the second part ('dynamic analysis'), employing techniques from the theory of stochastic approximation [26], we show that the TTL $\boldsymbol{\theta}$ for d-TTL and the TTL pair $(\boldsymbol{\theta}, \boldsymbol{\theta}^s)$ for f-TTL converge almost surely. Further, the average hit rate (and average normalized size for f-TTL) satisfies Theorem 1.

The evolution of the caching process is represented as a discrete time stochastic process uniformized over the arrivals into the system. At each arrival, the system state is as follows: *(1)* the timers of recurrent objects (i.e. $(\psi_c^0, \psi_c^1, \psi_c^2)$ for $c \in \mathcal{K}$), *(2)* the current value of the pair $(\boldsymbol{\vartheta}, \boldsymbol{\vartheta}^s)$, and *(3)* the object requested on the last arrival. However, due to the presence of a constant fraction of non-stationary arrivals in Assumption 1.1, we maintain a state with incomplete information. Specifically, our (incomplete) state representation does not contain the timer values of the rare objects present in the system. This introduces a bias (which is treated as noise) between the actual process, and the evolution of the system under incomplete state information.

In the static analysis, we prove that the system with fixed $\boldsymbol{\vartheta}$ and $\boldsymbol{\vartheta}^s$ exhibits uniform convergence to a unique stationary distribution. Further, using techniques from regeneration process and the 'rarity condition' in Equation (1), we calculate the asymptotic average hit rates and the asymptotic average normalized sizes of each type for the caching process. We then argue that asymptotic averages of both the hit rate and normalized size of the incomplete state system is the same as the original system. This is important for the dynamic analysis because this characterizes the time averages of the adaptation of $\boldsymbol{\vartheta}$ and $\boldsymbol{\vartheta}^s$.

In the dynamic analysis, we analyze the system under variable $\boldsymbol{\vartheta}$ and $\boldsymbol{\vartheta}^s$, using results of almost sure convergence of (actor-critic) stochastic approximations with a two timescale separation [29]. The proof follows the ODE method; the following are the key steps in the proof:

1) We show that the effects of the bias introduced by the non-stationary process satisfies Kushner-Clark condition [26].

[13]Due to lack of space we present the appendices as supplementary material to the main article.

2) The expectation (w.r.t. the history up to step l) of the l-th update as a function of $(\boldsymbol{\vartheta}, \boldsymbol{\vartheta}^s)$ is Lipschitz continuous.
3) The incomplete information system is uniformly ergodic.
4) The ODE (for a fixed $\boldsymbol{\vartheta}^s$) representing the mean evolution of $\boldsymbol{\vartheta}$ has a unique limit point. Therefore, the limit point of this ODE is a unique function of $\boldsymbol{\vartheta}^s$.
5) (f-TTL analysis with two timescales) Let the ODE at the slower time scale, representing the mean evolution of $\boldsymbol{\vartheta}^s$, have stationary points $\{(\boldsymbol{\vartheta}, \boldsymbol{\vartheta}^s)_i\}$. We characterize each stationary point, and show that it corresponds to one of the three cases stated in Theorem 1. Finally, we prove that all the limit points of the evolution are given by the stationary points of the ODE.

As stated in Theorem 1, the f-TTL algorithm converges to one of three scenarios, under general arrivals in Assumption 1.1. However, under Assumption 1.2, we show that the scenario 2 cannot occur, as formalized in Corollary 1. The proof of Corollary 1 follows from Theorem 1 and the following Lemma 1.

Lemma 1. *Under Assumption 1.2 with $\|\mathbf{L}\|_\infty$-rarity condition and for any type t, suppose f-TTL algorithm achieves an average hit rate h_t with two different TTL pairs, 1) (θ_t, θ_t^s) with $\theta_t^s = 0$ (full filtering), and 2) $(\hat{\theta}_t, \hat{\theta}_t^s)$, with $\hat{\theta}_t^s > 0$, where $\max\{\theta_t, \hat{\theta}_t\} \leq L_t$. Then the normalized size achieved with the first pair is less or equal to the normalized size achieved with the second pair. Moreover, in the presence of rare objects of type t, i.e. $\alpha_t > 0$, this inequality in the achieved normalized size is strict.*

The proof of this lemma is presented in Appendix A and the technique, in its current form, is specific to Assumption 1.2. The condition that object size is non-decreasing w.r.t. π_c (third bullet) in Assumption 1.2, is necessary for the guarantee to hold for object hit rate target under our proof. This condition can be removed if all objects of each type have the same size or a byte hit rate is targeted.

V. Implementation of d-TTL and f-TTL

One of the main practical challenges in implementing d-TTL and f-TTL is adapting θ and θ^s to achieve the desired hit rate in the presence of unpredictable non-stationary traffic. We observe the following major differences between the theoretical and practical settings. First, the arrival process in practice changes over time (e.g. day-night variations) whereas our model assumes the stationary part is fixed. Second, the hit rate performance in finite time horizons is often of practical interest. While our content request model accounts for non-stationary behavior in finite time windows, the algorithms are shown to converge to the target hit rate *asymptotically*. But, this may not be true in finite time windows. We now discuss some modifications we make to translate theory to practice and evaluate these modification in Section VI.

Fixing the maximum TTL. The truncation parameter (maximum TTL value) $\boldsymbol{L}$ defined in Section III is crucial in the analysis of the system. However, in practice, we can choose an arbitrarily large value such that we let θ explore a larger space to achieve the desired hit rate in both d-TTL and f-TTL.

Constant step sizes for θ and θ^s updates. Algorithms 1 and 2 use decaying step sizes $\eta(l)$ and $\eta_s(l)$ while adapting θ and θ^s. This is not ideal in practical settings where the traffic composition is constantly changing, and we need θ and θ^s to capture those variations. Therefore, we choose carefully hand-tuned constant step sizes that capture the variability in traffic well. We discuss the sensitivity of the d-TTL and f-TTL algorithms to changes in the step size in Section VI-F.

Target cache size to target normalized size. In the f-TTL algorithm a normalized size is targeted. However, a CDN operator may specify target cache sizes, cs_t^* *bytes* for type t, based on the performance requirements of different content types. In this scenario, the target cache size is translated to a normalized size target. Using the (estimated) arrival rate, λ_t *bytes/sec*, we compute the normalized size target $s_t^* = cs_t^*/\lambda_t$ *sec*, for each type t. In our experiments in Section VI, for a target cache size that is 50% of the cache size of d-TTL, f-TTL achieves the same hit rate as d-TTL but at almost half the cache space.

Tuning normalized size targets. In practice, f-TTL may not be able to achieve small normalized size targets in the presence of time varying and non-negligible non stationary traffic. In such cases, CDN operators can use the target normalized size as a tunable knob to adaptively filter out unpredictable non-stationary content. For instance, with a small target normalized size during a sudden surge of non-stationary content, θ^s is aggressively reduced. This in turn filters out a lot of non-stationary content while an appropriate increase in θ maintains the target hit rate. However, an extremely aggressive normalized size target may not be the right choice, as it could lead to a large average cache size due to an excessive increase in θ. Specifically, in our simulations, at a target OHR of 40%, setting a non-zero target normalized size leads to nearly 15% decrease in the average cache size as compared to a target normalized size of 0.

VI. Empirical Evaluation

We evaluate the performance of d-TTL and f-TTL, both in terms of the hit rate achieved and the cache size requirements, using actual production traces from a major CDN.

A. Experimental setup

Content request traces. We use an extensive data set containing access logs for content requested by users that we collected from a typical production server in Akamai's commercially-deployed CDN [3]. The logs contain requests for predominantly web content (hence, we only compute TTLs for a single content type in our evaluation). Each log line corresponds to a single request and contains a timestamp, the requested URL (anonymized), object size, and bytes served for the request. The access logs were collected over a period of 9 days. The traffic served in Gbps captured in our data set is shown in Figure 1. We see that there is a diurnal traffic pattern with the first peak generally around 12PM and the second peak occurring around 10-11PM. There is a small dip in traffic during the day between 4-6PM. This could be during evening commute when there is less internet traffic. The lowest

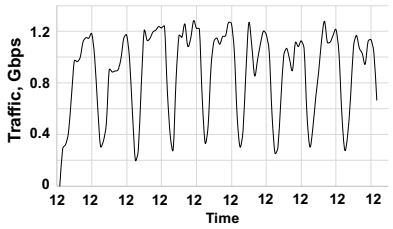

Fig. 1: Content traffic served to users from the CDN server, averaged every 2 hours. The traces were collected from 29^{th} January to 6^{th} February 2015.

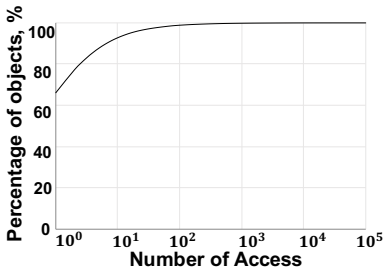

Fig. 2: Popularity of content accessed by users in the 9-day period.

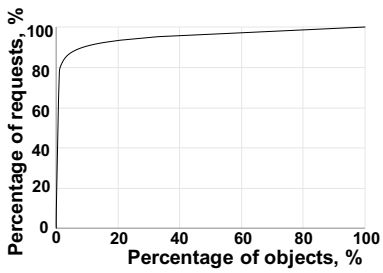

Fig. 3: A large fraction of the requests are for a small fraction of the objects.

traffic is observed at the early hours of the morning between 4AM and 9AM.

The content requests traces used in this work contain 504 million requests (resp., 165TB) for 25 million distinct objects (resp., 15TB). From Figure 2, we see that about 70% of the objects in the trace are one-hit wonders. This indicates that a large fraction of objects need to be cached with no contribution towards the cache hit rate. Moreover, from Figure 3, we see that about 90% of the requests are for only 10% of the most popular objects indicating that the remaining 90% of the objects contribute very little to the cache hit rate. Hence, both these figures indicate that the request trace has a large fraction of unpopular content. The presence of a significant amount of non-stationary traffic in the form of "one-hit-wonders" in production traffic is consistent with similar observations made in earlier work [20].

Trace-based cache simulator. We built a custom event-driven simulator to simulate the different TTL caching algorithms. The simulator takes as input the content traces and computes a number of statistics such as the hit rate obtained over time, the variation in θ, θ^s and the cache size over time. We implemented and simulated both d-TTL and f-TTL using the parameters listed in Table I.

We use constant step sizes, η=1e-2 and η_s=1e-9, while adapting the values of θ and θ^s. The values chosen were found to capture the variability in our input trace well. We evaluate the sensitivity of d-TTL and f-TTL to changes in η and η_s in Section VI-F.

TABLE I: Simulation parameters. In this table s^* is the target normalized size and w_{avg} is the average object size.

Simulation length	9 days	Number of requests	504 m
Min TTL value	0 sec	Max TTL value	10^7 sec
Step size for θ	η	Step size for θ^s	$\frac{\eta_s}{s^* w_{\text{avg}}}$

B. How TTLs adapt over time

To understand how d-TLL and f-TTL adapt their TTLs over time in response to request dynamics, we simulated these algorithms with a target object hit rate of 60% and a target normalized size that is 50% of the normalized size achieved by d-TTL. In Figure 4 we plot the traffic in Gbps, the variation in θ for d-TTL, θ for f-TTL and θ^s over time, all averaged over 2 hour windows. We consider only the object hit rate scenario to explain the dynamics. We observe similar performance when we consider byte hit rates.

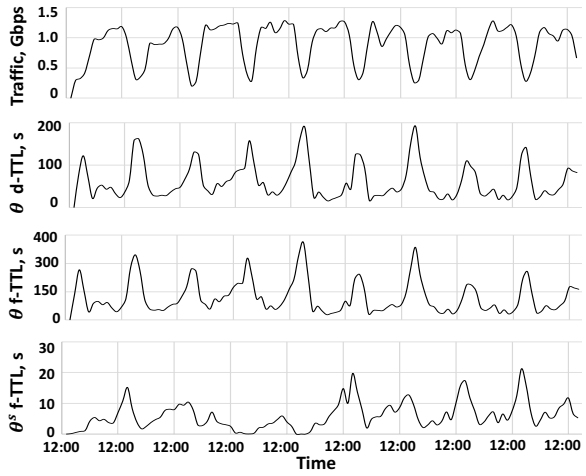

Fig. 4: Variation in θ for d-TTL, θ for f-TTL and θ^s over time with target object hit rate=60%.

From Figure 4, we see that the value of θ for d-TTL is smaller than that of f-TTL. This happens due to the fact that f-TTL filters out rare objects to meet the target normalized size, which can in turn reduce the hit rate, resulting in an increase in θ to achieve the target hit rate. We also observe that θ for both d-TTL and f-TTL is generally smaller during peak hours when compared to off-peak hours. This is because the inter-arrival time of popular content is smaller during peak hours. Hence, a smaller θ is sufficient to achieve the desired hit rate. However, during off-peak hours, traffic drops by almost 70%. With fewer content arrivals per second, θ increases to provide the same hit rate. In the case of f-TTL, the increase in θ, increases the normalized size of the system, which in turn leads to a decrease in θ^s. This matches with the theoretical intuition that d-TTL adapts θ only to achieve the target hit rate while f-TTL adapts both θ and θ^s to reduce the cache size while also achieving the target hit rate.

C. Hit rate performance of d-TTL and f-TTL

The performance of a caching algorithm is often measured by its hit rate curve (HRC) that relates its cache size with the (object or byte) hit rate that it achieves. HRCs are useful

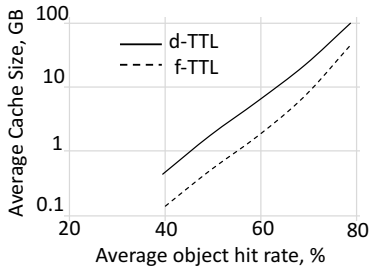

Fig. 5: Hit rate curve for object hit rates.

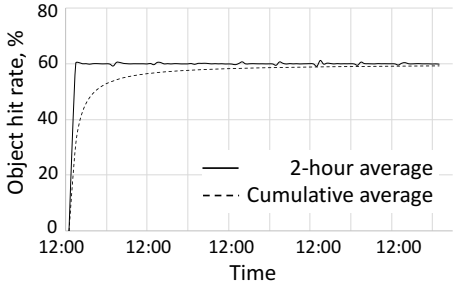

Fig. 6: Object hit rate convergence over time for d-TTL; target object hit rate=60%.

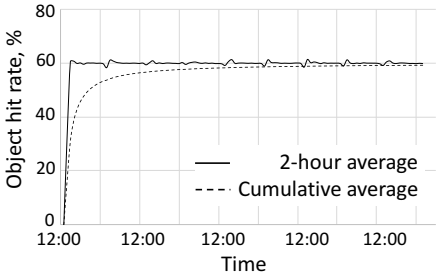

Fig. 7: Object hit rate convergence over time for f-TTL; target object hit rate=60%.

for CDNs as they help provision the right amount of cache space to obtain a certain hit rate. We compare the HRCs of d-TTL and f-TTL for object hit rates and show that f-TTL significantly outperforms d-TTL by filtering out the rarely-accessed non-stationary objects. The HRCs for byte hit rates are shown in Appendix B.

To obtain the HRC for d-TTL, we fix the target hit rate at 80%, 70%, 60%, 50% and 40% and measure the hit rate and cache size achieved by the algorithm. Similarly, for f-TTL, we fix the target hit rates at 80%, 70%, 60%, 50% and 40%.

Further, we fix *target cache sizes* of f-TTL as given in 9-th column in Table II.[14] Next, using the average traffic arrival rate (over the entire 9-day trace) we set the corresponding *normalized size targets* (= size target/ arrival rate) for f-TTL.

The HRCs for object hit rates are shown in Figures 5. Note that the y-axis is presented in log scale for clarity. The hit rate performance for byte hit rates is discussed in Appendix B.

From Figure 5 we see that f-TTL always performs better than d-TTL i.e. for a given hit rate, f-TTL requires lesser cache size on average than d-TTL. In particular, on average, f-TTL achieves the target cache size with less than 6% error. In Appendix C, we discuss the performance of f-TTL for other target cache sizes.

D. Convergence of d-TTL and f-TTL

For the dynamic TTL algorithms to be useful in practice, they need to converge to the target hit rate with low error. In this section we measure the object hit rate convergence over time, averaged over the entire time window and averaged over 2 hour windows for both d-TTL and f-TTL. We set the target object hit rate to 60% and a target cache size as given in Table II. The byte hit rate convergence is discussed in Appendix B.

From Figures 6 and 7, we see that the 2 hour averaged object hit rates achieved by both d-TTL and f-TTL have a cumulative error of less than 1.3% while achieving the target object hit rate on average. We see that both d-TTL and f-TTL tend to converge to the target hit rate, which illustrates that both d-TTL and f-TTL are able to adapt well to the dynamics of the input traffic.

[14]The size targets are chosen to achieve 50% size of d-TTL. In practice it can be set according to the choice of the designer.

In general, we also see that d-TTL has a lower variability for object hit rates compared to f-TTL due to the fact that d-TTL does not have any bound on the normalized size while achieving the target hit rate, while f-TTL is constantly filtering out non-stationary content to meet the target normalized size while also achieving the target hit rate.

E. Accuracy of d-TTL and f-TTL

A key goal of d-TTL and f-TTL is to achieve a target hit rate, even in the presence of bursty and non-stationary requests. We evaluate the performance of both these algorithms by fixing the target hit rate and comparing the hit rates achieved by d-TTL and f-TTL with caching algorithms such as Fixed TTL (TTL-based caching algorithm that uses a constant TTL value) and LRU (constant cache size), provisioned using Che's approximation [15]. We only present the results for object hit rates (OHR) in Table II. Similar behavior is observed for byte hit rates.

For this evaluation, we fix the target hit rates (column 1) and analytically compute the TTL (characteristic time) and cache size using Che's approximation (columns 2 and 6) on the request traces assuming Poisson traffic. We then measure the hit rate and cache size of Fixed TTL (columns 3 and 4) using the TTL computed in column 2, and the hit rate of LRU (column 5) using the cache size computed in column 6. Finally, we compute the hit rate and cache size achieved by d-TTL and f-TTL (columns 7,8,10 and 11) to achieve the target hit rates in column 1 and a target cache size that is 50% of that of d-TTL (column 9).

We make the following conclusions from Table II.

1) The d-TTL and f-TTL algorithms meet the target hit rates with a small error of 1.2% on average. This is in contrast to the Fixed TTL algorithm which has a high error of 14.4% on average and LRU which has an even higher error of 20.2% on average. This shows that existing algorithms such as Fixed TTL and LRU are unable to meet the target hit rates while using heuristics such as Che's approximation, which cannot account for non-stationary content.

2) The cache size required by d-TTL and f-TTL is 23.5% and 12% respectively, of the cache size estimated by Che's approximation and 35.8% and 18.3% respectively, of the cache size achieved by the Fixed TTL algorithm, on average. This indicates that both LRU and the Fixed TTL algorithm,

TABLE II: Comparison of target hit rate and average cache size achieved by d-TTL and f-TTL with Fixed-TTL and LRU.

Target	Fixed TTL (Che's approx.)			LRU (Che's approx.)		d-TTL		Target	f-TTL	
OHR (%)	TTL (s)	OHR (%)	Size (GB)	OHR (%)	Size (GB)	OHR (%)	Size (GB)	Size (GB)	OHR (%)	Size (GB)
80	2784	83.29	217.11	84.65	316.81	78.72	97.67	48.84	78.55	55.08
70	554	75.81	51.88	78.37	77.78	69.21	21.89	10.95	69.14	11.07
60	161	68.23	16.79	71.64	25.79	59.36	6.00	3.00	59.36	2.96
50	51	60.23	5.82	64.18	9.2	49.46	1.76	0.88	49.47	0.86
40	12	50.28	1.68	54.29	2.68	39.56	0.44	0.22	39.66	0.20

TABLE III: Impact of exponential changes in constant step size η on the performance of d-TTL (robustness analysis).

Target	Average OHR (%)			Average cache size (GB)			5% outage fraction		
OHR (%)	η = 0.1	η = 0.01	η = 0.001	η = 0.1	η = 0.01	η = 0.001	η = 0.1	η = 0.01	η = 0.001
60	59.35	59.36	59.17	9.03	6.00	5.41	0.01	0.01	0.05
80	79.13	78.72	77.69	150.56	97.67	75.27	0.07	0.11	0.23

TABLE IV: Impact of linear changes in constant step size $\eta = 0.01$ on the performance of d-TTL (sensitivity analysis).

Target	Average OHR (%)			Average cache size (GB)			5% outage fraction		
OHR (%)	η(1+0.05)	η	η(1-0.05)	η(1+0.05)	η	η(1-0.05)	η(1+0.05)	η	η(1-0.05)
60	59.36	59.36	59.36	5.98	6.00	6.02	0.01	0.01	0.01
80	78.73	78.72	78.71	98.21	97.67	97.1	0.11	0.11	0.11

TABLE V: Impact of exponential changes in constant step size η_s on the performance of f-TTL (robustness analysis).

Target	Average OHR (%)			Average cache size (GB)			5% outage fraction		
OHR (%)	η_s = 1e-8	η_s = 1e-9	η_s = 1e-10	η_s = 1e-8	η_s = 1e-9	η_s = 1e-10	η_s = 1e-8	η_s = 1e-9	η_s = 1e-10
60	59.36	59.36	59.36	5.46	2.96	1.88	0.01	0.01	0.02
80	78.65	78.55	78.47	89.52	55.08	43.34	0.12	0.14	0.17

TABLE VI: Impact of linear changes in constant step size η_s = 1e-9 on the performance of f-TTL (sensitivity analysis).

Target	Average OHR (%)			Average cache size (GB)			5% outage fraction		
OHR (%)	η_s(1+0.05)	η_s	η_s(1-0.05)	η_s(1+0.05)	η_s	η_s(1-0.05)	η_s(1+0.05)	η_s	η_s(1-0.05)
60	59.36	59.36	59.36	3.01	2.96	2.91	0.01	0.01	0.01
80	78.55	78.55	78.54	55.65	55.08	54.27	0.14	0.14	0.14

provisioned using Che's approximation, grossly overestimate the cache size requirements.

We note that sophisticated models, such as the shot noise model [6] or the advanced popularity estimation [30], can potentially improve the accuracy over simple Che's approximation. We leave such study for future work.

F. Robustness and sensitivity of d-TTL and f-TTL

We use constant step sizes while adapting the values of θ and θ_s in practical settings for reasons discussed in Section V. In this section, we evaluate the robustness and sensitivity of d-TTL and f-TTL to the chosen step sizes. The robustness captures the change in performance due to large changes in the step size, whereas the sensitivity captures the change due to small perturbations around a specific step size. For ease of explanation, we only focus on two target object hit rates, 60% and 80% corresponding to medium and high hit rates. The observations are similar for other target hit rates and for byte hit rates.

Table III illustrates the robustness of d-TTL to exponential changes in the step size η. For each target hit rate, we measure the average hit rate achieved by d-TTL, the average cache size and the 5% outage fraction, for each value of step size. The 5% outage fraction is defined as the fraction of time the hit rate achieved by d-TTL differs from the target hit rate by more than 5%.

From this table, we see that a step size of 0.01 offers the best trade-off among the three parameters, namely average hit rate, average cache size and 5% outage fraction. Table IV illustrates the sensitivity of d-TTL to small changes in the step size. We evaluate d-TTL at step sizes $\eta = 0.01 \times (1 \pm 0.05)$. We see that d-TTL is insensitive to small changes in step size.

To evaluate the robustness and sensitivity of f-TTL, we fix the step size $\eta = 0.01$ to update θ and evaluate the performance of f-TTL at different step sizes, η_s, to update θ_s. The results for robustness and sensitivity are shown in Tables V and VI respectively. For f-TTL, we see that a step size of η_s=1e-9 offers the best tradeoff among the different parameters namely average hit rate, average cache size and 5% outage fraction. Like, d-TTL, f-TTL is insensitive to the changes in step size parameter η_s.

In Table III and Table V, a large step size makes the d-TTL and f-TTL algorithms more adaptive to the changes in traffic statistics. This results in reduced error in the average OHR and reduced 5% outage fraction. However, during periods of high burstiness, a large step size can lead to a rapid increase in the cache size required to maintain the target hit rate. The opposite happens for small step sizes.

VII. Related Work

Caching algorithms have been studied for decades in different contexts such as CPU caches, memory caches, CDN caches and so on. We briefly review some relevant prior work.

TTL-based caching. TTL caches have found their place in theory as a tool for analyzing capacity based caches [9], [12], [13], starting from characteristic time approximation of LRU caches [14], [15]. Recently, its generalizations [5], [18] have commented on its wider applicability. However, the generalizations hint towards the need for more tailored approximations and show that the vanilla characteristic time approximation can be inaccurate [5]. On the applications side, recent works have demonstrated the use of TTL caches in utility maximization [13] and hit ratio maximization [11]. Specifically, in [13] the authors provide an online TTL adaptation highlighting the need for adaptive algorithms. However, unlike prior work, we propose the first adaptive TTL-based caching algorithms that provides provable hit rate and normalized size performance in the presence of non-stationary traffic such as one-hit wonders and traffic bursts.

We also review some capacity-based caching algorithms.

Capacity-based caching. Capacity-based caching algorithms have been in existence for over 4 decades and have been studied both theoretically (e.g. exact expressions [31]–[33] and mean field approximations [34]–[36] for hit rates, mixing time of caching algorithms [37]) and empirically (in the context of web caching [38]). Various cache replacement algorithms have been proposed based on the frequency of object requests (e.g. LFU), recency of object requests (e.g. LRU) or a combination of the two parameters (e.g., LRU-K, 2Q, LRFU [27], [28], [39]). Given that CDNs see a lot of non-stationary traffic, cache admission policies such as those using bloom filters [23] have also been proposed to maximize the hit rate under space constraints. Further, non-uniform distribution of object sizes have led to more work that admit objects based on the size (e.g., LRU-S, AdaptSize [40], [41]). While several capacity-based algorithms have been proposed, most don't provide theoretical guarantees in achieving target hit rates.

Cache tuning and adaptation. Most existing adaptive caching algorithms require careful parameter tuning to work in practice. There have been two main cache tuning methods: (1) global search over parameters based on prediction model, e.g. [42], [43], and (2) simulation and parameter optimization based on shadow cache, e.g. [44]. The first method often fails in the presence of cache admission policies; whereas, the second method typically assumes stationary arrival processes to work well. However, with real traffic, static parameters are not desirable [22] and an adaptive/self-tuning cache is necessary. The self-tuning heuristics include, e.g., ARC [22], CAR [45], PB-LRU [46], which try to adapt cache partitions based on system dynamics. While these tuning methods are meant to deal with non-stationary traffic, they lack theoretical guarantees unlike our work, where we provably achieve a target hit rate and a feasible normalized size by dynamically changing the TTLs of cached content.

Finally, we discuss work related to cache hierarchies, highlighting differences between those and the f-TTL algorithm.

Cache hierarchies. Cache hierarchies, made popular for web caches in [15], [47], [48], consist of separate caches, mostly LRU [48], [49] or TTL-based [9], arranged in multiple levels; with users at the lowest level and the server at the highest. A requested object is fetched from the lowest possible cache and, typically, replicated in all the caches on the request path. Analysis for network of TTL-caches were presented in [9], [10]. In a related direction, the performance of complex networks of size based caches were approximated in [49].

While similar in spirit, the f-TTL algorithm differs in its structure and operation from hierarchical caches. Besides the adaptive nature of the TTLs, the higher and lower-level caches are assumed to be co-located and no object is replicated between them—a major structural and operational difference. Further, the use of shadow cache and shallow cache in lower-level cache $\mathcal{C}_s$ distinguishes f-TTL from the above.

VIII. Conclusions

In this paper we designed adaptive TTL-based caching algorithms that can automatically learn and adapt to the request traffic and provably achieve any feasible hit rate and cache size. Our work fulfills a long-standing deficiency in the modeling and analysis of caching algorithms in the presence of bursty and non-stationary request traffic. In particular, we presented a theoretical justification for the use of two-level caches in CDN settings where large amounts of non-stationary traffic can be filtered out to conserve cache space while also achieving target hit rates. On the practical side, we evaluated our TTL caching algorithms using traffic traces from a production Akamai CDN server. The evaluation results show that our adaptive TTL algorithms can achieve the target hit rate with high accuracy; further, the two-level TTL algorithm can achieve the same target hit rate at a much smaller cache size.

Acknowledgment

This work is partially supported by the US Dept. of Transportation supported D-STOP Tier 1 University Transportation Center, NSF grants CNS-1652115, CNS-1717179, and CNS-1413998. We are grateful to the anonymous reviewers for their valuable comments that helped improve this paper.

References

[1] 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*. ACM, 2017, pp. 45–46.

[2] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *Internet Computing, IEEE*, vol. 6, no. 5, pp. 50–58, 2002, http://www.computer.org/internet/ic2002/w5050abs.htm.

[3] E. Nygren, R. Sitaraman, and J. Sun, "The Akamai Network: A platform for high-performance Internet applications," *ACM SIGOPS Operating Systems Review*, vol. 44, no. 3, pp. 2–19, 2010.

[4] Cisco, "Visual Networking Index: The Zettabyte Era—Trends and Analysis," june 2016, goo.gl/id2RB4.

[5] F. Olmos, B. Kauffmann, A. Simonian, and Y. Carlinet, "Catalog dynamics: Impact of content publishing and perishing on the performance of a lru cache," in *Teletraffic Congress (ITC), 2014 26th International*. IEEE, 2014, pp. 1–9.

[6] E. Leonardi and G. L. Torrisi, "Least recently used caches under the shot noise model," in *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 2015, pp. 2281–2289.

[7] N. Gast and B. Van Houdt, "Asymptotically exact ttl-approximations of the cache replacement algorithms lru (m) and h-lru," in *Teletraffic Congress (ITC 28), 2016 28th International*, vol. 1. IEEE, 2016, pp. 157–165.

[8] J. Jung, A. W. Berger, and H. Balakrishnan, "Modeling ttl-based internet caches," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, vol. 1. IEEE, 2003, pp. 417–426.

[9] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley, "Analysis of ttl-based cache networks," in *Performance Evaluation Methodologies and Tools (VALUETOOLS), 2012 6th International Conference on*. IEEE, 2012, pp. 1–10.

[10] D. S. Berger, P. Gland, S. Singla, and F. Ciucu, "Exact analysis of ttl cache networks," *Performance Evaluation*, vol. 79, pp. 2–23, 2014.

[11] D. S. Berger, S. Henningsen, F. Ciucu, and J. B. Schmitt, "Maximizing cache hit ratios by variance reduction," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 2, pp. 57–59, 2015.

[12] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM Transactions on Modeling and Performance Evaluation of Computing Systems*, vol. 1, no. 3, p. 12, 2016.

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

[14] R. Fagin, "Asymptotic miss ratios over independent references," *Journal of Computer and System Sciences*, vol. 14, no. 2, pp. 222–250, 1977.

[15] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE Journal on Selected Areas in Communications*, vol. 20, no. 7, pp. 1305–1314, 2002.

[16] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, "Impact of traffic mix on caching performance in a content-centric network," in *Computer Communications Workshops (INFOCOM WKSHPS), 2012 IEEE Conference on*. IEEE, 2012, pp. 310–315.

[17] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," in *Proceedings of the 24th International Teletraffic Congress*. International Teletraffic Congress, 2012, p. 8.

[18] G. Bianchi, A. Detti, A. Caponi, and N. Blefari Melazzi, "Check before storing: what is the performance price of content integrity verification in lru caching?" *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 3, pp. 59–67, 2013.

[19] F. Guillemin, B. Kauffmann, S. Moteau, and A. Simonian, "Experimental analysis of caching efficiency for youtube traffic in an isp network," in *Teletraffic Congress (ITC), 2013 25th International*. IEEE, 2013, pp. 1–9.

[20] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *SIGCOMM Comput. Commun. Rev.*, vol. 45, no. 3, pp. 52–66, Jul. 2015. [Online]. Available: http://doi.acm.org/10.1145/2805789.2805800

[21] V. R. Konda and J. N. Tsitsiklis, "Onactor-critic algorithms," *SIAM journal on Control and Optimization*, vol. 42, no. 4, pp. 1143–1166, 2003.

[22] N. Megiddo and D. S. Modha, "Outperforming lru with an adaptive replacement cache algorithm," *Computer*, vol. 37, no. 4, pp. 58–65, 2004.

[23] B. M. Maggs and R. K. Sitaraman, "Algorithmic nuggets in content delivery," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 3, pp. 52–66, 2015.

[24] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: Characterization and implications for cdns and web sites," in *Proceedings of the 11th international conference on World Wide Web*. ACM, 2002, pp. 293–304.

[25] B. Fox, "Semi-markov processes: A primer," DTIC Document, Tech. Rep., 1968.

[26] H. Kushner and G. G. Yin, *Stochastic approximation and recursive algorithms and applications*. Springer Science & Business Media, 2003, vol. 35.

[27] T. Johnson and D. Shasha, "X3: A low overhead high performance buffer management replacement algorithm," 1994.

[28] E. J. O'neil, P. E. O'neil, and G. Weikum, "The lru-k page replacement algorithm for database disk buffering," *ACM SIGMOD Record*, vol. 22, no. 2, pp. 297–306, 1993.

[29] S. Bhatnagar, R. S. Sutton, M. Ghavamzadeh, and M. Lee, "Natural actor–critic algorithms," *Automatica*, vol. 45, no. 11, pp. 2471–2482, 2009.

[30] F. Olmos and B. Kauffmann, "An inverse problem approach for content popularity estimation," in *Proceedings of the 9th EAI International Conference on Performance Evaluation Methodologies and Tools*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2016, pp. 33–40.

[31] W. King, "Analysis of demand paging algorithms." in *FIP Congress*, 1971, pp. 485–490.

[32] E. Gelenbe, "A unified approach to the evaluation of a class of replacement algorithms," *IEEE Transactions on Computers*, vol. 100, no. 6, pp. 611–618, 1973.

[33] O. I. Aven, E. G. Coffman, and Y. A. Kogan, *Stochastic analysis of computer storage*. Springer Science & Business Media, 1987, vol. 38.

[34] R. Hirade and T. Osogami, "Analysis of page replacement policies in the fluid limit," *Operations research*, vol. 58, no. 4-part-1, pp. 971–984, 2010.

[35] N. Tsukada, R. Hirade, and N. Miyoshi, "Fluid limit analysis of fifo and rr caching for independent reference models," *Performance Evaluation*, vol. 69, no. 9, pp. 403–412, 2012.

[36] N. Gast and B. Van Houdt, "Transient and steady-state regime of a family of list-based cache replacement algorithms," *ACM SIGMETRICS Performance Evaluation Review*, vol. 43, no. 1, pp. 123–136, 2015.

[37] J. Li, S. Shakkottai, J. Lui, and V. Subramanian, "Accurate learning or fast mixing? dynamic adaptability of caching algorithms," *arXiv preprint arXiv:1701.02214*, 2017.

[38] S. Podlipnig and L. Böszörmenyi, "A survey of web cache replacement strategies," *ACM Computing Surveys (CSUR)*, vol. 35, no. 4, pp. 374–398, 2003.

[39] 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.

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

[41] D. S. Berger, R. K. Sitaraman, and M. Harchol-Balter, "Adaptsize: Orchestrating the hot object memory cache in a content delivery network," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. USENIX Association, 2017, pp. 483–498.

[42] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Dynacache: Dynamic cloud caching," in *7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 15)*, 2015.

[43] 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.

[44] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling performance cliffs in web memory caches," in *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, 2016, pp. 379–392.

[45] S. Bansal and D. S. Modha, "Car: Clock with adaptive replacement." in *FAST*, vol. 4, 2004, pp. 187–200.

[46] Q. Zhu, A. Shankar, and Y. Zhou, "Pb-lru: a self-tuning power aware storage cache replacement algorithm for conserving disk energy," in *Proceedings of the 18th annual international conference on Supercomputing*. ACM, 2004, pp. 79–88.

[47] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache." in *USENIX Annual Technical Conference*, 1996, pp. 153–164.

[48] R. K. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain, "Overlay networks: An Akamai perspective," in *Advanced Content Delivery, Streaming, and Cloud Services*. John Wiley & Sons, 2014.

[49] E. J. Rosensweig, J. Kurose, and D. Towsley, "Approximate models for general cache networks," in *INFOCOM, 2010 Proceedings IEEE*. IEEE, 2010, pp. 1–9.