Trend-Aware Video Caching through Online Learning

Suoheng Li, *Student Member, IEEE,* Jie Xu, *Member, IEEE,*
Mihaela van der Schaar, *Fellow, IEEE,* and Weiping Li, *Fellow, IEEE*

***Abstract*—This paper presents Trend-Caching, a novel cache replacement method that optimizes cache performance according to the trends of video content. Trend-Caching explicitly learns the popularity trend of video content and uses it to determine which video 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 videos. We prove that the learning regret of Trend-Caching (i.e., the gap between the hit rate achieved by Trend-Caching and that by the optimal caching policy with hindsight) is sublinear in the number of video requests, thereby guaranteeing both fast convergence and asymptotically optimal cache hit rate. We further validate the effectiveness of Trend-Caching 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. Furthermore, Trend-Caching has low complexity.**

I. Introduction

The rapid growth of rich media-enabled applications has greatly changed the way that people use the Internet, bringing the demand for high quality multimedia content into an unprecedented level. 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, new content platforms [1], and emerging technologies such as virtual reality. As a consequence, the video content that needs to be delivered in real-time has grown significantly in terms of volume, size and diversity. To provide high Quality-of-Service 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 [2]. The merits of content caching are numerous. For instance, content caching avoids long-distance transmissions of content, thereby enabling fast content delivery to end-users; content caching offloads inter-network traffic to intra-network, thereby reducing transit service payments for Internet service providers. In fact, content caching is now considered as a basic network functionality in emerging network architectures such as Content-Centric Networking [3] and Information-Centric Networking [4].

Content caching is not a new technology - Akamai [5] 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 content caching systems in order to accommodate this vast traffic. Cloud providers now start to launch their own caching services [6] and many websites also build their own caching systems to accelerate content distribution [7]. To improve content caching efficiency, a significant amount of research effort has been devoted to optimizing the network architecture, e.g., path optimization [8], server placement [9], content duplication strategy [10], etc. However, less attention has been paid 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 [11]. These algorithms are easy to implement but may suffer major performance degradation since they ignore the evolving trends of video content, which may alter the future traffic pattern on the network, thereby resulting in a low cache hit rate. Thus, an efficient content caching scheme should be trend-aware, meaning that it should explicitly incorporate the trend, i.e. the future popularity of content, into the caching decision making.

Trend-aware caching is not only very difficult but also inefficient for traditional web caching because web pages have small and diverse sizes and hence, estimating the trend for each web page would significantly increase the complexity. However, trend-aware caching is perfect fit for video content distribution since multimedia content is large enough so that estimating for each content would bring much less overhead. Moreover, modern practice tends to split multimedia content into several equal-sized chunks, relieving the caching system from jointly considering the popularity and the content size when making caching decisions. However, designing trend-aware caching schemes still faces many challenges. Firstly, the trend of a content is not readily available at the caching decision time but rather needs to be forecasted. Secondly, the trend 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

S. Li and W. Li are with the School of Information Science and Technology, University of Science and Technology of China, Hefei, Anhui 230027, P. R. China. (email: lzzitc@mail.ustc.edu.cn, wpli@ustc.edu.cn)

S. Li, and M. van der Schaar are with the Department of Electrical Engineering, University of California, Los Angeles, USA. (email: mihaela@ee.ucla.edu)

J. Xu is with the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, USA. (email: jiexu@miami.edu)

This research is supported by CSC. Weiping Li and Suoheng Li acknowledge the partial support from Intel Collaborative Research Institute on 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.)

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 trend of video content to perform efficient caching and propose an online learning algorithm, Trend-Caching, 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 if the cache if full). 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 trend is revealed over time. The contributions of this paper are summarized below:

- We propose Trend-Caching, an online algorithm that learns the relationship between the future popularity of a content and its recent access pattern. Using the popularity forecasting result, Trend-Caching knows the trend of each content and makes proper cache replacement decisions to maximize the cache hit rate. The amortized time complexity of Trend-Caching is logarithmic in the number of received requests.
- We rigorously analyze the performance of Trend-Caching 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 caching decision, is sublinear in the number of content requests received by our system. This guarantees fast convergence and implies that Trend-Caching asymptotically achieves the optimal performance.
- We propose Collaborative Trend-Caching (CoTrend-Caching) for caching decisions making among multiple geographically distributed cache nodes. Each node learns not only the future popularity of content but also, during the process, the best neighbor cache nodes that it can consult when it receives novel content.
- We demonstrate the effectiveness of Trend-Caching through experiments using real-world traces from movie.douban.com, the largest Rotten Tomatoes-like website in China. Results show that our algorithms are able to achieve significant improvements 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 Trend-Caching algorithm is proposed in Section V. Section VI presents theoretical analysis of the algorithm. Simulation results are shown in Section VIII. Finally, Section IX concludes the paper.

II. Related Work

The common approaches for content caching that have already been adopted in the Internet nowadays are summarized in [12]. As mentioned in the introduction, a significant amount of research effort was devoted to optimizing the network architecture, including path selection [8], content duplication strategy [13] [10], server placement [9], etc. For instance, [8] systematically describes the design of the Akamai Network. Authors in [13] assume that content popularity is given and then propose light-weight algorithms that minimize bandwidth cost. In [10], an integer programming approach to designing a multicast overlay network is proposed. [9] utilizes the geographic information extracted from social cascades to optimize content placement. 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 [11], which are simple but do not explicitly consider the trend of content when making caching decisions.

Forecasting popularity of online content has been extensively studied in the literature [14] [15]. Various solutions are proposed based on time series models such as autoregressive integrated moving average [16], regression models [17] and classification models [18]. Propagation features of content derived from social media are recently utilized to assist popularity prediction, leading to an improved forecasting accuracy [19] [20] [21]. While these works suggest ways to forecast the popularity of multimedia content, few works consider how to integrate popularity forecasting into caching decision making. In [22], propagation information of content over social media is utilized to optimize content replication strategies. An optimization-based approach is proposed in [23] 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 changing popularity of content, several learning methods for content caching are proposed. In [24], 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 the future popularity of the content and determine whether to cache this content. There are two main drawbacks of this method. First, the model fitting procedure is carried out independently for each content and hence, similarity information between content cannot be utilized to facilitate and improve popularity forecasting. Second, this method incurs a high training complexity if the number of content is large since the training process needs to be run periodically for every content. In [25], 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.

However, this algorithm requires an explicit assumption on the distribution of content popularity, which greatly limits its practical value since this popularity distribution is not know *a priori* and may change over time. Moreover, this work also learns the popularity independently across content, ignoring the similarity between content, thereby resulting in a slow learning speed. In contrast, Trend-Caching does not make any assumption on the popularity distribution of content. Instead, Trend-Caching exploits the similarity between content and gradually learns the expected popularity of content given their current context information (i.e. features that characterize the situation under which the content is requested such as access patterns in the past day and the type of the content). In addition to these works, [26] proposes an online learning method that uses bandits to predict the relevance between multimedia content. [27] presents a mixed caching strategy that considers both the age and popularity of content. [28] studies the collaboration between distributed cache nodes and proposes optimization-based algorithms. [29] presents an epidemic model based cache replacement algorithm that utilizes social propagation information. Compared to our previous work [30], this paper provides more experimental results and an extension (CoTrend-Caching) to further improve cache performance.

Our proposed Trend-Caching algorithm exploits the similarity information between content in a similar way as an online k-Nearest Neighbors (kNN) algorithm and its variants [31]. However, there are several important distinctions. First, online kNN algorithms fail to synthesize the relationship between contexts and popularity of content, resulting in large time and space complexity (e.g. the time complexity is linear to K for each request where K is the number of received content requests). In contrast, Trend-Caching builds an explicit mapping from context space to popularity and refines this mapping over time as new requests are received. Thus, Trend-Caching incurs a much smaller time and space complexity (the amortized time complexity is $O(\log K)$). Second, online kNN algorithms exploit the similarity information in a heuristic way and hence are not able to provide any performance guarantee on the forecasting accuracy or speed of learning. In contrast, Trend-Caching utilizes the similarity information only when the confidence is sufficiently high. As a result, Trend-Caching provides a rigorous performance guarantee on the popularity forecasting performance as well the cache hit rate. In particular, we prove that the performance loss incurred by Trend-Caching due to uncertainty is sublinear in the number of received content requests, compared to the optimal oracle caching scheme that knows the future popularity of all content. This implies that Trend-Caching is able to achieve the optimal performance asymptotically.

Table I summarizes the difference of Trend-Caching from existing works on cache replacement with popularity estimation.

TABLE I
COMPARISON WITH EXISTING WORKS ON CACHE REPLACEMENT WITH POPULARITY ESTIMATION.

	Trend-Caching (This paper)	[25]	[24]	[22]	[23]
Model-free	Yes	No	No	No	No
Online/offline learning	Online	Online	Offline	Offline	Online
Performance guarantees	Yes	Yes	No	No	Yes
Tracks changing trends	Yes	No	Retrains	Yes	Yes
Time complexity	Log	Linear	Linear	Log	Polynomial

III. System Overview

A. Architecture

Fig. 1 illustrates the overall architecture of Trend-Caching. It shows that our Trend-Caching algorithm can be deployed in a variety of caching scenarios. Since the complexity of Trend-Caching is relatively low, it can even be deployed on devices with limited storage space and computation power, such as wireless access points which provide content caching capability support. The Trend-Caching can be easily configured such that it can be deployed in servers/access points having different computation/storage capacities. Since caching nodes make caching decisions in a distributed way, they can use different configurations and parameters.

Several cache servers are set up in different locations to serve local users with content from their local cache. When a requested content does not exist in its local cache, the cache server forwards the request to back-end storage servers or other cache servers. According to the decision of cache replacement algorithm, the cache server may keep a copy of the content in its local cache and drop old content to make room for the new one.

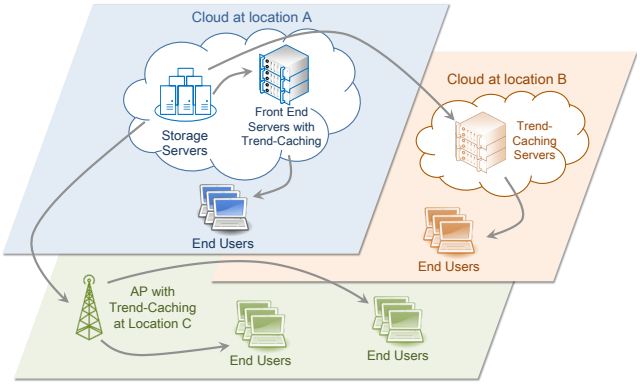


Fig. 1. Overall architecture of Trend-Caching. Since Trend-Caching has low complexity, it can even be deployed on devices with limited hardware resources, such as access points.

For economic and operational considerations, service providers may only have their storage servers deployed at a limited number of locations, or even in only one location. To provide services with high quality, they set up their own caching systems at multiple locations or use 3rd-party caching services to deliver their content to end users.

The modules of a single trend-aware cache node is depicted in Fig. 2. In addition to the basic modules (i.e. *Cache Management*, *Local Cache*, and *Request Processor*) in a conventional cache node, the trend-aware 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*. These features will be used to generate the context vector of a content request, which captures the situation under which the request is made. For instance, the context vector may include the number of times the content is requested in the last hour, the last day or the last week.
- The *Learning Interface* module implements the Trend-Caching algorithm. For each request, the *Learning Interface* is invoked twice. When a reqeuest is received, the *Learning Interface* decides whether to store the requested content into local cache based on the forecasted popularity of content. After the popularity of the content is revealed, the *Learning Interface* updates the *Learning Database* which maintains the mapping from the context space to the popularity space.

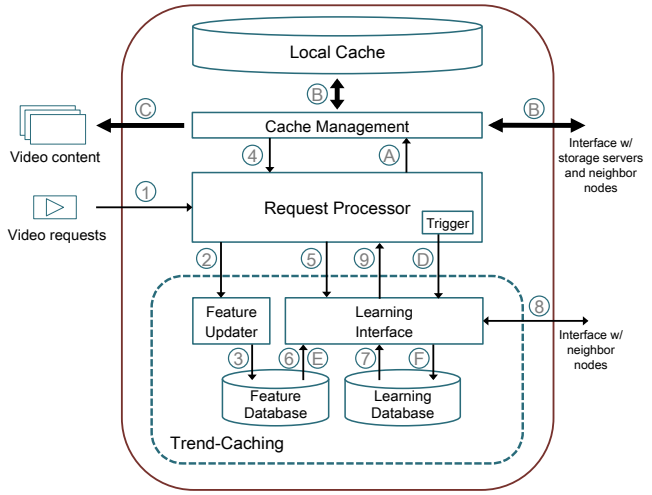


Fig. 2. Modules of a single cache node in a caching system. A typical work flow is also presented.

Remark: The learning algorithm forecasts the future popularity of a requested content based on the context information of the request. We limit the context vector to contain only recent activities but not the full access history of the requested content. In this way, outdated information is eliminated and thus the learning algorithm is able to track changes more quickly. Therefore, *Feature Database* is not necessarily a persistent database. In our implementation, a sliding window is used to filter out outdated data.

B. Operational Principles

Each content request involves three sequential procedures. First, when a request arrives to the cache node, Trend-Caching **updates** the *Feature Database* to keep up-to-date features of the requested content. Second, Trend-Caching 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, Trend-Caching **learns** the relationship between the context vector and the popularity of content and then encodes this knowledge into the *Learning Database*. Such knowledge will be used in future requests for content with similar context vectors. The detailed operations of the trend-aware cache node are described below. However, we note that our main focus is on the modules that enable the learning capability.

- **Update**:
  1) A request from an end user is received by the *Request Processor*.
  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 reports whether the requested content is in local cache.
  5) If the requested content is not found, the Trend-Caching algorithm decides whether or not to cache the content.
  6) The Trend-Caching algorithm extracts the context vector from the *Feature Database*.
  7) The Trend-Caching algorithm searches the *Learning Database* with the context vector to forecast the popularity of the requested content.
  8) The algorithm may also consult other cache nodes about their popularity estimations for the same request.
  9) The Trend-Caching algorithm makes a caching decision based on the forecasted popularity of the content.
  A) Based on the caching decision, the requested content is either cached locally or not (e.g. the node acts as a proxy, or redirects the user to another server).
  B) If the content is decided to be cached, *Cache Management* module obtains the content from upstream servers, push the content into the local cache, and removes staled items from the local cache.
  C) The request is served.
- **Learn**:
  D) The *Request Processor* triggers a learning process.
  E) The Trend-Caching algorithm extracts the context vector and revealed popularity from the *Feature Database*.
  F) Trend-Caching updates the *Learning Database* with the context vector and revealed popularity.

IV. System Model

Consider the setting where a content provider has a set of video 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. The caching system aims to offload requests to its local cache at its best effort. In this paper, we first focus on the single-node case where each cache node operates independently. The multiple-node case where

[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.

inter-node communication is allowed will be discussed later in Section VII. 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][3] 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. For example, we may use the number of times $c(k)$ being requested during the last hours and the number of times $c(k)$ being requested during the last day to form a 2-dimensional context vector for each request. 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 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.

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

A summary of notations is presented in Table II.

TABLE II
NOTATIONS USED IN PROBLEM FORMULATION AND TREND-CACHING ALGORITHM.

c, $c(k)$, $\mathcal{C}, C$	content, content requested in the k-th request, set of all content, total number of content
req_k, Req	k-th request, set of all requests
K	number of requests
s	capacity of the local cache
$t(k)$	time when the k-th request is initiated
x, $x(k)$, $\mathcal{X}$	context vector, context vector of the k-th request, context space ($x \in \mathcal{X}$)
$Y_k, Y_k(c)$	size C binary vector describing the cache status, c-th element in Y_k indicating whether c is cached
$H(K,\pi), H(\pi)$	cache hit rate of the first K requests under policy π, long-term cache hit rate under π
d	dimension of the context space ($\mathcal{X} = [0,1]^d$)
ϕ	time interval between popularity re-estimations
θ	time between receiving a request and learning from it
$\pi(Y\|c,x)$	caching strategy that describes how local cache is updated after serving a request with context x
π^*, π_0	the optimal policy, Trend-Caching
$M_k, \tilde{M}_k$	request rate of content $c(k)$, forecasted request rate of content $c(k)$
$M_i^{sort}, \tilde{M}_i^{sort}$	request rate of the i-th popular content, forecasted request rate of the i-th popular content
Q	priority queue that stores all cached content in increasing order of their popularity
c^{least}, $\tilde{M}^{least}$	top element in Q, forecasted request rate of c^{least}
P_i, $\mathcal{P}(k)$	hypercube, partition of the context space when processing the k-th request
l_i	level of hypercube P_i
$P^*(k)$	the hypercube that $x(k)$ belongs to
$\mathcal{M}(P_i), \mathcal{N}(P_i)$	variable that counts the sum of request rate in P_i variable that counts the number of requests in P_i,
$\zeta(l_i)$, z_1, z_2	threshold for hypercube splitting ($\zeta(l_i) = z_1 2^{z_2 \cdot l_i}$), algorithm parameter, algorithm parameter
$\mathcal{N}_{thresh}$	threshold in Collaborative Trend-Caching, for deciding whether to consult neighbor nodes

[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 and we use this assumption to simplify the theoretical analysis. We want to emphasize that our algorithm can also be used to cache unequal-sized chunks, e.g. using evicting strategy in [32].

[3]Chunks that belong to the same video may have strong mutual dependency in their access patterns (e.g. users will request these chunks in sequence). Our algorithm can fully utilize this dependency because the context vectors of these chunks will have similar values, which will lead to similar forecasted popularity.

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
  Run Learn($x(k)$, M_k)
end procedure

Fig. 3. Procedure of processing a single request. **Learn** and **Estimate** are two procedures defined in Section V-B.

V. TREND-CACHING ALGORITHM

A. Algorithm Overview

The Trend-Caching algorithm is presented in Fig. 3. 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, Trend-Caching 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, Trend-Caching 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, Trend-Caching 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, Trend-Caching 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 Trend-Caching replaces the least popular content c^{least} with $c(k)$ in the local cache and updates Q accordingly; otherwise, Trend-Caching does nothing to the local cache. To keep the popularity estimates of content in the local cache up to date, Trend-Caching periodically updates the forecast for the cached content after every ϕ requests.

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. 4. 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. 5. The procedure of adaptive context space partitioning.

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, Trend-Caching 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 $P^*(k)$ are 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 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

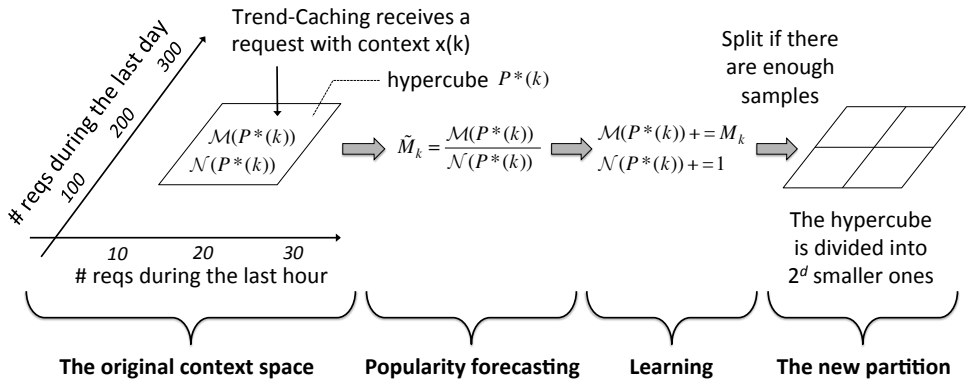


Fig. 6. An illustration of popularity forecasting and adaptive context space partitioning in which, for illustration purposes, we suppose that the context space is two-dimensional ($d = 2$).

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. On one hand, if we partition the context space very slowly, then each hypercube will cover a large area in the context space (e.g. in the extreme case, the context space never splits). The popularity estimate of a hypercube can be very inaccurate since the requests in this hypercube can be very different. On the other hand, if we partition the context space very quickly, then each hypercube covers a very small area in the context space (e.g. in the extreme case, each context point is a hypercube and hence the context space has a infinite number of hypercubes). The popularity estimate of a hypercube can be very inaccurate as well since the number of requests in the hypercube is small. Therefore, $\zeta(l_i)$ must be carefully designed in order to obtain an accurate popularity forecast. Inspired by [21] and [33], we design $\zeta(l_i)$ 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, Trend-Caching can achieve the optimal performance asymptotically. The detailed algorithm of adaptive context space partitioning is shown in Fig. 5.

As a simple illustration, Fig. 6 shows how Trend-Caching 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 Trend-Caching 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 [34] [35] 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.*

We now bound the forecast error made by the Trend-Caching algorithm. The proof can be found in Appendix B.

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 $K \to \infty$, $\mathbb{E}\sum_{k=1}^{K} |\tilde{M}_k - M_k|/K \to 0$. In other words, Trend-Caching 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 Trend-Caching 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 Trend-Caching and the performance loss of Trend-Caching 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

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

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

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 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. The proof again can be found in Appendix C.

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 Trend-Caching achieves the optimal performance asymptotically.

Theorem 1. *Trend-Caching 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 $\phi \gg s$:

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

□

C. Complexity of Trend-Caching

Both time complexity and space complexity are very important design considerations of Trend-Caching. In this section, we provide the complexity analysis of Trend-Caching and discuss practical issues.

[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.

We analyze the time and space complexity of Trend-Caching in this subsection. For the k-th request, the amortized time complexity for updating *Features Database*, querying *Learning Database*, updating *Learning Database*, and operating the priority queue are $O(1)$, $O(\log k)$, $O(\log k)$, and $O(\log s)$, respectively. The re-estimation procedure adds a $O(s\log k)$ time complexity every ϕ requests. Hence the amortized time complexity for each request is $O((2+s/\phi)\log K + \log s + 1) \approx O(\log K)$. This complexity can be reduced to $O(\log s)$ if we stop splitting hypercubes at a certain level. In this case we trade off time complexity with performance: the long-term cache hit rate will not converge to the optimum; instead it will stay within a small range near the optimum.

The space complexity of Trend-Caching is slightly more difficult to analyze. In general, there are two procedures that require large memory space, one is to store the learned knowledge to *Learning Database*, and the other is to keep the access history of active videos to *Feature Database*. For *Learning Database*, the worst-case space complexity is $O(K^{\frac{d}{d+z_2}})$, which is derived for the case where context vectors have uniformly distributed values. The best-case space complexity is $O(\log K)$, which is derived for the case where all requests have similar context vectors. Since the value in a context vector already represents the density of the distribution, in a real system the context vector will always follow a nonuniform distribution, and the complexity should be closer to $O(\log K)$. For *Feature Database*, its complexity is linear in the number of active videos that are recently requested. Although this complexity depends on access patterns of all videos and is hard to analyze, it is clear that it does not increase as more requests are observed. As a result, the overall space complexity is approximately $O(\log K)$ in the long-term.

In practice, if space and search complexity are a concern, one can easily implement an extension of Trend-Caching in which the context space is partitioned up to a certain level (i.e. once a hypercube reaches a predefined smallest size, the splitting stops). In this way, the storing and searching complexity is bounded. This extension may reduce the forecasting performance since learning is stopped at a certain point but the performance loss will be very small if the terminal hypercube is small enough.

VII. Multiple Cache Nodes

A real world caching system usually consists of multiple geographically distributed nodes. In such a setting, we may run a Trend-Caching instance on each node separately, which should provide good performance since the analysis in Section VI is still effective for each node. However, the similarity of access patterns among different nodes is not exploited if each cache node operates separately, which could lose potential performance. In this subsection, we extend Trend-Caching and propose Collaborative Trend-Caching (CoTrend-Caching) to utilize this similarity by allowing limited communications between nodes, i.e. step 8 in Fig. 2.

The idea is to let each cache node not only searches its own *Learning Database* but also consults neighbor nodes when the node is not confidence with its local estimation,

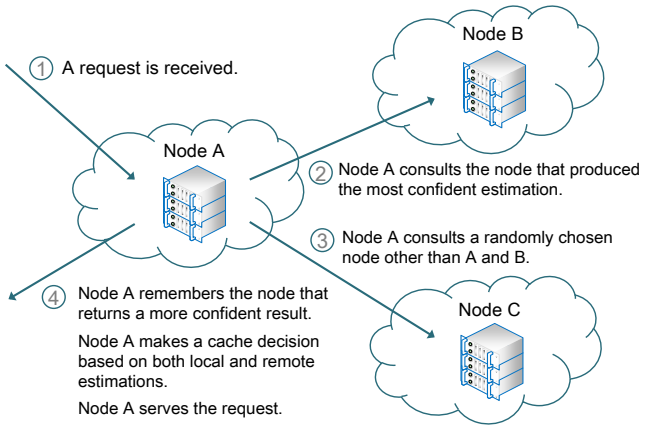


Fig. 7. Basic idea of CoTrend-Caching.

procedure ESTIMATEEX($x(k)$) ▷ Extended estimation
  Determines $P^*(k)$ that $x(k)$ belongs to
  if $\mathcal{N}(P^*(k)) \leq \mathcal{N}_{thresh}$ **then**
    $NB_1 \leftarrow NB(P^*(k))$
    $NB_2 \leftarrow$ a random cache node except NB_1
    Sends $x(k)$ to NB_1 and NB_2
    Obtains $\mathcal{M}_1$, $\mathcal{N}_1$ from NB_1
    Obtains $\mathcal{M}_2$, $\mathcal{N}_2$ from NB_2
    if $\mathcal{N}_2 > \mathcal{N}_1$ **then**
      $(\mathcal{N}_1, \mathcal{M}_1, NB_1) \leftarrow (\mathcal{M}_2, \mathcal{N}_2, NB_2)$
    end if
    $NB(P^*(k)) \leftarrow NB_1$
    return $\dfrac{\mathcal{M}_1 + \mathcal{M}(P^*(k))}{\mathcal{N}_1 + \mathcal{N}(P^*(k))}$
  else
    return $\mathcal{M}(P^*(k))/\mathcal{N}(P^*(k))$
  end if
end procedure

Fig. 8. Procedure of the extended **Estimate** for multi-node case.

e.g. very little similar requests have been observed. In this way, a better popularity forecast can be made. The consulted node will provide the consulting node with popularity estimate based on its own experience but not update its own *Learning Database*. The reason is that the consulted node does not have the realized popularity of this requested content to update its *Learning Database*. The final decision is then made according to both estimation sources.

To help a node find good neighbors, each time two neighbor nodes are consulted, and the one with higher confidence is remembered. The next time the node will consult the remembered neighbor node and a randomly picked node. This procedure is illustrated in Fig. 7. More specifically, we add a variable NB to each hypercube to track the last chosen neighbor and initialize it to any possible neighbor node. Let $NB(P^*(k))$ denote the variable NB in the hypercube that $x(k)$ belongs to. When a new request is received, the cache node extracts the context vector of the request and finds the hypercube $P^*(k)$ in the same way as normal Trend-Caching. The algorithm then selects two different neighbor nodes to consult. The first node NB_1 is $NB(P^*(k))$ and the second node NB_2 is randomly chosen from all available nodes except $NB(P^*(k))$. For each of these two nodes, the algorithm sends the context vector and requests the value of the two counters $\mathcal{M}$ and $\mathcal{N}$. Upon receiving the consult request, a neighbor node finds the hypercube according to the context vector contained in the consult request and returns the values of the counters $\mathcal{M}$ and $\mathcal{N}$ in the hypercube. This step is similar to the normal estimate procedure that we previously described in Fig. 4, but the difference is that this time a node directly returns the raw values of the two counters instead of the estimated popularity. The reason is that we need $\mathcal{N}$ as an indication of the confidence of the popularity estimation: a larger $\mathcal{N}$ means more observed samples and thereby implies higher confidence. After obtaining the values of the two counters from both neighbors, the original node stores them in $\mathcal{M}_1$, $\mathcal{N}_1$, $\mathcal{M}_2$, and $\mathcal{N}_2$ respectively. The algorithm then selects the more confident node and swap all obtained values between the two neighbors if $\mathcal{N}_2 > \mathcal{N}_1$, so that NB_1 always represents the node that is more confident about its estimation. After remembering NB_1 in the hypercube, the algorithm estimates the future popularity of the requested content by using information from both local and NB_1. Note that, the intention of CoTrend-Caching is to help increase learning speed by using information from neighbor nodes. However, while using this external information, a node's local estimates may also be damaged if the requested video has significantly different popularity distributions between the neighbor node and local. To avoid this issue and prevent an already confident local result being polluted during collaboration, a node may choose to use the external information only when its confidence of local estimate is low, e.g. $\mathcal{N}(P^*(k)) \leq \mathcal{N}_{thresh}$. Fig. 8 presents the detailed popularity estimation process in CoTrend-Caching.

VIII. Experimental Results

A. Dataset

We use data crawled from movie.douban.com as our main dataset for the evaluation of Trend-Caching and CoTrend-Caching. 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. 2 and evaluate the performance of Trend-Caching. 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 Trend-Caching with benchmarks listed below:

- **Modified Combinatorial Upper Confidence Bounds (MCUCB) [25].** The algorithm learns the popularity of cached content through their view counts and updates the local cache after serving every U requests. All content files are ranked according to the estimated upper bound of their popularity, among which top s content are cached. There are two parameters in this algorithm, U and F^γ. In our simulations, we set $U = 100$, which is the same value used in [25], and $F^\gamma = 150$. We have tuned F^γ by trying a wide range of values under $s = 250$ and picked the best one that achieves the highest cache hit rate.
- **First In First Out (FIFO) [36].** 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) [37].** 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) [38].** 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 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) [32].** 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 that is initialized to 0 and updated whenever a content is evicted. The cached content are ranked according to the values of their keys. When a content is requested, its key is updated to the sum of the number of access and the value of the cache age counter. When a content is evicted, the cache age counter is updated to the value of the content's key. The content with the lowest rank is replaced by the new content when the cache is full.
- **Optimal Caching.** The cache node runs Belady's MIN algorithm [39] 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.

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

D. Performance Comparison

Fig. 9 shows the overall average cache hit rates achieved by Trend-Caching and the benchmark algorithms under various cache capacity. As can be seen, Trend-Caching 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, Trend-Caching learns continuously and stores the learned knowledge into the learning database which can be utilized in the future. The learning is also refined over time when more content requests are received. By considering the future popularity of content, Trend-Caching makes smarter

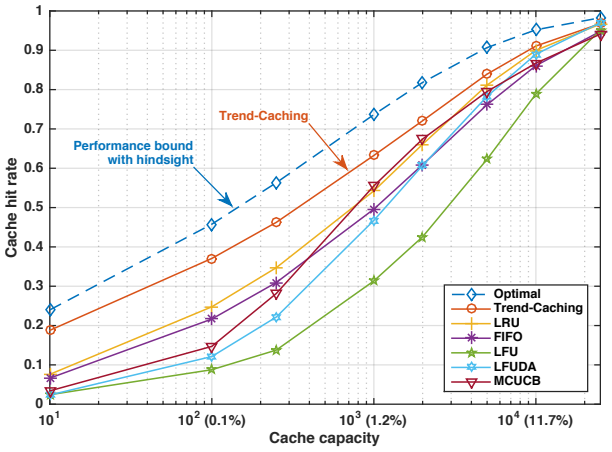


Fig. 9. Cache hit rate under different cache capacity. The percentage number in brackets along the x-axis is the ratio of the cache capacity to the total number of content, i.e., s/C.

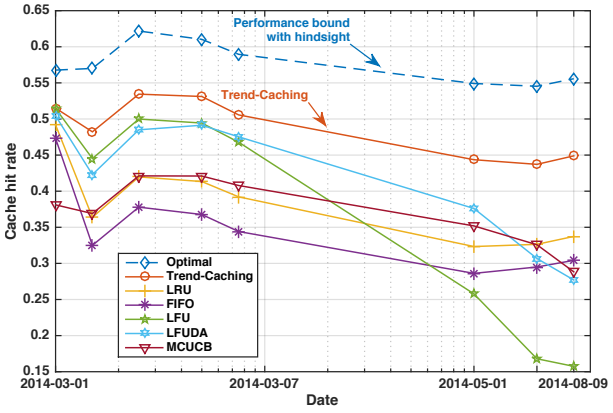


Fig. 10. Cache hit rate over time for the first 3 million requests starting from 1 March 2014. Cache capacity is 250 ($s = 250$). Each point in the figure represents the percentage of cache hit within the window between itself and its proceeding point.

decisions on which content should be cached and which content should be replaced. The advantage of Trend-Caching becomes greater when the cache capacity is smaller since more careful caching decisions need to be made. To illustrate the enormous improvement by adopting Trend-Caching on reducing the cache storage requirement, consider a common target cache hit rate of 0.5. In this case, Trend-Caching requires a cache capacity of 300 while LFU needs a cache capacity of 3000.

In Fig. 10, 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 caching decisions. Then, the advantage of Trend-Caching becomes obvious as more requests arrive. In this time, Trend-Caching successfully learns from the large volume of requests and hence makes accurate popularity predictions. (2) The cache hit rate achieved by Trend-Caching 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, Fig. 10 is still useful to illustrate the learning behavior of Trend-Caching 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, Trend-Caching responses quickly to the changes in popularity distribution, and therefore maintains a steady cache hit rate.

TABLE III
CACHE HIT RATE OF TREND-CACHING 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 IV
NUMBER OF CACHE REPLACEMENT UNDER DIFFERENT CACHE CAPACITY. LOWER VALUE MEANS THAT LESS NETWORK TRAFFIC IS USED TO PULL VIDEO CONTENT FROM UPSTREAM SERVERS.

	$s = 10$	$s = 250$	$s = 10000$
Trend-Caching	6.4 $\times 10^3$	118 $\times 10^3$	237 $\times 10^3$
LRU	7331 $\times 10^3$	5179 $\times 10^3$	795 $\times 10^3$
FIFO	7407 $\times 10^3$	5478 $\times 10^3$	1103 $\times 10^3$
LFU	7742 $\times 10^3$	6841 $\times 10^3$	1676 $\times 10^3$
LFUDA	7747 $\times 10^3$	6175 $\times 10^3$	873 $\times 10^3$
MCUCB	193 $\times 10^3$	1495 $\times 10^3$	220463 $\times 10^3$
Optimal	855 $\times 10^3$	1120 $\times 10^3$	208 $\times 10^3$

Table III 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 IV lists the number of cache replacement that each algorithm has performed during the simulation. The number of cache replacement is proportional to the amount of traffic the cache node has consumed to fetch video content from upstream servers. Although we do not explicitly consider it in our objective function, this metric is still importance because it provides a clear estimation of operational cost that an algorithm could incur. From the results we see that Trend-Caching needs very few cache replacement compared to other algorithms. Trend-Caching even outperforms the optimal caching strategy when the cache capacity is small. This is not surprising since the optimal strategy seeks all opportunities to increase its cache hit rate, which may lead to frequent cache updates when the cache capacity is small. We also notice that MCUCB makes a large amount of cache replacement when the cache capacity is large. This is due to an intrinsic drawback of the algorithm: when the cache capacity is large enough that some pieces of cached content get rarely accessed, the algorithm will replace them with similarly popular ones. However, if those new pieces of content still receive few access in the next period, the algorithm will switch again to other similarly popular ones, possibly the old content that was replaced in the previous period, thereby generating a lot of redundant traffic.

To investigate the performance of Collaborative Trend-Caching in the multi-node case, we set up a six-node environment and split all requests based on their originated locations. All nodes run different instances of the same algorithm at the same time. Requests are dispatched towards the node that they are geographically close to. Results under this setting are shown in Fig. 11. Clearly, CoTrend-Caching outperforms

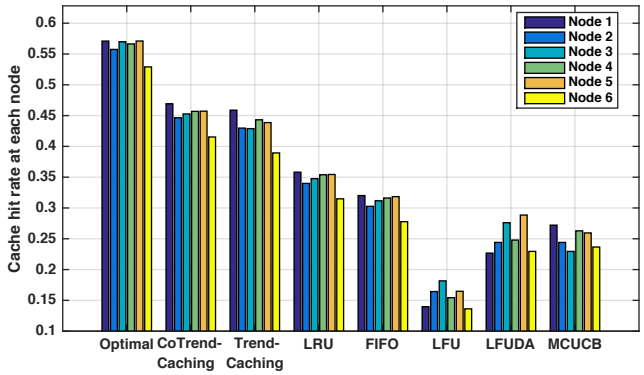


Fig. 11. Cache hit rate for multi-node case. The six bars in each cluster represent the cache hit rates on these nodes. The cache capacity of each node is set to 250.

TABLE V
PERFORMANCE COMPARISON BETWEEN TREND-CACHING AND COTREND-CACHING. WE LIST THE PROPORTION OF REQUESTS, CACHE HIT RATES, AND HIT RATE IMPROVEMENT OF EACH NODE. THE IMPROVEMENT IS CALCULATED AS THE REDUCTION OF HIT RATE GAP BETWEEN TREND-CACHING AND THE OPTIMAL STRATEGY.

Node	1	2	3	4	5	6
Distribution	0.34	0.13	**0.08**	0.24	0.11	**0.09**
Optimal Strategy	57.10	55.74	56.98	56.64	57.11	52.91
Trend-Caching	45.88	42.97	42.89	44.31	43.87	38.94
CoTrend-Caching	46.91	44.64	45.27	45.69	45.72	41.53
Improvement (%)	9.18	13.12	**16.90**	11.15	13.96	**18.54**

all other algorithms. To further compare CoTrend-Caching with Trend-Caching, we list the detailed simulation results in Table. V. The first row in the table shows the distribution of requests, i.e. the percentage of number of requests that are received by each cache node. The following rows represent the overall cache hit rate for the optimal strategy, Trend-Caching, and CoTrend-Caching respectively. The last row shows the improvement of CoTrend-Caching over Trend-Caching. We calculate the improvement as the reduction of cache hit rate gap between Trend-Caching and the optimal strategy. E.g., on node 1, CoTrend-Caching improves the cache hit rate by $\frac{(57.10-46.91)-(57.10-45.88)}{57.10-45.79} * 100\% = 9.18\%$. From Table. V we can see that nodes with fewer requests benefit more from collaboration. For instance, node 3 and node 6 (bold numbers in the table) receive the least number of requests, and have the highest cache hit rate improvement. This phenomenon is expected since the few requests a node receives, the more information it can gain from collaboration.

Finally, we compare the running speed of Trend-Caching with the benchmarks in Table VI. In our implementations, all algorithms are written in pure Python [40] 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 VI, Trend-Caching processes more than 20 thousand requests per second and outperforms both LFU and LFUDA. This means that Trend-Caching 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.

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

	$s = 100$	$s = 10000$
Trend-Caching	28.9 $\times 10^3$	25.8 $\times 10^3$
LRU	127.9 $\times 10^4$	72.5 $\times 10^4$
FIFO	146.3 $\times 10^4$	86.4 $\times 10^4$
LFU	24.5 $\times 10^3$	9.4 $\times 10^3$
LFUDA	17.3 $\times 10^3$	5.3 $\times 10^3$
MCUCB	1.6 $\times 10^3$	0.7 $\times 10^3$

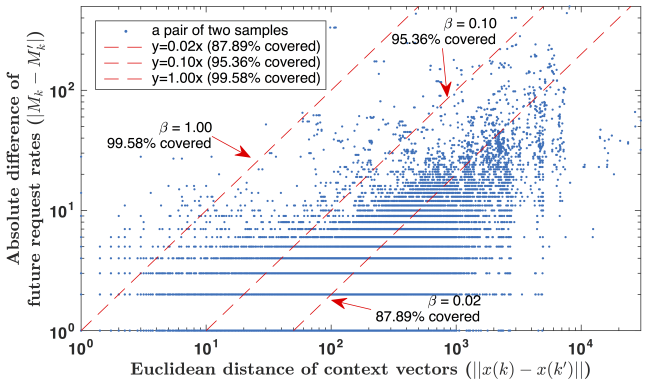


Fig. 12. Verification of Assumption 1 by using the real data trace from movie.douban.com. Each blue point represents a pair of two samples (i.e. k and k'). Each red dashed line in the figure represents a different value of β, and we expect all blue points to be below the line.

E. Verification of Assumption 1

In Assumption 1, we try to bound the maximum expected difference of two videos' future request rates ($\mathbb{E}|M_k - M_{k'}|$) based on the Euclidean distance of these two requests' context vectors ($||x(k) - x(k')||$). To verify if this assumption holds in practice, we illustrate the relationship between request rates and context vectors using our dataset. Specifically, we randomly select 1000 time points in our dataset, and for each time point, we randomly select 1000 unique videos and calculate the context vector and future request rate of each video at each selected time point. When calculating the context vector and the future request rate, we use the same method and parameters used in previous experiments. The above data processing results in a total of $1000 * 1000 = 10^6$ samples, each sample consisting of a context vector and a realized future request rate. Next, we construct 10^6 pairs of samples by randomly picking samples from the sample pool. For each pair of samples, we calculate the absolute difference of request rates and the Euclidean distance of context vectors. The results are visualized in Fig. 12 with x-axis representing the Euclidean distance of context vectors ($||x(k) - x(k')||$) and y-axis representing the absolute difference of future request rates ($|M_k - M_{k'}|$). For a better representation, we use logarithmic scale for both x-axis and y-axis. In the figure, each blue dot represents a pair of samples. Since we use the logarithmic scale in the figure, Assumption 1 is equivalent to the following statement: there exists a straight line with a 45-degree angle to the x-axis such that all blue dots are below the line (i.e.

$\log|M_k - M_{k'}| \leq \log||x(k) - x(k')|| + \log\beta$). To verify our assumption, we draw several red dashed lines in the figure, each representing a different β. We can see that 99.5% pairs can be covered with $\beta = 1$, and even with a relatively small β (e.g. 0.1), more than 95% pairs are covered. This result provides strong evidence to the validity of Assumption 1.

IX. Conclusion

This paper proposed a novel online learning approach to performing efficient and fast cache replacement. Our algorithm (Trend-Caching) forecasts the trend (i.e., future popularity) of video content and makes cache replacement decisions based on forecasted trend. Trend-Caching 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 Trend-Caching, 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. We also provide an extension to the original proposed algorithm, Collaborative Trend-Caching, that can exploit the trend similarity among multiple locations. Extensive simulations with real world traces validate the effectiveness of our algorithms, as well as the insensitivity to parameters and fast running speed.

One future direction is to consider a more complex setting where multiple parties are involved. In such setting, cache nodes can be managed by different parties and may have different configurations and parameters. E.g., a node may cache only predefined content categories in order to comply with hardware limitations and caching policies. The challenge in such scenario is how to design cooperative algorithms that have low communication complexity and align the interests of all parties.

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 $P = P(x(k))$ 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

$$
\begin{aligned}
&\mathbb{E}|\tilde{M}_k - M_k| \\
=&\mathbb{E}\left|\mathcal{M}_P/\mathcal{N}_P - M_k\right| \\
=&\mathbb{E}\left|\frac{\sum_{k'} M_{k'}}{\mathcal{N}_P} - M_k\right| \\
\leq&\frac{\sum_{k'}\beta||x(k') - x(k)||}{\mathcal{N}_P} \\
=&\beta\sqrt{d}\frac{\begin{array}{r}\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l-1}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)2^{-i/2} \\ + ... + (\mathcal{N}_P - \lceil z_1 2^{l\cdot z_2}\rceil)2^{-l/2}\end{array}}{\mathcal{N}_P} \\
\leq&\beta\sqrt{d}\frac{\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l-1}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)2^{-i/2}}{\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l-1}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)} \\
\leq&\beta\sqrt{d}\frac{\lceil z_1 2^{z_2}\rceil + \sum_{i=1}^{l-1}(\lceil z_1 2^{(i+1)z_2}\rceil - \lceil z_1 2^{i\cdot z_2}\rceil)2^{-i/2}}{z_1 2^{l\cdot z_2}} \\
<&\beta\sqrt{d}\frac{z_1 2^{z_2} + \sum_{i=1}^{l-1}(z_1 2^{(i+1)z_2} - z_1 2^{i\cdot z_2})2^{-\frac{i}{2}} + \sum_{i=0}^{l-1} 2^{-\frac{i}{2}}}{z_1 2^{l\cdot z_2}} \\
<&\beta\sqrt{d}\frac{z_1 2^{z_2} + \sum_{i=1}^{l-1}(z_1 2^{(i+1)z_2} - z_1 2^{i\cdot z_2})2^{-i/2} + 4}{z_1 2^{l\cdot z_2}} \\
=&\beta\sqrt{d}\frac{2^{z_2}\cdot\frac{2-\sqrt{2}}{2} + (1-2^{z_2})2^{(z_2-\frac{1}{2})l} + 4(1-2^{z_2-\frac{1}{2}})/z_1}{2^{lz_2}(1-2^{z_2-\frac{1}{2}})} \\
=&\beta\sqrt{d}\cdot(C_1\cdot 2^{-z_2 l} + C_2\cdot 2^{-\frac{1}{2}l}) \qquad (6)
\end{aligned}
$$

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

$$
\begin{cases}
C_1 = \dfrac{2^{z_2}\cdot\frac{2-\sqrt{2}}{2}}{1-2^{z_2-\frac{1}{2}}} + \dfrac{4}{z_1} \\
C_2 = \dfrac{1-2^{z_2}}{1-2^{z_2-\frac{1}{2}}}
\end{cases}
$$

□

Appendix 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.

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

$$
\begin{aligned}
&<\beta\sqrt{d}\{z_1 2^{z_2}+1+\sum_{i=1}^{l} \\
&\quad [2^{id}(z_1 2^{(i+1)z_2}-z_1 2^{i\cdot z_2}+1)(C_1 2^{-z_2 i}+C_2 2^{-\frac{i}{2}})]\} \\
&<\beta\sqrt{d}\{z_1 2^{z_2}+1+C_1\frac{2^{d-z_2}}{1-2^{d-z_2}}+C_2\frac{2^{d-\frac{1}{2}}}{1-2^{d-\frac{1}{2}}}+ \\
&z_1(2^{z_2}-1)[2^d C_1\frac{2^{dl}-1}{2^d-1}+2^{d+z_2-\frac{1}{2}}C_2\frac{2^{(d+z_2-\frac{1}{2})l}-1}{2^{d+z_2-\frac{1}{2}}-1}]\} \\
&=\beta\sqrt{d}(C_3+C_4 2^{dl}+C_5 2^{(d+z_2-\frac{1}{2})l}) \\
&\leq\beta\sqrt{d}(C_3+|C_4|2^{dl}+|C_5|2^{(d+z_2-\frac{1}{2})l}) \qquad (7)
\end{aligned}
$$

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

$$
\begin{aligned}
K \geq&\lceil z_1 2^{z_2}\rceil+\sum_{i=1}^{l-1}[2^{id}(\lceil z_1 2^{(i+1)z_2}\rceil-\lceil z_1 2^{i\cdot z_2}\rceil)] \\
>&z_1 2^{z_2}+\sum_{i=1}^{l-1}[2^{id}(z_1 2^{(i+1)z_2}-z_1 2^{i\cdot z_2}-1)] \\
>&[z_1(2^{z_2}-1)-1]2^{(d+z_2)l} \qquad (8)
\end{aligned}
$$

thereby

$$
2^l<[z_1(2^{z_2}-1)-1]^{-\frac{1}{d+z_2}}\cdot K^{\frac{1}{d+z_2}} \qquad (9)
$$

By combining (7) and (9) we have:

$$
\begin{aligned}
&\mathbb{E}\sum_{k=1}^{K}(\tilde{M}_k-M_k) \\
&<\beta\sqrt{d}\left(C_3+|C_6|K^{\frac{d}{d+z_2}}+|C_7|K^{\frac{d+z_2-1/2}{d+z_2}}\right) \qquad (10)
\end{aligned}
$$

where

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

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] □

Appendix 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:

$$
\tilde{M}_{i_1}^{sort}\geq\tilde{M}_{i_2}^{sort}\geq...\geq\tilde{M}_{i_s}^{sort}\geq...\geq\tilde{M}_{i_C}^{sort} \qquad (11)
$$

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

$$
\sum_{j=1}^{s}\tilde{M}_{i_j}^{sort}\geq\sum_{i=1}^{s}\tilde{M}_i^{sort} \qquad (12)
$$

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 (12), we have

$$
\sum_{j=1}^{s}\tilde{M}_{i_j}^{sort}\geq\sum_{i=1}^{s}\tilde{M}_i^{sort}\geq(1-f(s))\sum_{i=1}^{C}M_i^{sort}-s\cdot\Delta M \qquad (13)
$$

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

$$
\begin{aligned}
&\frac{1}{\phi}\sum_{k=m\phi+1}^{m\phi+\phi}Y_k(c(k)) \\
=&\frac{\sum_{j=1}^{s}\lfloor M_{i_j}^{sort}\Delta t-1\rfloor}{\sum_{j=1}^{C}M_j^{sort}\Delta t} \\
\geq&\frac{\sum_{j=1}^{s}(M_{i_j}^{sort}-2/\Delta t)}{\sum_{j=1}^{C}M_j^{sort}} \\
\geq&\frac{\sum_{j=1}^{s}(\tilde{M}_{i_j}^{sort}-\Delta M-2/\Delta t)}{\sum_{j=1}^{C}M_j^{sort}} \\
\geq&\frac{(1-f(s))\sum_{j=1}^{C}M_j^{sort}-2s(\Delta M+1/\Delta t)}{\sum_{j=1}^{C}M_j^{sort}} \\
=&(1-f(s))-\frac{2s\cdot\Delta M}{\sum_{j=1}^{C}M_j^{sort}}-\frac{2s}{\phi} \qquad (14)
\end{aligned}
$$

□

[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.

References

[1] S. Ren and M. van der Schaar, "Pricing and investment for online tv content platforms," *IEEE Transactions on Multimedia*, vol. 14, no. 6, pp. 1566–1578, Dec 2012.

[2] 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.

[3] 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.

[4] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, July 2012.

[5] 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.

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

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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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.

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

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] C. Tekin and M. van der Schaar, "Contextual online learning for multimedia content aggregation," *IEEE Transactions on Multimedia*, vol. 17, no. 4, pp. 549–561, Feb 2015.

[27] Y. Zhou, L. Chen, C. Yang, and D. M. Chiu, "Video popularity dynamics and its implication for replication," *IEEE Transactions on Multimedia*, vol. 17, no. 8, pp. 1273–1285, Aug 2015.

[28] J. Dai, Z. Hu, B. Li, J. Liu, and B. Li, "Collaborative hierarchical caching with dynamic request routing for massive content distribution," in *Proc. INFOCOM'12*, 2012, pp. 2444–2452.

[29] H. Li, Y. Le, F. Wang, J. Liu, and K. Xu, "Snacs: Social network-aware cloud assistance for online propagated video sharing," in *Proc. Cloud'15*, 2015, pp. 877–884.

[30] S. Li, J. Xu, M. van der Schaar, and W. Li, "Popularity-driven content caching," in *Proc. INFOCOM'16*, 2016.

[31] C. D. Manning, P. Raghavan, and H. Schütze, *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[32] 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.

[33] R. Kleinberg, A. Slivkins, and E. Upfal, "Multi-armed bandits in metric spaces," in *Proc. STOC'08*, 2008, pp. 681–690.

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

[35] 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.

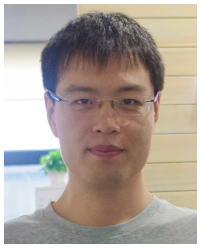
[36] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)." Technical report, Telecom Paris-Tech, 2011.

[37] 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.

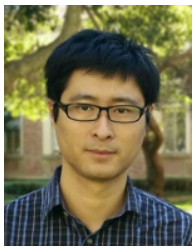
[38] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," in *Proc. ISCA'10*, 2010, pp. 60–71.

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

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

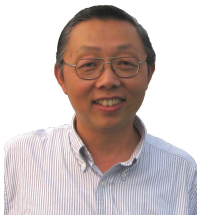


Suoheng Li received his B.S. and Ph.D. degrees in electronic engineering from University of Science and Technology of China (USTC), Hefei, China, in 2011 and 2016, respectively. From 2014 to 2015, he was a visiting scholar at UCLA. In 2016, he joined Yitu Technology, a Shanghai-based startup, as a researcher. His research interests include multimedia transmission, software-defined netowrking, and online learning.



Jie Xu received the B.S. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2008 and 2010, respectively, and Ph.D. in 2015 from the Department of Electric Engineering Department, University of California, Los Angeles (UCLA). He is currently an Assistant Professor at the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, USA. His primary research interests include game theory, online learning and networking.

Mihaela van der Schaar is Chancellor's Professor of Electrical Engineering at University of California, Los Angeles. She is an IEEE Fellow, was a Distinguished Lecturer of the Communications Society (2011-2012), the Editor in Chief of IEEE Transactions on Multimedia (2011-2013) and a member of the Editorial Board of the IEEE Journal on Selected Topics in Signal Processing (2011). Her research interests include engineering economics and game theory, multi-agent learning, online learning, decision theory, network science, multi-user networking, Big data and real-time stream mining, and multimedia.



Weiping Li (S'84-M'87-SM'97-F'00) received the B.S. degree in electrical engineering from University of Science and Technology of China (USTC), Hefei, China, in 1982, and the M.S. and Ph.D. degrees in electrical engineering from Stanford University, Stanford, CA, USA, in 1983 and 1988, respectively. In 1987, he joined Lehigh University, Bethlehem, PA, USA, as an Assistant Professor with the Department of Electrical Engineering and Computer Science. In 1993, he was promoted to Associate Professor with tenure. In 1998, he was promoted to Full Professor. From 1998 to 2010, he worked in several high-tech companies in the Silicon Valley (1998-2000, Optivision, Palo Alto; 2000-2002, Webcast Technologies, Mountain View; 2002-2008, Amity Systems, Milpitas, 2008-2010, Bada Networks, Santa Clara; all in California, USA). In March 2010, he returned to USTC and is currently a Professor with the School of Information Science and Technology.