Popularity-Driven Content Caching

Suoheng Li*, Jie Xu†, Mihaela van der Schaar‡, Weiping Li§
*§Department of Electrical Engineering, University of Science and Technology of China, Hefei, Anhui, P. R. China
Email: *lzzitc@mail.ustc.edu.cn, §wpli@ustc.edu.cn
†Department of Electrical and Computer Engineering, University of Miami, Coral Gables, USA, Email: jiexu@miami.edu
‡Department of Electrical Engineering, University of California, Los Angeles, USA, Email: mihaela@ee.ucla.edu

***Abstract*—This paper presents a novel cache replacement method—Popularity-Driven Content Caching (PopCaching). PopCaching learns the popularity of content and uses it to determine which content it should store and which it should evict from the cache. Popularity is learned in an online fashion, requires no training phase and hence, it is more responsive to continuously changing trends of content popularity. We prove that the learning regret of PopCaching (i.e., the gap between the hit rate achieved by PopCaching and that by the optimal caching policy with hindsight) is sublinear in the number of content requests. Therefore, PopCaching converges fast and asymptotically achieves the optimal cache hit rate. We further demonstrate the effectiveness of PopCaching by applying it to a movie.douban.com dataset that contains over 38 million requests. Our results show significant cache hit rate lift compared to existing algorithms, and the improvements can exceed 40% when the cache capacity is limited. In addition, PopCaching has low complexity.**

I. Introduction

The last few years have witnessed the proliferation of rich media-enabled applications involving streaming of high quality media content. For instance, online social network users share nowadays not only texts and images, but also audio and video content. High-quality video is also demanded by the prevalence of retina-level resolution displays and emerging technologies such as virtual reality. As a consequence, the content that needs to be streamed in real-time has grown significantly in terms of volume, size and diversity. To provide high Quality-of-Service (QoS) with limited network resources while keeping costs low, various network architectures and algorithms have been proposed. Among them, content caching is a key technology due to its effectiveness in supporting streaming applications [1]. In fact, content caching is now considered as a basic network functionality in emerging network architectures such as Content-Centric Networking [2].

Content caching is not a new technology - Akamai [3] and its competitors have been providing content distribution and caching services for decades. However, the recent rapid growth of video traffic has led both the industry and the academia to re-engineer the content caching systems in order to accommodate this vast traffic. Cloud providers now start to launch their own caching services [4] and many websites also build their own caching systems to accelerate content distribution [5]. To improve content caching efficiency, a significant amount of research effort has been devoted to optimizing the network architecture, e.g., path optimization [6], server placement [7], content duplication strategy [8], etc. However, less attention has been devoted to improving caching strategies, i.e., which content should be cached, where and when. Today's content distribution network (CDN) providers still use simple cache replacement algorithms such as Least Recently Used (LRU), Least Frequently Used (LFU), or their simple variants [9]. These algorithms are easy to implement but may suffer major performance degradation since they ignore the future popularity that a content may acquire, which may alter the future traffic demand pattern on the network, thereby resulting in a low cache hit rate. Thus, an efficient content caching scheme should be popularity-driven, meaning that it should incorporate the future popularity of content into the caching decision making. However, designing such popularity-driven content caching schemes faces many challenges. Firstly, the future popularity of a content is not readily available at the caching decision time but rather needs to be forecasted. Secondly, the popularity of a content changes over time and hence, the content caching schemes should continuously learn, in an online fashion, in order to track such changes and adjust forecasts. Thirdly, using the estimated popularity of content to derive the optimal caching decision represents yet another challenge.

In this paper, we rigorously model how to use the popularity of content to perform efficient caching and propose an online learning algorithm, PopCaching, that learns the short-term popularity of content (i.e., how much traffic due to a content is expected in the near future) and, based on this, optimizes the caching decisions (i.e., whether to cache a content and which existing content should be replaced). The algorithm requires neither *a priori* knowledge of the popularity distribution of content nor a dedicated training phase using an existing training set which may be outdated or biased. Instead, it adapts the popularity forecasting and content caching decision online, as content is requested by end-user and its popularity is revealed over time. The contributions of this paper are summarized below:

- We propose PopCaching, an online algorithm that learns the relationship between the future popularity of a con-

This research is supported by CSC. Weiping Li and Suoheng Li acknowledge the partial support from Intel Collaborative Research Institute for Mobile Networking and Computing. Jie Xu and Mihaela van der Schaar acknowledge the support of NSF CCF 1524417. (This work was performed when Suoheng Li was visiting UCLA and Jie Xu was at UCLA.)

tent and its recent access pattern. Using the popularity forecasting result, PopCaching makes proper cache replacement decisions to maximize the cache hit rate. The amortized time complexity of PopCaching is logarithmic in the number of received requests.
- We rigorously analyze the performance of PopCaching in terms of both popularity forecasting accuracy and overall cache hit rate. We prove that the performance loss, compared with the optimal strategy that knows the future popularity of every content when making the cache decision, is sublinear in the number of content requests received by our system. This guarantees fast convergence and implies that PopCaching asymptotically achieves the optimal performance.
- We demonstrate the effectiveness of PopCaching through experiments using real-world traces from movie.douban.com which is the largest Rotten Tomatoes-like website in China. Results show that PopCaching is able to achieve a significant improvement in cache efficiency against existing methods, especially when the cache capacity at the cache server is limited (more than 100% improvement).

The remainder of the paper is organized as follows. Section II provides a review of related works. Section III introduces the system architecture and operational principles. In Section IV we formally formulate the cache replacement problem. The PopCaching algorithm is proposed in Section V. Theoretical analysis of the algorithm is presented in Section VI. Simulation results are shown in Section VII. Finally, Section VIII concludes the paper.

II. Related Work

The common approaches for content caching that have already been adopted in the Internet nowadays are summarized in [10]. As mentioned in the introduction, a significant amount of research effort was devoted to optimizing the network architecture, including routing path [6], server placement [7], content duplication strategy [11] [8], etc. For instance, [6] systematically describes the design of the Akamai Network. Authors in [7] utilize geographic information extracted from social cascades to optimize content placement. In [11], it is assumed that content popularity is given and light-weight algorithms that minimize bandwidth cost are presented. In [8], an integer programming approach to designing a multicast overlay network is proposed. However, much less attention has been devoted in literature to developing efficient caching schemes. The most commonly deployed caching schemes include Least Recently Used (LRU), Least Frequently Used (LFU) and their variants [9], which are simple but do not explicitly consider the future popularity of content when making caching decisions.

Forecasting popularity of online content has been extensively studied in the literature [12] [13]. Various solutions are proposed based on time series models such as autoregressive integrated moving average [14], regression models [15] and classification models [16]. Propagation features of content derived from social media are recently utilized to assist popularity prediction, leading to an improved forecasting accuracy [17] [18] [19]. While these works suggest ways to forecast the popularity of content, few works consider how to integrate popularity forecasting into caching decision making. In [20], propagation information of content over social media is utilized to optimize content replication strategies. An optimization-based approach is proposed in [21] to balance performance and cache replacement cost. These works develop model-based popularity forecasting schemes in which model parameters are obtained using training datasets. However, relying on specific models may be problematic in a real system since some information may not be fully available to the caching infrastructure. Moreover, because the popularity distribution of content may vary over time, relying on existing training sets, which may be outdated, may lead to inaccurate forecasting results.

To adapt to the varying popularity of content, several learning methods for content caching are proposed. In [22], each requested content is fitted into a set of predetermined models using the historical access patterns of the content. The best model that produces the smallest error is selected to predict what content should be cached. In [23], the content replacement problem is modeled as a multi-armed bandit problem and online algorithms are designed to learn the popularity profile of each content. The main drawback of these two methods is that they both learn the popularity independently across content, ignoring the similarity between content, thereby resulting in high training complexity and a slow learning speed.

III. System Overview

A. Architecture

The modules of the considered popularity-driven cache node is depicted in Fig. 1. In addition to the basic modules (i.e. *Cache Management*, *User Interface*, *Content Fetcher*, *Local Cache*, and *Request Processor*) in a conventional cache node, the popularity-driven cache node also implements *Feature Updater*, *Learning Interface*, and two databases (i.e. *Feature Database* and *Learning Database*) to enable the learning capability.

- The *Feature Updater* module is responsible for updating the raw features (e.g. the view count history) of a content, which is stored in the *Feature Database*.
- The *Learning Interface* is the module that implements the PopCaching algorithm.

B. Operational Principles

Each content request involves three sequential procedures. First, when a request arrives to the cache node, PopCaching **updates** the *Feature Database* to keep up-to-date features of the requested content. Second, PopCaching sends a **query** to the *Learning Database* with the request's context vector to get a popularity forecast of the requested content, based on which the caching decision is made. Third, when the real popularity of the content is revealed after the request has been served, PopCaching **learns** the relationship between the context vector and the popularity of content and then encodes this knowledge

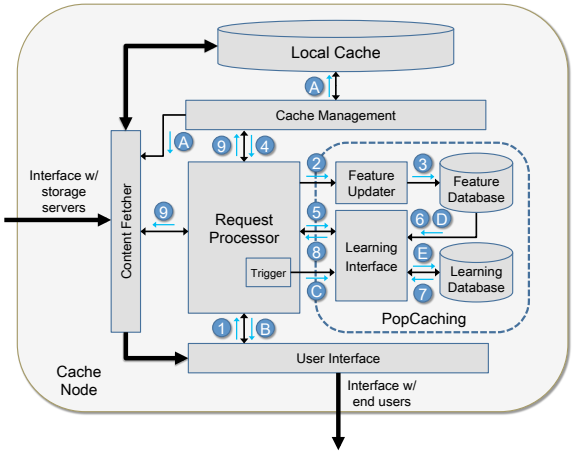

Fig. 1. Modules of a single cache node in a caching system. A typical work flow is also presented.

into the *Learning Database*. Such knowledge will be used in future requests for content with similar context vectors. The detailed operations of the popularity-driven cache node are described below. However, we note that our main focus is on the modules that enable the learning capability.

- **Update**:
  1) The *Request Processor* receives a request.
  2) The *Request Processor* initiates an update procedure with information of the received request.
  3) The *Feature Updater* calculates the latest feature values and writes them into the *Feature Database*.
- **Query**:
  4) The *Cache Management* module checks if the requested content is in local cache.
  5) If the requested content is not found, the PopCaching algorithm decides whether or not to cache the content.
  6) The PopCaching algorithm extracts the context vector from the *Feature Database*.
  7) The PopCaching algorithm searches the *Learning Database* with the context vector and makes a caching decision based on the forecasted popularity of the content.
  8) The PopCaching algorithm returns its decision.
  9) Based on the caching decision, the requested content is either cached locally or not.
  A) If the content is decided to be cached, *Cache Management* module updates the local cache.
  B) The request is served.
- **Learn**:
  C) The *Request Processor* triggers a learning process.
  D) PopCaching extracts the context vector and revealed popularity from the *Feature Database*.
  E) PopCaching updates the *Learning Database* with the context vector and revealed popularity.

IV. System Model

In this section, we formalize the popularity-driven caching problem.

Consider the setting where a content provider has a set of content $\mathcal{C} = \{1, 2, ..., C\}$ that can be requested by end users.[1] In practice, this set may be very large and we may have millions of content. To provide services with high quality, the content provider sets up a caching system (e.g. uses a 3rd-party caching service or sets up its own). The caching system aims to offload requests to its local cache at its best effort. In this paper, we focus on a single cache node in such a system where cache nodes operate independently. Let $s < C$ be the capacity of the node, i.e., the maximum number of content the node can store in its local cache. We assume that all content are of the same size,[2] so the node can hold up to s content. We denote requests for content by $Req = \{req_1, req_2, ..., req_k, ..., req_K\}$, which come in sequence. Each request in this set is represented by $req_k = \langle c(k), x(k), t(k) \rangle, \forall 1 \leq k \leq K$, where $c(k) \in \mathcal{C}$ is the content being requested, $t(k)$ is the time of the request (e.g. when the end user initiates the request), and $x(k)$ is the context vector of the request. The context $x \in \mathbb{R}^d$ is a d-dimensional vector that describes under what circumstance the request is made, which may include features like the user's profile, the property of the requested content, and system states. Without loss of generality, we normalize the context and let $x \in [0, 1]^d \triangleq \mathcal{X}$.

For each coming request req_k, we first check if it can be handled by the node's local cache. Formally, let $Y_k(c(k)) \in \{0, 1\}$ represent whether content $c(k)$ is in the local cache at the time when req_k needs to be served. For instance, $Y_k(c(k)) = 1$ means that req_k can be served by the local cache. Furthermore, we use a binary vector $Y_k = [Y_k(1), Y_k(2), ..., Y_k(C)]$ to denote the whole cache status at time $t(k)$, where $Y_k(c)$ is the c-th element in Y_k. We want to emphasize that Y_k is only used for analysis and our algorithm does not require storing the whole Y_k.

When $c(k)$ is not found in the local cache, the node retrieves it from the storage servers and decides whether to store $c(k)$ in its local cache. Specifically, the node may replace an existing content with the new content $c(k)$. Let $c_{old}(k) \in \{c : Y_k(c) = 1\}$ denote the old content that is replaced by $c(k)$. Hence, the cache status vector is changed to Y_{k+1} according to the following equation:

$$Y_{k+1}(c) = \begin{cases} 0 & \text{if } c = c_{old} \\ 1 & \text{if } c = c(k) \\ Y_k(c) & \text{otherwise} \end{cases}$$

A caching policy prescribes, for all k, whether or not to store a content $c(k)$ that is not in the local cache and, if yes, which existing content $c_{old}(k)$ should be replaced. Formally, a caching policy can be represented by a function $\pi : (\{0, 1\}^C, \mathcal{C}, \mathcal{X}) \longmapsto \{0, 1\}^C$ that maps the current cache

[1] While we use C to denote the total number of content, it is used for theoretical analysis and our algorithm does not need to know this number.

[2] This same size assumption can be justified as follows: each content is split into chunks of a fixed size and each chuck is then considered as a content. This is a common practice in real world systems. For instance, the widely adopted Dynamic Adaptive Streaming over HTTP (DASH) protocol usually splits each video into several equal-sized chunks.

status vector, the requested content and the context vector of the request to the new cache status vector. Whenever a request req_k is served, the cache status is updated according to π:

$$Y_{k+1} = \pi(Y_k | c(k), x(k)) \tag{1}$$

To evaluate the efficiency of the caching system, we use cache hit rate $H(K, \pi)$, which is defined as the percentage of requests that are served from the local cache up to the K-th request. In addition, $H(\pi)$ denotes the long-term average hit rate, which is defined as follows:

$$H(\pi) = \lim_{K \to \infty} H(K, \pi) = \lim_{K \to \infty} \frac{1}{K} \sum_{k=1}^{K} Y_k(c(k)) \tag{2}$$

In this way, $H(\pi)$ describes how the caching system performs in the long term by adopting the caching policy π. Note that even though π is not explicitly written on the right hand side of (2), the evolution of the cache status vector Y_k is governed by π.

Our objective is to find a policy π that maximizes the overall cache hit rate so that we achieve the highest cache efficiency.

$$\pi^* = \arg\max_{\pi} H(\pi) \tag{3}$$

V. PopCaching Algorithm

A. Algorithm Overview

The PopCaching algorithm is presented in Figure 2. For each incoming request req_k, we first extract the features of the request and update the *Feature Database* module. Specifically, we use a sliding window to log the recent access history of each content. We then examine the local cache to see whether the requested content $c(k)$ has already been cached. If $c(k)$ exists in the local cache, then the end user is served using the content copy in the local cache; otherwise, we fetch $c(k)$ from the storage servers to serve the end user. In the second case, PopCaching makes a forecast on the future popularity of $c(k)$ and decides whether or not to push $c(k)$ in the local cache and which existing content should be removed from the local cache. To do this, PopCaching extracts the context vector $x(k)$ associated with the current request from the *Feature Database* and issues a forecast of the request rate for $c(k)$, denoted by $\tilde{M}_k$, using the popularity forecast algorithm that will be introduced in the next subsection. Then, PopCaching compares $\tilde{M}_k$ with the popularity estimate of the least popular content already in the local cache, denoted by $\tilde{M}^{least}$. To quickly find the least popular content, PopCaching maintains a priority queue Q that stores the cached content along with their estimated request rates. The top element of Q is simply the least popular content. If $\tilde{M}_k > \tilde{M}^{least}$, then PopCaching replaces the least popular content c^{least} with $c(k)$ in the local cache and update Q accordingly; otherwise, PopCaching does nothing to the local cache. To keep the popularity estimates of content in the local cache up to date, PopCaching periodically updates the forecast for the cached content after every ϕ requests.

procedure PROCESSONEREQUEST(req_k)
  Update the feature database for $c(k)$
  if $c(k)$ is in the local cache **then**
    Serve the end user from the local cache
  else
    Fetch $c(k)$ from the storage servers
    Serve the end user with $c(k)$
    Extract $x(k)$ from the feature database
    $\tilde{M}_k \leftarrow$ Estimate($x(k)$)
    $\langle \tilde{M}^{least}, c^{least} \rangle \leftarrow$ the top element in Q
    if $\tilde{M}_k > \tilde{M}^{least}$ **then** ▷ update local cache
      Remove the top element from Q
      Insert $\langle \tilde{M}_k, c(k) \rangle$ into Q
      Replace c^{least} with $c(k)$ in the local cache
    end if
  end if
  if $k \mod \phi = 0$ **then**
    Re-estimate the request rate for all content in Q
    Rebuild the priority queue Q
  end if
  $c(k)$'s popularity M_k is revealed after time θ
  Call Learn($x(k)$, M_k)
end procedure

Fig. 2. Procedure of processing a single request. **Learn** and **Estimate** are two procedures defined in Section V-B.

B. Popularity Forecasting

Each request req_k is characterized by its context vector $x(k)$ of size d and hence, it can be seen as a point in the context space $\mathcal{X} = [0, 1]^d$. At any time, the context space $\mathcal{X}$ is partitioned into a set of hypercubes $\mathcal{P}(k) = \{P_i\}$. These hypercubes are non-overlapping and $\mathcal{X} = \bigcup_{P_i \in \mathcal{P}(k)} P_i$ for all k. The partitioning process will be described in the next subsection. Clearly, $x(k)$ belongs to a unique hypercube in the context space partition, denoted by $P^*(k)$. For each hypercube $P_i \in \mathcal{P}(k)$, we maintain two variables $\mathcal{N}(P_i)$ and $\mathcal{M}(P_i)$ to record the number of received requests in P_i and the sum of the revealed future request rate for those requests, respectively. The forecasted future popularity for requests with contexts in this partition P_i is computed using the sample mean estimate $\tilde{M}(P_i) = \mathcal{M}(P_i)/\mathcal{N}(P_i)$.

The popularity forecasting is done as follows. When a request req_k with context $x(k)$ is received, PopCaching first determines the hypercube $P^*(k)$ that $x(k)$ belongs to in the current partitioning $\mathcal{P}(k)$. The forecasted popularity for req_k is simply $\tilde{M}(P^*(k))$. After the true popularity M_k of the content of req_k is revealed, the variables of $P^*(k)$ is updated to $\mathcal{M}(P^*(k)) \leftarrow \mathcal{M}(P^*(k)) + M_k$ and $\mathcal{N}(P^*(k)) \leftarrow \mathcal{N}(P^*(k)) + 1$. Depending on the new value of $\mathcal{N}(P^*(k))$, the hypercube may split into smaller hypercubes and hence, the partitioning of the context space evolves. The next subsection describes when and how to split the hypercubes.

C. Adaptive Context Space Partitioning

This subsection describes how to build the partition $\mathcal{P}$ as requests are received over time. Let l_i denote the level of a hypercube P_i which can also be considered as the

procedure LEARN($x(k), M_k$) ▷ Learn from req_k
    Determines $P^*(k)$ that $x(k)$ belongs to
    $\mathcal{N}(P^*(k)) \leftarrow \mathcal{N}(P^*(k)) + 1$
    $\mathcal{M}(P^*(k)) \leftarrow \mathcal{M}(P^*(k)) + M_k$
    SPLIT($P^*(k)$)
end procedure
procedure ESTIMATE($x(k)$) ▷ Estimate $\tilde{M}_k$
    Determines $P^*(k)$ that $x(k)$ belongs to
    return $\mathcal{M}(P^*(k))/\mathcal{N}(P^*(k))$
end procedure

Fig. 3. Procedures of **Learn** and **Estimate** for a single request

procedure SPLIT(P_i)
    if $\mathcal{N}(P_i) \geq z_1 2^{z_2 \cdot l_i}$ **then**
        Split P_i into 2^d hypercubes $\{P_j\}$
        Set $\mathcal{M}(P_j) \leftarrow \mathcal{M}(P_i)$ for each P_j
        Set $\mathcal{N}(P_j) \leftarrow \mathcal{N}(P_i)$ for each P_j
        Set level $l_j \leftarrow l_i + 1$ for each P_j
    end if
end procedure

Fig. 4. The procedure of adaptive context space partitioning

generation of this hypercube. At the beginning, the partition $\mathcal{P}$ contains only one hypercube which is the entire context space $\mathcal{X}$ and hence, it has a level 0. Whenever a hypercube P_i accumulates sufficiently many sample requests (i.e. $\mathcal{N}(P_i)$ is greater than some threshold $\zeta(l_i)$), we equally split it along each dimension to create 2^d smaller hypercubes. Each of these child hypercubes P_j has an increased level of $l_j \leftarrow l_i + 1$ and inherits the variables $\mathcal{M}(P_i)$ and $\mathcal{N}(P_i)$ from its parent hypercube, i.e. $\mathcal{M}(P_j) \leftarrow \mathcal{M}(P_i)$ and $\mathcal{N}(P_j) \leftarrow \mathcal{N}(P_i)$. Due to this splitting process, a hypercube of level l_i has length 2^{-l_i} along each axis.

The threshold $\zeta(l_i)$ determines the rate at which the context space is partitioned. Partitioning the context space too fast or too slow will both cause inaccurate estimates. Therefore, in PopCaching, $\zeta(l_i)$ is designed to have the form $z_1 2^{z_2 \cdot l_i}$ where $z_1 > 0$ and $z_2 > 0$ are two parameters of the algorithm. In Section VI, we will show that by carefully selecting the parameters, PopCaching can achieve the optimal performance asymptotically. Figure 5 illustrates how PopCaching makes a forecast of the popularity of a requested content, learns from that request after its popularity is revealed, and updates the partition of the context space accordingly.

VI. PERFORMANCE ANALYSIS

In this section, we analyze the performance of the PopCaching algorithm. We first bound the popularity forecasting error, and then use it to derive the bound on the overall cache hit rate $H(\pi)$.

A. Upper Bound on the Popularity Forecast Error

To enable rigorous analysis, we make the following widely adopted assumption [24] [25] that the expected popularity of similar content are similar. This is formalized in terms of a uniform Lipschitz continuity condition.

Assumption 1. *(Uniform Lipschitz continuity) There exists a positive real number $\beta > 0$ such that for any two requests k and k', we have $\mathbb{E}|M_k - M_{k'}| \leq \beta||x(k) - x(k')||$ where $||\cdot||$ represents the Euclidean norm.*

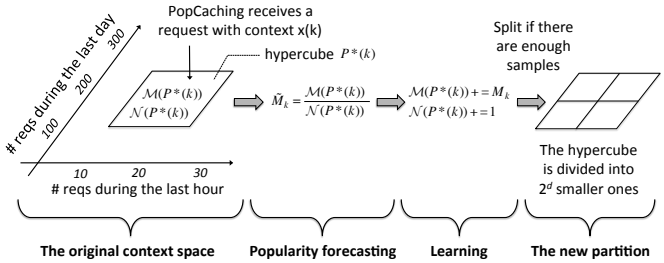

Fig. 5. An illustration of popularity forecasting and adaptive context space partitioning in which, for illustration purposes, we suppose that the context space is two-dimensional (i.e. $d = 2$).

We now bound the forecast error made by the PopCaching algorithm.

Proposition 1. *The expected total forecast error for the first K requests, $\mathbb{E}\sum_{k=1}^{K}|\tilde{M}_k - M_k|$, is upper bounded by $\tilde{O}(K^\mu)$ for some $\mu < 1$. If we choose $z_2 = 0.5$, then $\mu = \frac{d}{d+0.5}$.*

The error bound proved in Proposition 1 is sublinear in K, which implies that as $K \to \infty$, $\mathbb{E}\sum_{k=1}^{K}|\tilde{M}_k - M_k|/K \to 0$.[3]

In other words, PopCaching makes the optimal prediction as sufficiently many content requests are received. The error bound also tells how much error would have been incurred by running PopCaching for any finite number of requests. Hence, it provides a rigorous characterization on the learning speed of the algorithm.

B. Lower Bound on the Cache Hit Rate

In the previous subsection, we showed that the popularity forecast error is upper-bounded sublinearly in K. In this subsection, we investigate the lower bound on the cache hit rate that can be achieved by PopCaching and the performance loss of PopCaching compared to an oracle optimal caching algorithm that knows the future popularity of all content.

We first note that the achievable cache hit rate $H(\pi)$ depends not only on the caching policy π but also the access patterns of requests. For instance, a more concentrated access pattern implies a greater potential to achieve a high cache hit rate. To bound $H(\pi)$, we divide time into periods with each period containing ϕ requests [4]. In the m-th period (i.e. requests $k : m\phi < k \leq (m+1)\phi$), let M^{sort} be the sorted vector of the popularity of all content in $\{k : m\phi < k \leq (m+1)\phi\}$. The normalized total popularity of the j most popular content in this period is thus $\frac{\sum_{i=1}^{j} M_i^{sort}}{\sum_{i=1}^{C} M_i^{sort}}$. Recall that C is the total number of content files. Let the function

$$f(j) = 1 - \frac{\sum_{i=1}^{j} M_i^{sort}}{\sum_{i=1}^{C} M_i^{sort}}. \quad (4)$$

be the normalized total rate of the $(C - j)$ least popular content. Clearly, $f(j)$ is a monotonically decreasing function and $f(C) = 0$. Note that (1) in different periods, $f(j)$ can

[3]The complete analysis and proofs can be found in the Appendix B.

[4]For analysis simplicity, we assume that K is a multiple of ϕ. Generalization is straightforward.

be different; (2) we do not make any assumption on the popularity distribution and $1 - f(j)$ is simply the probability mass function of M^{sort}.

The next proposition connects the popularity forecasting error to the achievable cache hit rate.

Proposition 2. *For any time period m, if the popularity forecasting error satisfies $|\tilde{M}_i^{sort} - M_i^{sort}| \leq \Delta M, \forall i \in \{1, 2, ..., C\}$, then the achieved cache hit rate is at least $1 - f(s) - \frac{2s}{\phi} - \frac{2s \cdot \Delta M}{\sum_{i=1}^{C} M_i^{sort}}$ in that period.*

To understand the bound in 2, we split it into two parts. The first part $1 - f(s) - \frac{2s}{\phi}$ depends on the access pattern $f(\cdot)$ and the cache capacity s but not the forecasting error ΔM. Therefore it represents how well a caching policy can perform in the best case (i.e. when it makes no popularity forecasting errors). As expected, if the access pattern is more concentrated (i.e. $f(s)$ is smaller), the cache hit rate is higher. When the period ϕ is sufficiently long, then as the cache capacity $s \to C$, the cache hit rate $(1 - f(s) - \frac{2s}{\phi}) \to 1$. The second part $\frac{2s \cdot \Delta M}{\sum_{i=1}^{C} M_i^{sort}}$ measures the cache hit rate loss due to popularity forecasting errors. A larger forecasting error ΔM leads to a bigger loss.

By combining Proposition 1 and Proposition 2, we show in Theorem 1 that PopCaching achieves the optimal performance asymptotically.

Theorem 1. *PopCaching achieves a cache hit rate that asymptotically converges to that obtained by the oracle optimal strategy, i.e., $\mathbb{E}H(\pi^*) = \mathbb{E}H(\pi_0)$.* [5]

Proof. Since $f(j)$ and ΔM may vary among different time periods, we now use $f_m(j)$ and ΔM_m to denote their corresponding values in the m-th period. Let M^{inf} be the infimum of $\sum_{i=1}^{C} M_i^{sort}$ over all time periods. According to Proposition 2 and utilizing $\phi \gg s$:

$$\begin{aligned}
\mathbb{E}(H(\pi^*) - H(\pi_0)) &\leq \lim_{K \to \infty} \frac{\phi}{K} \mathbb{E} \sum_{m=0}^{\frac{K}{\phi}-1} \frac{2s \cdot \Delta M_m}{\sum_{i=1}^{C} M_i^{sort}} \\
&\leq \lim_{K \to \infty} \frac{2s\phi}{K} \cdot \mathbb{E} \frac{\sum_{k=1}^{K} |\tilde{M}_k - M_k|}{M^{inf}} \\
&= \lim_{K \to \infty} \frac{\tilde{O}(K^{\frac{d}{d+1/2}})}{K} = 0 \qquad (5)
\end{aligned}$$

□

VII. Experimental Results

A. Dataset

We use data crawled from movie.douban.com as our main dataset for the evaluation of PopCaching. The website movie.douban.com is one of the largest social platforms devoted to film and TV content reviews in China. On top of traditional functionalities of a social network platform, it provides a Rotten Tomatoes-like database, where a user can post comments (e.g. short feedback to a movie), reviews (e.g. a long article for a movie), ratings, etc. In our experiments, we suppose that there is an online video provider who provides video content to users on movies.douban.com. To simulate the content request process, we take each comment on a video content by a user as the request for this content. More specifically, we assume that every movie comment in our dataset is a downloading/streaming request towards our hypothesized video provider and the time when the comment is posted is considered as the time when the request is initiated. Even though movie.douban.com may not actually store any encoded video, using the comment process to simulate the request process can be well justified: it is common that people post comments on the video content right after they have watched it and hence, the comment data should exhibit similar access patterns to those of content request data observed by an online video provider.

To obtain data from movie.douban.com, we implemented a distributed crawler to enumerate videos, accessible comments[6] and active users (i.e., users who have posted at least one comment.) To guarantee the correctness of the main dataset, we also wrote a random crawler to get a small dataset and cross-checked with the main dataset. As an overview, the main dataset contains 431K (431 thousand) unique videos, among which 145K are active (i.e., videos having at least one comment), 46M (46 million) accessible comments, and 1.3M active users.

B. Simulator Setup

We build a discrete event simulator according to Fig. 1 and evaluate the performance of PopCaching. The context vector in this experiment has four dimensions ($d = 4$): how many times the content is requested during the last 5 hours, 30 hours, 5 days, and 30 days respectively. Besides, there are four parameters in our algorithm, θ, ϕ, z_1, and z_2. The simulation results presented in this section are all obtained with $\theta = 1000$ seconds, $\phi = 10000$, $z_1 = 2$, and $z_2 = 0.5$ if not explicitly clarified.

C. Benchmarks

We compare the performance of PopCaching with benchmarks listed below:

- **First In First Out (FIFO) [26].** The cache acts as a pipe: the earliest stored content is replaced by the new content when the cache is full.
- **Least Recently Used (LRU) [27].** The cache node maintains an ordered list to track the recent access of all cached content. The least recently accessed one is replaced by the new content when the cache is full.
- **Least Frequently Used (LFU) [28].** The cache node maintains an ordered list to track the numbers of access of all content. The least frequently used one is replaced by the new content when the cache is full. Note that LFU may have very poor long-term performance due to

[5] We have made an implicit assumption here that all requests during a time period is randomly distributed. Hence it is less likely to see consecutive requests for unpopular content and the best caching strategy π^* is to just store the most popular ones during that period.

[6] A comment may be deleted by its owner or administrators. An inaccessible comment has a unique ID but cannot be downloaded.

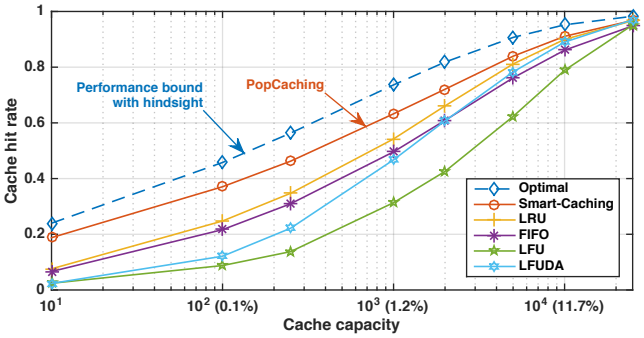

Fig. 6. Cache hit rate under different cache capacity. The percentage number in brackets is the ratio of the cache capacity to the total number of content, i.e., s/C.

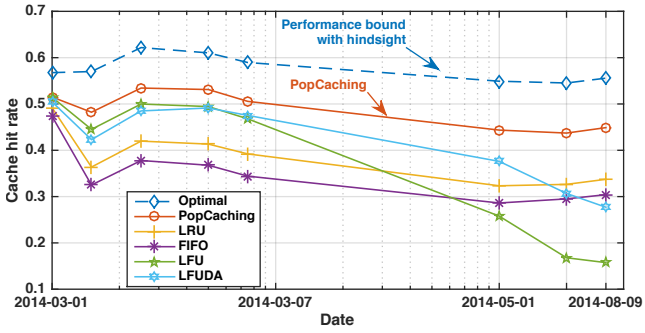

Fig. 7. Cache hit rate over time for the first 3 million requests starting from 1 March 2014. Cache capacity is 250 ($s = 250$). Each point in the figure represents the percentage of cache hit within the window between itself and its proceeding point.

a cache pollution problem: if a previously popular content becomes unpopular, LFU may still hold it for a long time, resulting in inefficient utilization of the cache.

- **Least Frequently Used with Dynamic Aging (LFUDA) [29].** LFUDA is a variant of LFU that tries to solve the cache pollution problem by maintaining a cache age counter that punishes the access frequency of old content.
- **Optimal Caching.** The cache node runs Belady's MIN algorithm [30] that achieves theoretically optimal performance with hindsight. Note that Belady's algorithm is not implementable in a real system due to the fact that it needs future information.

D. Performance Comparison

Figure 6 shows the overall average cache hit rates achieved by PopCaching and the benchmark algorithms under various cache capacity. As can be seen, PopCaching significantly outperforms all the other algorithms in all the simulations. In particular, the performance improvement against the second best solution exceeds 100% when the cache capacity is small. This is because the benchmark algorithms does not take the future popularity of content into account when making the caching decisions. They consider only the current popularity of the content which may differ from the future popularity, thereby causing more cache misses. Moreover, the benchmark algorithms treat each content independently without trying to learn from the past experience the relationship between popularity and the context information. For instance, when a content is evicted from the local cache, all knowledge about this content is lost and cannot be used for future decision making. Instead, PopCaching learns continuously and stores the learned knowledge into the learning database which can be utilized in the future. The advantage of PopCaching becomes greater when the cache capacity is smaller since more careful caching decisions need to be made. To illustrate the enormous improvement by adopting PopCaching on reducing the cache storage requirement, consider a common target cache hit rate of 0.5. In this case, PopCaching requires a cache capacity of 300 while LFU needs a cache capacity of 3000.

In Figure 7, we plot the cache hit rate versus the date that a request is initiated in order to show how the caching performance varies over time. Each point of a curve in the figure represents the percentage of cache hit within the time window between itself and the proceeding point. We draw the figure for only a time duration of 180 days because afterwards the cache hit rates of all algorithms converge (except LFU and LFUDA). Several points are worth noting: (1) On the first day, all algorithms show similar performance. This is because the distribution of content popularity is relatively stable during a single day, thus making it easy to make cache decisions. Then, the advantage of PopCaching becomes obvious as more requests arrive. In this time, Popcaching successfully learns from the large volume of requests and hence makes accurate popularity predictions. (2) The cache hit rate achieved by PopCaching shown in this figure is not always increasing. This is due to the fact that the curve is generated for a single realization of the request arrival process. When averaging over a large number of realizations, the expected cache hit rate is expected to be non-decreasing. Unfortunately, our dataset lacks a large number of independent realizations and hence, we are not able to plot such a figure. Nevertheless, Figure 7 is still useful to illustrate the learning behavior of PopCaching and its superior performance over the existing solutions. (3) LFU and LFUDA fail to track the changing trends of content popularity: the cache hit rate of both algorithms drop rapidly after a few tens of days. This is because LFU makes caching decisions using the past popularity of content which becomes outdated as time goes by. LFUDA alleviates this problem by introducing a cache age counter but does not completely eliminate it. In contrast, PopCaching responses quickly to the changes in popularity distribution, and therefore maintains a steady cache hit rate.

TABLE I
CACHE HIT RATE UNDER DIFFERENT VALUES OF ϕ.

ϕ	$s = 100$	$s = 1000$	$s = 10000$
10^2	37.56	63.73	91.05
10^3	37.57	63.78	91.05
10^4	37.52	63.95	91.06
10^5	37.06	63.99	90.97

Table I shows the impact of choosing different algorithm parameters on the achievable caching performance for various cache capacity. As we can see, the cache hit rate does not significantly change even if different ϕ is used. This is much desired in practice since the algorithm is robust to different system settings.

TABLE II
COMPARISON OF RUNNING SPEED (INCLUDING SIMULATOR OVERHEAD) UNDER DIFFERENT CACHE CAPACITY. RESULTS ARE SHOWN IN NUMBER OF THOUSAND REQUESTS PER SECOND.

Capacity (s)	**PopCaching**	LRU	FIFO	LFU	LFUDA
100	28.9	1279	1463	24.5	17.3
10000	25.8	725	864	9.37	5.26

Finally, we compare the running speed of PopCaching with the benchmarks in Table II. In our implementations, all algorithms are written in pure Python [31] and are single-threaded. LRU is implemented in ordered dictionary and LFU in double-linked list with dictionary. All results are measured on a mainstream laptop with a 2.8GHz CPU. As we can see in Table II, PopCaching processes more than 20 thousand requests per second and outperforms both LFU and LFUDA. This means that PopCaching can be integrated into existing systems without introducing a significant overhead. Note that constant time algorithms such as LRU have an obvious advantage in this comparison, but they may not benefit a content caching system much since in such a system the bottleneck of running speed is usually not the caching algorithm.

VIII. CONCLUSION

This paper proposed a novel online learning approach to perform efficient, and fast cache replacement. Our algorithm (PopCaching) forecasts the popularity of content and makes cache replacement decisions based on it. PopCaching does not directly learn the popularity of each content. Instead, the algorithm learns the relationship between the future popularity of a content and the context in which the content is requested, thus utilizing the similarity between the access patterns of different content. The learning procedure takes place online and requires no *a priori* knowledge of popularity distribution or a dedicated training phase. We prove that the performance loss of PopCaching, when compared to the optimal strategy, is sublinear in the number of processed requests, which guarantees a fast speed of learning as well as the optimal cache efficiency in the long term. Extensive simulations with real world traces also validate the effectiveness of our algorithm, as well as its insensitivity to parameters and fast running speed.

APPENDIX

A. Some Lemmas

Lemma 1. *For any request that falls into hypercube P, the expected estimation error for that request is upper bounded by $\beta\sqrt{d}\frac{2^{z_2}\cdot\frac{2-\sqrt{2}}{2}+(1-2^{z_2})2^{(z_2-\frac{1}{2})l}+4(1-2^{z_2-\frac{1}{2}})/z_1}{2^{lz_2}(1-2^{z_2-\frac{1}{2}})}$ for $l\geq 1$ and $\beta\sqrt{d}$ for $l=0$, where l is the level of P.*

Proof. For $l=0$, there is only one hypercube, so the estimation error is bounded by $\mathbb{E}|\tilde{M}_k - M_k| \leq \beta||x(k')-x(k)|| \leq \beta\sqrt{d}$. For $l\geq 1$, since $P(x(k))$ is at level l, it contains $\lceil z_1 2^{z_2}\rceil$ samples at level 0, $\lceil z_1 2^{2z_2}\rceil - \lceil z_1 2^{z_2}\rceil$ samples at level 1, ..., and $\mathcal{N}_P - \lceil z_1 2^{z_2}\rceil$ samples at level l_i. Let $\sum_{k'}$ denotes the summation over all requests k' that are in P or P's ancestors, the expected estimation error for req_k is bounded by

$$\begin{aligned}
&\mathbb{E}|\tilde{M}_k - M_k| \leq \frac{\sum_{k'}\beta||x(k')-x(k)||}{\mathcal{N}_P}\\
=&\beta\sqrt{d}\frac{\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l-1}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)2^{-i/2}}{\mathcal{N}_P}\\
&\frac{+\ldots+(\mathcal{N}_P - \lceil z_1 2^{l\cdot z_2}\rceil)2^{-l/2}}{\mathcal{N}_P}\\
<&\beta\sqrt{d}\frac{z_1 2^{z_2} + \sum_{i=1}^{l-1}(z_1 2^{(i+1)z_2} - z_1 2^{i\cdot z_2})2^{-i/2} + 4}{z_1 2^{l\cdot z_2}}\\
=&\beta\sqrt{d}\cdot(C_1\cdot 2^{-z_2 l} + C_2\cdot 2^{-\frac{1}{2}l}) \qquad (6)
\end{aligned}$$

where C_1 and C_2 are constant numbers that are only related to system parameters z_1 and z_2:

$$C_1 = \frac{2^{z_2}\cdot\frac{2-\sqrt{2}}{2}}{1-2^{z_2-\frac{1}{2}}} + \frac{4}{z_1},\ C_2 = \frac{1-2^{z_2}}{1-2^{z_2-\frac{1}{2}}} \qquad (7)$$

□

B. Proof of Proposition 1

Proof. From (6) we know that the upper bound of the expected estimation error is related to the level of the hypercube, where higher level leads to smaller error. Consider the worst case scenario when each coming request always hits the hypercube with the least level. Let l be the highest level of all hypercubes. Then there will be $\lceil z_1 2^{z_2}\rceil$ samples entering the hypercube at level 0, $2^{id}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)$ samples entering hypercubes at level i ($1\leq i\leq l-1$), and remaining samples entering level l.

$$\begin{aligned}
&\mathbb{E}\sum_{k=1}^{K}|\tilde{M}_k - M_k|\\
<&\beta\sqrt{d}\{\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l}\\
&[2^{id}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)(C_1 2^{-z_2 i} + C_2 2^{-\frac{i}{2}})]\}\\
<&\beta\sqrt{d}\{z_1 2^{z_2} + 1 + C_1\frac{2^{d-z_2}}{1-2^{d-z_2}} + C_2\frac{2^{d-\frac{1}{2}}}{1-2^{d-\frac{1}{2}}} +\\
&z_1(2^{z_2}-1)[2^d C_1\frac{2^{dl}-1}{2^d-1} + 2^{d+z_2-\frac{1}{2}}C_2\frac{2^{(d+z_2-\frac{1}{2})l}-1}{2^{d+z_2-\frac{1}{2}}-1}]\}\\
\leq&\beta\sqrt{d}(C_3 + |C_4|2^{dl} + |C_5|2^{(d+z_2-\frac{1}{2})l}) \qquad (8)
\end{aligned}$$

Meanwhile, we also derive the relationship between K and l:

$$\begin{aligned}
K \geq&\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l-1}[2^{id}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)]\\
>&[z_1(2^{z_2}-1)-1]2^{(d+z_2)l} \qquad (9)
\end{aligned}$$

thereby

$$2^l < [z_1(2^{z_2}-1)-1]^{-\frac{1}{d+z_2}}\cdot K^{\frac{1}{d+z_2}} \qquad (10)$$

then

$$\begin{aligned}
&\mathbb{E}\sum_{k=1}^{K}(\tilde{M}_k - M_k)\\
<&\beta\sqrt{d}\left(C_3 + |C_6|K^{\frac{d}{d+z_2}} + |C_7|K^{\frac{d+z_2-1/2}{d+z_2}}\right) \qquad (11)
\end{aligned}$$

where

$$\begin{cases}
C_3 = z_1 2^{z_2} + 1 + C_1\frac{2^{d-z_2}}{1-2^{d-z_2}} + C_2\frac{2^{d-\frac{1}{2}}}{1-2^{d-\frac{1}{2}}}\\
C_6 = z_1(2^{z_2}-1)\frac{2^d}{2^d-1}C_1[z_1(2^{z_2}-1)-1]^{-\frac{d}{d+z_2}}\\
C_7 = z_1(2^{z_2}-1)\frac{2^{d+z_2-\frac{1}{2}}}{2^{d+z_2-\frac{1}{2}}-1}C_2[z_1(2^{z_2}-1)-1]^{-\frac{d+z_2-\frac{1}{2}}{d+z_2}}
\end{cases}$$

The equation shows that the sum of the expected estimation error is upper bounded by $\tilde{O}(|C_6|K^{\frac{d}{d+z_2}} + |C_7|K^{\frac{d+z_2-1/2}{d+z_2}})$, and when we choose $z_2 = \frac{1}{2}$, it becomes $\tilde{O}(K^{\frac{d}{d+1/2}})$.[7] □

C. Proof of Proposition 2

Proof. In this proof we only consider the case where the capacity of cache is smaller than the number of all content, that is $s < C$. When $s \geq C$, we can just cache all content and always achieve the best cache hit rate, where our conclusion in this proof still holds but is not meaningful.

Based on our algorithm, we always try to fill the cache with the s-most popular content. Normally we would choose $\{M_1^{sort}, M_2^{sort}, ..., M_s^{sort}\}$, but due to estimation error, we may not correctly choose the s-largest values. Assuming we have chosen $\{M_{i_1}^{sort}, M_{i_2}^{sort}, ..., M_{i_s}^{sort}\}$ based on the estimated sorting below:

$$\tilde{M}_{i_1}^{sort} \geq \tilde{M}_{i_2}^{sort} \geq ... \geq \tilde{M}_{i_s}^{sort} \geq ... \geq \tilde{M}_{i_C}^{sort} \tag{12}$$

Since the sum of the s-largest elements in a set should be no less than the sum of any s elements in the set, we have:

$$\sum_{j=1}^{s} \tilde{M}_{i_j}^{sort} \geq \sum_{i=1}^{s} \tilde{M}_{i}^{sort} \tag{13}$$

According to (4) and $\tilde{M}_i^{sort} \geq M_i^{sort} - \Delta M$, we know $\sum_{i=1}^{s} \tilde{M}_i^{sort} \geq \sum_{i=1}^{s}(M_i^{sort} - \Delta M) \geq (1 - f(s))\sum_{i=1}^{C} M_i^{sort} - s \cdot \Delta M$, which intuitively means that we can always find s elements in $\{\tilde{M}_{i_1}^{sort}, \tilde{M}_{i_2}^{sort}, ..., \tilde{M}_{i_C}^{sort}\}$ where the sum of them is at least $(1 - f(s))\sum_{i=1}^{C} M_i^{sort} - s \cdot \Delta M$. Combining this with (13), we have

$$\sum_{j=1}^{s} \tilde{M}_{i_j}^{sort} \geq \sum_{i=1}^{s} \tilde{M}_{i}^{sort} \geq (1-f(s))\sum_{i=1}^{C} M_i^{sort} - s\Delta M \tag{14}$$

At each time period (e.g. $m\phi < k \leq m\phi + \phi$), each corresponding content of $\{M_{i_1}^{sort}, M_{i_2}^{sort}, ..., M_{i_s}^{sort}\}$ is cached either before this time period or after its first cache miss. Hence we bound the worst case cache hit rate during each time period as

$$\begin{aligned}
&\frac{1}{\phi}\sum_{k=m\phi+1}^{m\phi+\phi} Y_k(c(k)) = \frac{\sum_{j=1}^{s}\lfloor M_{i_j}^{sort}\Delta t - 1\rfloor}{\sum_{j=1}^{C} M_j^{sort}\Delta t} \\
\geq &\frac{(1 - f(s))\sum_{j=1}^{C} M_j^{sort} - 2s(\Delta M + 1/\Delta t)}{\sum_{j=1}^{C} M_j^{sort}} \\
= &(1 - f(s)) - \frac{2s \cdot \Delta M}{\sum_{j=1}^{C} M_j^{sort}} - \frac{2s}{\phi}
\end{aligned} \tag{15}$$

□

REFERENCES

[1] F. Dobrian, V. Sekar, A. Awan, I. Stoica, D. Joseph, A. Ganjam, J. Zhan, and H. Zhang, "Understanding the impact of video quality on user engagement," in *Proc. SIGCOMM'11*, 2011, pp. 362–373.

[2] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content," *Communications of the ACM*, vol. 55, no. 1, pp. 117–124, 2012.

[3] J. Dilley, B. Maggs, J. Parikh, H. Prokop, R. Sitaraman, and B. Weihl, "Globally distributed content delivery," *IEEE Internet Computing*, vol. 6, no. 5, pp. 50–58, 2002.

[7] when $z_2 = \frac{1}{2}$ the bound actually becomes $\tilde{O}(K^{\frac{d}{d+1/2}}\log K)$, but due to $\tilde{O}$ notation, the logarithmic term is suppressed.

[4] Amazon CloudFront. [Online]. Available: http://aws.amazon.com/cloudfront/details/

[5] Google Global Cache. [Online]. Available: https://peering.google.com/about/ggc.html

[6] E. Nygren, R. K. 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.

[7] S. Scellato, C. Mascolo, M. Musolesi, and J. Crowcroft, "Track globally, deliver locally: Improving content delivery networks by tracking geographic social cascades," in *Proc WWW'11*, 2011, pp. 457–466.

[8] K. Andreev, B. M. Maggs, A. Meyerson, and R. K. Sitaraman, "Designing overlay multicast networks for streaming," in *Proc. SPAA'03*, 2003, pp. 149–158.

[9] M. Z. Shafiq, A. X. Liu, and A. R. Khakpour, "Revisiting caching in content delivery networks," in *Proc. SIGMETRICS'14*, 2014, pp. 567–568.

[10] R. K. Sitaraman, M. Kasbekar, W. Lichtenstein, and M. Jain, "Overlay networks: An akamai perspective," in *Advanced Content Delivery, Streaming, and Cloud Services*, 2014, ch. 16, pp. 305–328.

[11] S. Borst, V. Gupta, and A. Walid, "Distributed caching algorithms for content distribution networks," in *Proc. INFOCOM'10*, 2010, pp. 1–9.

[12] G. Gursun, M. Crovella, and I. Matta, "Describing and forecasting video access patterns," in *Proc. INFOCOM'11*, 2011, pp. 16–20.

[13] G. Szabo and B. A. Huberman, "Predicting the popularity of online content," *Communications of the ACM*, vol. 53, no. 8, pp. 80–88, 2010.

[14] D. Niu, Z. Liu, B. Li, and S. Zhao, "Demand forecast and performance prediction in peer-assisted on-demand streaming systems," in *Proc. INFOCOM'11*, 2011, pp. 421–425.

[15] Z. Wang, L. Sun, C. Wu, and S. Yang, "Guiding internet-scale video service deployment using microblog-based prediction," in *Proc. INFOCOM'12*, 2012, pp. 2901–2905.

[16] M. Rowe, "Forecasting audience increase on YouTube," in *Workshop on User Profile Data on the Social Semantic Web*, 2011.

[17] H. Li, X. Ma, F. Wang, J. Liu, and K. Xu, "On popularity prediction of videos shared in online social networks," in *Proc. CIKM'13*, 2013, pp. 169–178.

[18] S. Roy, T. Mei, W. Zeng, and S. Li, "Towards cross-domain learning for social video popularity prediction," *IEEE Transactions on Multimedia*, vol. 15, no. 6, pp. 1255–1267, 2013.

[19] J. Xu, M. van der Schaar, J. Liu, and H. Li, "Forecasting popularity of videos using social media," *IEEE Journal of Selected Topics in Signal Processing*, vol. 9, no. 2, pp. 330–343, 2015.

[20] Z. Wang, W. Zhu, X. Chen, L. Sun, J. Liu, M. Chen, P. Cui, and S. Yang, "Propagation-based social-aware multimedia content distribution," *ACM Transactions on Multimedia Computing, Communications, and Applications*, vol. 9, no. 1, pp. 52:1–52:20, 2013.

[21] Y. Wu, C. Wu, B. Li, L. Zhang, Z. Li, and F. Lau, "Scaling social media applications into geo-distributed clouds," *IEEE/ACM Transactions on Networking*, vol. 23, no. 3, pp. 689–702, 2015.

[22] J. Famaey, F. Iterbeke, T. Wauters, and F. De Turck, "Towards a predictive cache replacement strategy for multimedia content," *Journal of Network and Computer Applications*, vol. 36, no. 1, pp. 219–227, 2013.

[23] P. Blasco and D. Gunduz, "Learning-based optimization of cache content in a small cell base station," in *Proc. ICC'14*, 2014, pp. 1897–1903.

[24] A. Slivkins, "Contextual bandits with similarity information," *The Journal of Machine Learning Research*, vol. 15, no. 1, pp. 2533–2568, 2014.

[25] S. Traverso, M. Ahmed, M. Garetto, P. Giaccone, E. Leonardi, and S. Niccolini, "Temporal locality in today's content caching: Why it matters and how to model it," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 5, pp. 5–12, 2013.

[26] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)." Technical report, Telecom Paris-Tech, 2011.

[27] M. Ahmed, S. Traverso, P. Giaccone, E. Leonardi, and S. Niccolini, "Analyzing the performance of lru caches under non-stationary traffic patterns," *arXiv:1301.4909*, 2013.

[28] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. ISCA'10*, 2010, pp. 60–71.

[29] J. Dilley and M. Arlitt, "Improving proxy cache performance: analysis of three replacement policies," *IEEE Internet Computing*, vol. 3, no. 6, pp. 44–50, 1999.

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

[31] pypy. [Online]. Available: http://pypy.org