Integrated Prefetching and Caching for Adaptive Video Streaming over HTTP: An Online Approach

Ke Liang  
Advanced Digital Sciences Center  
ke.liang@adsc.com.sg

Jia Hao  
National University of Singapore  
haojia@comp.nus.edu.sg

Roger Zimmermann  
National University of Singapore  
rogerz@comp.nus.edu.sg

David K.Y. Yau  
Singapore University of Technology and Design  
david_yau@sutd.edu.sg

ABSTRACT
We present an integrated prefetching and caching proxy, termed iPac, for HTTP-based adaptive video streaming services like Netflix and YouTube. The challenge we address is maximizing the byte-hit ratio for proxies through prefetching in the context of the limited bandwidth between proxies and content servers. The problem is NP-hard, and the best approximation ratio that any optimal offline algorithm can achieve is $1 - \frac{1}{e}\approx 0.63$. Considering that offline algorithms cannot be applied to real-time applications with stringent time constraints, we propose a novel $0.5$-competitive online prefetching algorithm which, to the best of our knowledge, has the best lower bound so far. We evaluate the performance of iPac by deploying it over the Amazon EC2 cloud accepting user requests from the video clients deployed on the PlanetLab based on a real trace of user requests for YouTube videos. Our experimental results demonstrate that iPac can significantly improve the performance in terms of byte-hit ratio (up to 84%) and video rates (up to 34%), compared with the state-of-the-art approaches. The proposed iPac is compatible with existing HTTP-based adaptive streaming implementations without requiring any modification to existing content servers and video clients.

Categories and Subject Descriptors
H.4 [Information Systems Applications]: Miscellaneous;  
C.4 [Performance of Systems]: Performance attributes;  
F.1.2 [Theory of Computation]: Online computation

Keywords
Adaptive streaming, prefetching, caching, online algorithm

1. INTRODUCTION
The dramatic growth of video traffic on the Internet reflects a rapid increase in the demand for video content as more people than ever before consume video online. Meanwhile, users’ expectations for video quality have greatly increased due to the widespread penetration of high-speed network accesses. HTTP-based adaptive streaming has been widely adopted by video streaming service providers (e.g., Netflix) for video delivery, because it can provide decent video rates for a large number of users with different network access and time-varying network conditions. This is attributed to the ability of using conventional web servers, caches and CDNs, and seamless traverse NATs and firewalls.

In an HTTP-based adaptive streaming application, each video file is encoded into multiple versions at different bitrates. Moreover, each version is split into smaller video segments, each of which contains the video data of a fixed time duration (e.g., 4 seconds in Netflix). Video players will send requests for video segments via HTTP, and they can switch between different bitrates in response to their changing network conditions. As the number of videos ramps up, the number of video segments can be huge. More importantly, all the video segments need to be delivered within stringent time limits. Otherwise, the user perceived quality will be significantly degraded [10].

As the last-mile connectivity is constantly improving as 4G/LTE and fiber-optic broadband access is becoming popular\(^1\), the connection between the content server and the proxy is increasingly the bottleneck for large-scale video streaming applications. This phenomena that the “middle” of the network has become a limiting factor has also been observed by large-scale content distribution networks such as Akamai Technologies [18]. To meet this challenge, a considerable number of caching and prefetching strategies have been proposed to reduce the access latency associated with the large-scale distribution of video content.

Caching [22], [6] has been widely recognized as an effective mechanism to reduce the access latency by keeping popular

\(^1\)ABI Research’s fiber broadband subscribers forecast market data: http://www.abiresearch.com/research/1002909
content at proxies that are closer to users. Since the size of video content is usually several orders of magnitude larger than web objects, a cache may quickly run out of storage. Consequently, the temporal locality of video content cannot be well exploited. This will result in a lower byte-hit ratio, which is defined as the amount of the requested data found in the cache divided by the total amount of the requested data. Prior work (e.g., [23, 24]) has studied caching strategies for video delivery by partially caching a small portion of videos or video segments. However, these approaches have little flexibility in adapting to the changing popularity of videos and user request patterns. Furthermore, caching-only strategies are not desirable for video streaming applications where video players request video data sequentially. To this end, researchers are resorting to prefetching strategies that can reduce access latency.

The objective of prefetching is to maximize the byte-hit ratio by predicting future requests. To do this, prefetching will fetch uncached data before users actually need it. As a result, the access latency is reduced when the user requests for the prefetched data arrive at proxies, resulting in better user perceived quality. However, prefetching aggressively is a waste of network resources, since the prefetched data may not be actually used as video players are free to switch between different bitrates in HTTP-based adaptive streaming applications. Even worse, it may deteriorate the user perceived quality by incurring additional queuing delay for the video data delivered from the content server, due to the limited bandwidth between the proxy and the content server (i.e., proxy-server bandwidth).

To overcome these limitations, the prefetching strategies need to be carefully designed. Although prefetching strategies have gained increasing attention recently for video content delivery (e.g., [7, 16]), little research exists that considers the limited proxy-server bandwidth and gives a theoretic lower bound of prefetching algorithms that guarantee the worst-case performance. Without the worst-case guarantee, the performance of prefetching algorithms may not only be far from optimal, but it may also be detrimental in practice.

In this paper, we study the problem of maximizing the byte-hit ratio for proxies through prefetching in the context of both the limited proxy-server bandwidth and the real-time constraints of video segments. We formulate it as a problem of online submodular maximization with knapsack constraints. The problem is NP-hard [15], and the best approximation ratio for any greedy offline algorithm is $1 - e^{-1} \approx 0.63$ [25].

**Technical Contribution:** The heart of iPac (i.e., integrated prefetching and caching) is a prefetching mechanism that adaptively prefetches uncached video segments respecting the limited proxy-server bandwidth without degrading the user perceived quality. Here we propose a near-optimal online prefetching algorithm with a competitive ratio of 0.5, which, to the best of our knowledge, is the best so far. As a highlight of this paper, we evaluate the performance of iPac via a real-world implementation based on an actual trace of user requests for YouTube videos. It is worth noting that the proposed iPac is compatible with existing HTTP-based adaptive streaming implementations, and it requires no modification to content servers and video players.

The rest of paper is organized as follows. We describe the architecture of the proposed iPac in Section 2. In Section 3, we formulate the problem of maximizing the byte-hit ratio with respect to limited proxy-server bandwidth, and propose an online prefetching algorithm. We present our cache replacement algorithm in Section 5, and analyse the performance of the proposed prefetching algorithm in Section 4. The implementation details of iPac and experiment results are provided in Section 6. Section 7 discusses related work. Lastly, we conclude the paper in Section 8.

2. OVERVIEW OF iPAC ARCHITECTURE

The proposed iPac is placed between users and the content server which stores all the videos. As shown in Fig. 1, iPac consists of three modules, i.e., a cache manager, a prefetch manager, and a request pool. Each module processes requests in an online fashion and all the inter-module communication is asynchronous and non-blocking. Consequently, there is negligible processing latency incurred in iPac. Furthermore, every module can be deployed on multiple machines or VMs working independently and simultaneously (i.e., there is no intra-module communication) when the number of user requests ramps up. Therefore, the iPac architecture can be easily scaled to serve a large number of users.

Before we describe the functionality of the three modules, it is worth noting that the cache to which iPac connects is indeed a cluster of cache servers that can be organized using any topology (e.g., a hierarchical topology [12]) in practice. In addition, we assume that suitable mechanisms for request routing (e.g., [17]) in the cache servers are available.

**Cache Manager:** The cache manager in iPac handles all the user requests and the video segments received from the content server. If the requested segment is contained in the cache (i.e., cache-hit), the cache manager will send back the segment immediately. Otherwise (i.e., cache-miss), it has to send the request to the request pool, which will forward the request to the content server fetching the segment. This is comparatively slower due to the non-negligible proxy-server latency and the limited proxy-server bandwidth. Meanwhile, the cache manager will generate prefetch requests for every user request, and send those to the prefetch manager. As shown in Fig. 8, there is a high probability ($\geq 0.83$) that a video player will send the next request with the same bitrate as the current request because the video player should avoid frequent bitrate switches that may annoy users [8], [13]. Therefore, the cache manager only gen-
generates prefetch requests for successive video segments with the same bitrates as the current request.

One of the main challenges of prefetching is to determine how many video segments should be prefetched. Prefetching too many unneeded video segments is not only a waste of bandwidth, but even worse, it may degrade the user perceived quality due to limited proxy-server bandwidth. In addition, video players can change the bitrates of requested video segments in response to their network conditions, thus the number of possible prefetch requests to be sent could be large. To this end, the cache manager only generates α prefetch requests for each user request, where α is the prefetching window size of the user request defined as the number of next video segments with the same bitrate as the current request. We discuss the impact of α on the performance of iPac in Section 6.

Upon arrival of a video segment from the content server, the cache manager will cache it if the cache is not full. Otherwise, it should determine whether the video segment should be cached or be discarded based on the utility of the corresponding request. The utility of a request, defined as the number of users that are about to send the request in the next αT seconds, where T denotes the time interval of the two successive requests sent by the video player in the steady state. The utility of a request reflects how likely the video segment will be requested in the near future, instead of how frequently it has been requested in the past.

To maintain the utility of each request such that it can be updated and queried quickly, we resort to three hash maps: (1) a client-request (C-R) map, which maintains information of the latest request for each client, (2) a request-utility (R-U) map, which maintains the utility of each request, and (3) a request-prefetch (R-P) map, which is a bidirectional map that enables us to efficiently (with O(1) time complexity) query the corresponding user requests given a prefetch request, and vice versa.

The three maps are updated as follows. When a user request arrives (e.g., (2, OI1B) in Fig. 2, where 2 denotes the user id and OI1B denotes the unique id of the request), the R-P map is updated by inserting a new pair of request ids (rid, rid), where the left rid and the right rid are the hash values of the user request (i.e., OI1B) and the prefetch request (i.e., D6RF), respectively. Since the prefetch request is within the prefetching window of the user request (see the R-P map in Fig. 2), its utility in the R-U map will be increased by one, as shown in the blue flow in Fig. 2. On the other hand, the C-R map will be updated by replacing the rid (i.e., M51N) of the user with the latest request id (i.e., OI1B). Similarly, the R-U map will be updated accordingly if the prefetch request (i.e., TA2P) is out of the prefetching window of the user request (i.e., OI1B), as shown in the red flow in Fig. 2.

Prefetch Manager: Due to the limited proxy-server bandwidth, the prefetch manager in iPac should not send all the prefetch requests to the request pool, and thus some of them will be discarded. Note that the cache manager will send all the cache-miss requests to the request pool bypassing the prefetch manager. The proxy-server bandwidth usage may increase rapidly as the number of the cache-miss requests increases. In this case, prefetching aggressively will degrade the QoS for the application by incurring additional queuing delay for the video segments delivered from the content server.

To efficiently utilize the scarce proxy-server bandwidth without deteriorating the QoS for the application, the prefetch manager needs to orchestrate the prefetch requests respecting the current proxy-server bandwidth usage. More specifically, the prefetch manager should quickly make a decision whether the received prefetch request should be sent to the request pool or be discarded, based on the current usage of the request pool. We will formulate the problem and propose an online prefetching algorithm in Section 3.

Request Pool: As shown in Fig. 1, the request pool is the sink of both the cache manager and the prefetch manager. It contains the copies of all the cache-miss requests and the prefetch requests that have been sent to the content server. Once a video segment is received from the content server, the corresponding request in the request pool will be removed immediately. In iPac, the request pool is designed as an asynchronous message pool, which sends and deletes requests asynchronously. The size of the request pool, defined as the total amount of video data of the corresponding requests in the request pool, reflects the current proxy-server bandwidth usage. When the size of the request pool is large, an additional queuing delay, denoted by Δt, will be incurred due to the limited proxy-server bandwidth. The incurred Δt will increase the access latency for cache-miss requests. As a response, video players may switch to lower bitrates since the measured throughput at the client side is lowered due to the increased access latency. In this case, we should not send prefetch requests which compete for the proxy-server bandwidth with cache-miss requests. To this end, we set a threshold, denoted by H, for the request pool to limit the impact of prefetch requests to the user perceived video rates. In other words, prefetch requests are not allowed to be sent when the size of the request pool exceeds H.

Suppose a video player, say v, can switch between different bitrates in \{b_1, b_2, ..., b_m\}, where \(b_1 < b_2 < ... < b_m\). The current download throughput of v can be approximated as \(\frac{\text{RWND}}{\text{RTT}}\) (RFC 6349 [1]), where RWND is the size of the TCP receiver window of v, and RTT is the round trip time between iPac and v. Let \(b_i\) be the current bitrate of v. Then we can

![Figure 2: Update of the C-R map, the R-U map, and the R-P map upon arrival of a user request in the cache manager of the iPac proxy. The red flow and the dashed blue flow is triggered by the user id, and the hash value of the user request, respectively.](image)
To provide QoS guarantees for the application, we set the prefetch requests which are supposed to retrieve video delivery. Note that the cache manager will generate cache-miss requests, which are sent to the request pool unconditionally as they arrive. More specifically, the prefetch manager needs an effective way to make decisions on the prefetch requests, which should be sent to the request pool immediately or be discarded, as determined by the prefetch manager. Let $Q$ be the set of prefetch requests that has been sent to the request pool right after the decision on $q_i$ is made. Then we can derive $S_i \subseteq S_{i+1} \subseteq S$, where $S \subseteq Q$ is the set of prefetch requests that will be sent to the request pool after all the prefetch requests in $Q$ have been processed. We define a set function $f : P(S) \rightarrow \mathbb{R}$, where $P(S)$ denotes the power set of $S$, as the number of bytes that will be hit at iPac if all the video segments of the prefetch requests in $S$ have been cached:

$$f(S) = \sum_{q_i \in S} u_i w_i.$$  

It can be observed that $f$ is non-negative (i.e., $f(S) \geq 0, \forall S_i \subseteq Q$), monotone (i.e., $f(S_i) \leq f(S_{i+1}), \forall S_i \subseteq Q$), and submodular (i.e., $f(S_i) + f(S_j) \leq f(S_i \cup S_j) + f(S_i \cap S_j), \forall S_i, S_j \subseteq Q$). The problem of maximizing the byte-hit ratio given $Q$ with respect to $H$ can be generalized to the problem of submodular maximization with knapsack constraints:

$$\max_{S \subseteq Q} \left\{ f(S) : \sum_{i \in S} w_i \leq H \right\}$$  

The purpose of the above formulation is to design an online algorithm based on the diminishing returns property of sub-

### 3. ONLINE PREFETCH ALGORITHM

The problem that we address in this paper is to maximize the byte-hit ratio through prefetching in the context of limited proxy-server bandwidth and stringent time constraints of video delivery. Note that the cache manager will generate prefetch requests which are supposed to retrieve video segments from the content server before they are actually required, competing for the proxy-server bandwidth with the cache-miss video segments that users are waiting for. Thus the prefetch manager needs an effective way to make decisions on the prefetch requests, which should be sent to the request pool and which should be discarded. In this section, we first formulate the problem, and then propose an online prefetching algorithm. We summarize the notations we use in Table 1.

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T$</td>
<td>Time interval of two successive requests sent by a user.</td>
</tr>
<tr>
<td>$Q$</td>
<td>Set of prefetch requests $Q = {q_1, q_2, ..., q_n}$.</td>
</tr>
<tr>
<td>$H$</td>
<td>Threshold of the request pool in Bytes.</td>
</tr>
<tr>
<td>$C$</td>
<td>Cache size in Bytes.</td>
</tr>
<tr>
<td>$B$</td>
<td>Proxy-server bandwidth.</td>
</tr>
<tr>
<td>$a$</td>
<td>Prefetching window.</td>
</tr>
<tr>
<td>$u_i$</td>
<td>Utility of $q_i \in Q$.</td>
</tr>
<tr>
<td>$w_i$</td>
<td>Weight of $q_i \in Q$.</td>
</tr>
<tr>
<td>$p_i$</td>
<td>Profit of $q_i$, $p_i = u_i w_i$.</td>
</tr>
<tr>
<td>$r_i$</td>
<td>Fill ratio of the request pool when $q_i$ arrives.</td>
</tr>
<tr>
<td>$u_{max}$</td>
<td>Maximum utility of prefetch requests in the request pool.</td>
</tr>
</tbody>
</table>

Table 1: List of notations and their definitions.

### 3.1 Problem Formulation

In the offline setting, the set of prefetch requests, denoted by $Q = \{q_1, q_2, ..., q_n\}$ is known a priori. The goal is to select a subset of $Q$, denoted by $S \subseteq Q$, such that the byte-hit ratio is maximized given the threshold of the request pool. The problem can be formulated as a classic 0/1 knapsack problem:

$$\max \sum_{i=1}^{n} u_i w_i x_i$$  

s.t. $\sum_{i=1}^{n} w_i x_i \leq H$  

where $x_i \in \{0, 1\}$, $\forall i \in \{1, 2, ..., n\}$

where $u_i$ and $w_i$ denote the utility and the weight of $q_i \in Q$, respectively. The utility of $q_i$ is defined as the number of users that may send $q_i$ within $\alpha T$ seconds, and the weight of $q_i$ is defined as the size of the corresponding video data in bytes. The indicator variable $x_i$ indicates whether $q_i$ is in the subset $S$ or not. The 0/1 knapsack problem, which is NP-hard, has been extensively studied in the last decades and several greedy and approximation solutions can be found in [14]. Although a fully polynomial time approximation scheme (FPTAS) can be achieved in the offline setting, the prefetch manager has to wait until enough requests have arrived.

Therefore, offline algorithms cannot be applied to applications with stringent time constraints. Furthermore, user request patterns are unpredictable and may present an unknown distribution, and therefore greedy algorithms based on assumptions of user request patterns (e.g., [7, 5]) may not be desirable in practice. To meet this challenge, we need online solutions which can process prefetch requests immediately as they arrive. More specifically, the prefetch manager has to make an irrevocable decision for each prefetch request upon arrival.

In the online setting, the prefetch manager is fed a sequence of prefetch requests $Q = \{q_1, q_2, ..., q_n\}$, where the subscript denotes the arrival time of the prefetch requests. Each prefetch request in $Q$ should either be sent to the request pool immediately or be discarded, as determined by the prefetch manager. Let $S_i$ be the set of prefetch requests that has been sent to the request pool right after the decision on $q_i$ is made. Then we can derive $S_i \subseteq S_{i+1} \subseteq S$, where $S \subseteq Q$ is the set of prefetch requests that will be sent to the request pool after all the prefetch requests in $Q$ have been processed. We define a set function $f : P(S) \rightarrow \mathbb{R}$, where $P(S)$ denotes the power set of $S$, as the number of bytes that will be hit at iPac if all the video segments of the prefetch requests in $S$ have been cached:

$$f(S) = \sum_{q_i \in S} u_i w_i.$$  

When the size of the request pool reaches $H$, the incurred queuing delay is $\Delta t = 8H/B$, where $B$ is the proxy-server bandwidth in Kbps. Now, the throughput of $v$ is about $\frac{8 \times \text{RWND}}{\text{RTT}}$, and the bit rate of $v$ is degraded to $b_j$:

$$b_j \leq \frac{8 \times \text{RWND}}{\text{RTT}} \leq b_{i+1}.$$  

Combining Eq. 1 and Eq. 2, we have the following:

$$\Delta t \leq \frac{b_{i+1} - b_j}{b_j} = \frac{b_{i+1} - b_j}{b_j}.$$  

To provide QoS guarantees for the application, we set the threshold of the request pool as follows:

$$H = \inf \left\{ \frac{\text{RTT} \cdot B \cdot (b_{i+1} - b_j)}{8b_j} \right\},$$  

where $\text{RTT}$ is the average RTT between video players and iPac. It is worth noting that the threshold of the request pool is only for prefetch requests, and it does not work for cache-miss requests, which are sent to the request pool unconditionally.
modular set functions. That is, for any $S \subseteq S' \subseteq Q$, and $q_i \in Q \setminus S$,
\[ f(S \cup \{q_i\}) - f(S) \geq f(S' \cup \{q_i\}) - f(S') \]  \hspace{1cm} (10)
Inequality 10 implies that the marginal gain of a prefetch request that is added to the solution set (i.e., $S$) is non-increasing in the offline setting. Therefore, we can send prefetch requests aggressively when the size of the request pool is small. As the size of the request pool increases, we should be more selective such that we only choose the prefetch requests with large marginal gains. The main challenge is to obtain good performance in the face of uncertainty.

3.2 The Online Prefetching Algorithm

Note that a well known 0.5-approximation offline algorithm for the classic 0/1 knapsack problem has been proposed by Dantzig and Du et al. [9, 11], where the items are ordered according to their profit-to-weight ratio, and the knapsack is filled from the item with the maximum profit-to-weight ratio. Inspired by this, we define the profit of a prefetch request $q_i$ as the video segment size of $q_i$, i.e., $u_i w_i$. The profit-to-weight ratio of $q_i$, i.e., $u_i$, is defined as the gain of $q_i$.

Unfortunately, in the general case, the online knapsack problem (i.e., Eq. 6), or its equivalent problem of online submodular maximization with knapsack constraints (i.e., Eq. 9), has no online algorithm that can achieve any bounded competitive ratio [20]. To meet this challenge, we assume that certain knowledge of the request pool is maintained, such that the prefetch manager can make decisions adaptively with advice from the past, i.e., the decisions it has made earlier.

The knowledge of the request pool that we maintain is the maximum gain of the prefetch requests in the request pool, denoted by $u_{\text{max}}$. One intuition is that a prefetch request with a higher gain is more likely to be sent to the request pool. Driven by this intuition, we use a scale factor to determine whether a prefetch request should be sent to the request pool or not. More precisely, a prefetch request $q_i$ is sent to the request pool if and only if
\[ u_i \geq \theta_i u_{\text{max}}, \quad \theta_i \in (0, 1] \]  \hspace{1cm} (11)
where $\theta_i$ is the scale factor that is used to process $q_i$.

Another intuition is derived from the diminishing returns property of the submodular function that we aim to maximize (i.e., Eq. 10). Specifically, as the request pool fills, we should be more selective due to the limited proxy-server bandwidth (i.e., knapsack constraint). We define the scale factor as
\[ \theta_i = 1 + \ln r_i, \]  \hspace{1cm} (12)
where $r_i$ is the fill ratio of the request pool upon arrival of $q_i$. The fill ratio of the request pool is defined as the ratio of the size of the request pool to the threshold of the request pool. Combining Eq. 11 and Eq. 12, we describe our online prefetching algorithm in Algorithm 1.

$u_{\text{max}}$ is updated upon arrival of each prefetch request, as shown in line 2 of Algorithm 1. The time and space complexity of this update operation is $O(1)$, thus updating $u_{\text{max}}$ is very efficient in practice. Note that a prefetch request with the utility $u_{\text{max}}$ will be deleted if its corresponding video segment is received. As a result, the inaccuracy of $u_{\text{max}}$ may be propagated as the number of prefetch requests increases. Therefore, $u_{\text{max}}$ should be updated periodically. In the implementation of iPac, $u_{\text{max}}$ is updated every 10 seconds by finding the maximum utility of the prefetch requests in the request pool.

4. COMPETITIVE ANALYSIS

The main challenge of an online algorithm is to obtain good performance in the context of unknown forthcoming input, which is revealed in parts. To evaluate the performance of an online algorithm, the competitive ratio is used to compare the performance of an online algorithm to an optimal offline algorithm which knows the entire input sequence in advance. Given any input sequence $\sigma$, an online algorithm $A$ is called $c$-competitive if $A(\sigma) \leq c \cdot O(\sigma)$, where $O(\sigma)$ is the maximum profit achieved by any optimal offline algorithm with complete knowledge of $\sigma$.

**Theorem 1.** The proposed online prefetching algorithm, denoted by $A$, is 0.5-competitive. That is, given any input sequence $Q$,
\[ A(Q) \geq \frac{1}{2} \cdot O(Q), \]  \hspace{1cm} (13)
where $O(Q)$ is the output of any optimal offline algorithm.

**Proof.** Let $Q = \{q_1, q_2, ..., q_n\}$ be the set of prefetch requests, where the subscript denotes the arrival time of the prefetch request. Let $S$ and $S^*$ ($S, S^* \subseteq Q$) be the set of prefetch requests selected by $A$ and $O$, respectively. Let $r$ be the fill ratio of the request pool after all the prefetch requests in $Q$ are processed. Without loss of generality, we assume $r H < \sum_{i \in Q} u_i w_i$. That is, some prefetch requests in $Q$ should be discarded due to the knapsack constraint $H$. In addition, based on the standard setting of competitive analysis, we assume no prefetch requests in the request pool will be removed.

Note that all the prefetch requests will be sent to the request pool until the fill ratio of the request pool exceeds $e^{-1}$, as shown in lines 3-4 of Algorithm 1. We then focus on the case when $e^{-1} < r \leq 1$. The output of the proposed online algorithm, denoted by $A(Q)$, can be presented as
\[ A(Q) = f(S \cap S^*) + f(S \setminus S^*) \]  \hspace{1cm} (14)
\[ = f(S^*) + f(S \cap S^* \setminus S^*) + f(S \setminus S^* \setminus S^*), \]  \hspace{1cm} (15)
where $S^*$ denotes the set of requests that are selected when the fill ratio of the request pool is less than $e^{-1}$. Similarly,

---

**Algorithm 1: Online Prefetch Algorithm.**

**Input:** $q_i := (u_i, w_i), r_i-1, S_{i-1}$.

1. $r_i \leftarrow r_{i-1} + w_i / H$;
2. $u_{\text{max}} \leftarrow \max(u_i, u_{\text{max}})$;
3. if $r_i \leq \frac{1}{2}$ then
4. $S_i \leftarrow S_{i-1} \cup \{q_i\}$;
5. else if $u_i \geq u_{\text{max}}(1 + \ln r_i)$ then
6. $S_i \leftarrow S_{i-1} \cup \{q_i\}$;
7. else
8. $S_i \leftarrow S_{i-1}$;
9. $r_i \leftarrow r_{i-1}$;
10. Discard $q_i$;
for each $q$, for each $S^* \cap S^r$, $f(S^* \cap S^r) + f(S^r \cap S^r) + f(S^r \cap S^r)$, as a result, the ratio of $O(Q)$ to $A(Q)$ can be written as follows:

$$O(Q) = f(S \cap S^r) + f(S^r \cap S)$$

(16)

$$= f(S^r) + f(S \cap S^r) + f(S^r \cap S')$$

(17)

Since $A(Q) \leq O(Q)$ holds for any given $Q$, we can derive $f(S^r \cap S) \geq f(S^r \cap S')$. As a result, the ratio of $O(Q)$ to $A(Q)$ can be written as follows:

$$O(Q) \leq f(S^r) + f(S^r \cap S') + f(S^r \cap S')$$

(18)

Note that $u_i \geq u_{\text{max}}(1 + \ln r_i)$ holds for any $q_i \in S \cap S^r$, as shown in line 5 of Algorithm 1. Thus we have

$$f(S \cap S^r) = \sum_{q_i \in S \cap S^r} u_i w_i$$

(19)

$$f(S^r \cap S) \geq \sum_{q_i \in S^r \cap S} u_{\text{max}}(1 + \ln r_i) w_i$$

(20)

By replacing the term $f(S^r \cap S')$ in Eq. 18 with $f(S^r \cap S')$, the inequality in Eq. 18 still holds because $f$ is a non-negative function and $f(S^r \cap S') \geq f(S^r \cap S')$. In addition, for each $q_i \in S \cap S^r$, we have $u_i \geq u_{\text{max}}(1 + \ln r_i)$ because $q_i$ is discarded by $A$. Conversely, $u_i \geq u_{\text{max}}(1 + \ln r_i)$ holds for each $q_i \in S \cap S^r$. After applying the above inequalities respectively to $f(S^r \cap S^r)$ and $f(S^r \cap S^r)$, we have

$$f(S^r \cap S^r) \leq \sum_{q_i \in S \cap S^r} u_{\text{max}}(1 + \ln r_i) w_i$$

(21)

$$f(S^r \cap S^r) \geq \sum_{q_i \in S^r \cap S} u_{\text{max}}(1 + \ln r_i) w_i$$

(22)

Combining Eq. 21, Eq. 22 and Eq. 20 with Eq. 18, we rearrange Eq. 18 to yield the following:

$$O(Q) = \sum_{q_i \in S \cap S^r} u_{\text{max}}(1 + \ln r_i) w_i$$

(23)

$$A(Q) < \sum_{q_i \in S \cap S^r} u_{\text{max}}(1 + \ln r_i) w_i$$

$$\leq \sum_{q_i \in S \cap S^r} u_{\text{max}}(1 + \ln r_i) w_i$$

(24)

$$\leq (1 + \ln r_i)(r - e^{-1}) H$$

(25)

where Eq. 24 holds because $r_i \leq r$ for any $q_i \in Q$. Note that $r_i = r_j + u_i / H$ holds for any $q_i, q_j \in S$, where $q_j$ is the latest request that has been selected by $A$ when $q_i$ is processed (i.e., $j \leq i - 1$). This implies that the denominator of the right term of Eq. 25 can be bounded (see Fig. 3) as follows:

$$\sum_{q_i \in S \cap S^r} (1 + \ln r_i) w_i \geq \int_{1/e}^r (1 + \ln r) dr$$

(26)

$$= r \ln r + e^{-1} - 1.$$ 

(27)

By applying Eq. 27 to Eq. 25, we derive the following:

$$O(Q) < \frac{(1 + \ln r)(r - e^{-1})}{r \ln r + e^{-1}} < 2.$$ 

(28)

Eq. 28 follows by applying Lemma 2 and yields $A(Q) > \frac{1}{2} \cdot O(Q)$, thus the claimed lower bound is satisfied.

Figure 3: An illustration of Eq. (27). For each $q_i \in S \cap S^r$, $(1 + \ln r_i) w_i$ denotes the area of the gray rectangle with the width $w_i / H$ and the height $(1 + \ln r_i)$. The area that all the gray rectangles cover is larger than the area formed by the curve $u_{\text{max}}(1 + \ln x)$, the x-axis, and the vertical line where $x = r$. This implies that Eq. (27) holds for any $r_i, r \in (0, 1]$.

**Lemma 2.** For any $x \in \mathbb{R}$, and $e^{-1} < x \leq 1$,

$$1 + \ln x(x - e^{-1}) \leq 2.$$ 

(29)

**Proof.** We define a real value function $g$ over $x$ as $g(x) := 2(x \ln x + e^{-1}) - (1 + \ln x)(x - e^{-1})$. The derivative of $g$, denoted by $g'(x) = x \ln x + (ex)^{-1}$, is non-decreasing because the second derivative of $g$, denoted by $g''(x) = x^{-1} - e^{-1}x^{-2}$, is non-negative for any $e^{-1} < x \leq 1$. Hence $g(x)$ is non-decreasing for any $e^{-1} < x \leq 1$. This implies that $g(x) \geq 0$ for any $e^{-1} < x \leq 1$ because $g(e^{-1}) = 0$. By applying $g(x) \geq 0$ to Eq. 30, the claimed bound is yielded.

It is worth remarking that the best approximation ratio that any offline algorithm can achieve is $1 - e^{-1} \approx 0.63$ [25].

5. **CACHE REPLACEMENT ALGORITHM**

Most of the popular cache replacement algorithms that are used in practical web proxies are based on a cache replacement rule called Least Recently Used (LRU) [6]. It achieves many desirable characteristics for small-size web content (i.e., text and images), including high hit ratio and low complexity. However, the main disadvantage of the LRU cache replacement algorithm is that it only considers the past and it has no indication of the future. In video streaming applications, many requests for video segments are sequential, which gives us a good idea about future requests. Therefore the LRU cache replacement algorithm is not desirable for video streaming applications.

Our cache replacement algorithm, as shown in Algorithm 2, incorporates the LRU replacement rule with future information of video segments (i.e., the utilities). More specifically, we compare the utilities of the received video segments (which could be either cache-miss segments or prefetched segments) with the least used segments in the cache if the cache is full. The least used video segments will be removed.
only when their utilities are less than the newly received video segments. This is different from the LRU cache replacement rule, by which the least used contents are always deleted when the cache is full.

Algorithm 2: Cache Replacement Algorithm.

Input: Received video segment \((u_i, w_i)\), \(cnt\), cache.

1. \( s = w_i; \)
2. \( \text{while } cnt > 0 \text{ and } s > 0 \text{ do} \)
3. \( \{ \text{if } u_j \leq u_i \text{ then} \)
4. \( \quad s \leftarrow s - w_i; \)
5. \( \text{else} \)
6. \( \quad \text{cache.push}(u_j, w_j); \)
7. \( \quad cnt \leftarrow cnt - 1; \)
8. \( \text{if } s \leq 0 \text{ then} \)
9. \( \quad \text{cache.push}(u_i, w_i); \)
10. \( \text{else} \)
11. \( \quad \text{Discard the received video segment;} \)

We set a counter (i.e., \( cnt \) in Algorithm 2, and \( cnt = 10 \) in our implementation) in every cache replacement operation for efficiency considerations. The LRU-based cache is implemented using the Boost bidirectional map structure\(^3\) that maintains cached video segments in the form of \(<\text{rid}, <\text{utility, weight, data}> >\), where \text{rid} and \text{data} denote the hash value of the request, and the actual video segment, respectively. The left key of the cache structure is maintained using the unordered_set_of structure, which has \(O(1)\) time complexity for query operations. The right key (i.e., \text{cache} in Algorithm 2) is maintained using the list_of structure such that the time complexity of both insertion and deletion is \(O(1)\). Meanwhile, the least used video segment can be popped since the newly inserted video segments will always be inserted at the end of the list (i.e., right key structure).

6. EVALUATION

We evaluate iPac by comparing it with three alternatives: (i) a conventional LRU-based caching approach (LRU), (ii) a popularity-based caching approach (PC), which caches the top-100 popular videos in advance, and (iii) an aggressive prefetching approach (AP), which prefetches the next video segment for each user request. In addition, we investigate the impact of the prefetching window size to the performance of iPac by comparing iPac-1 (i.e., \( \alpha = 1 \)), iPac-2, and iPac-3.

6.1 Experiment Setup

We deploy iPac and the content server in the US West and US East regions of the Amazon EC2 Cloud, respectively. The outgoing bandwidth of iPac is not shaped, but the proxy/server bandwidth is limited to 1 Gbps at wire speed. We observe that the RTT between the content server and iPac is around 90 ms.

The video players, each of which represents a unique user, are distributed in 300 machines on the PlanetLab. The video player is implemented based on a VLC DASH plugin [2] with modifications for efficiency considerations (only rate adaptation logic and buffer management modules are used). Each video player connects to iPac, instead of the content server,

\(^3\)http://bit.ly/1C2XvQF

to retrieve videos. We assume that each video is encoded into multiple versions [3] at different bitrates ranging from 350 kbps to 2040 kbps. Each video player will buffer 10 video segments before playing the video (i.e., the initial buffering stage). After that, it stays in the steady stage sending one request per two seconds to keep the playout buffer full (i.e., \( T = 2 \)). Each video player can switch to different bitrates according to its network conditions. More specifically, it switches to a higher bitrate if the measured throughput is at least 10% higher than the target bitrate. Once the measured throughput is less than the current bitrate and less than 50% of the playout buffer is filled, it will switch to a lower bitrate.

It is worth noting that frequent bitrate switching will significantly degrade user QoE since it can affect the user experience [8]. We applied the delayed update approach [13] to the video player which can avoid frequent bitrate switches that may annoy users. Fig. 8 shows how frequently the video players switch between bitrates in our experiments. It can be observed that, with a high probability of 0.83 (95-percentile), a video player will request the next video segment with the same bitrate as the current request.

The trace of user requests is shown in Fig. 5, which is presented in [27] (latest publicly accessible YouTube dataset). There are 18,661 user sessions in this 24-hour trace, and the CDF of the user session lengths is plotted in Fig. 4(c). There are 13,713 videos that have been requested, and Fig. 4(d) shows the popularity of the requested videos. It should be noted that the data set we use in the experiments is collected in 2008 (YouTube did not use adaptive bitrate streaming at that time), and the dataset is not very large. However, it suffices to evaluate the performance of different approaches under the same circumstances.

The CDF of the RTTs between video players (deployed in PlanetLab) and iPac (deployed in Amazon EC2 Cloud), and the CDF of video segment sizes are plotted in Fig. 4(a) and Fig. 4(b), respectively. Since the average RTT between users and iPac is about 150 ms, the threshold of the request pool, i.e., \( H \), is set to 34 MBytes, based on Eq. 4 and Fig. 4(b).

As shown in Fig 4(c), each session length is less than 200 seconds with high probability (95-percentile). We assume the maximum bitrate of video segments is 2,040 kbps. Therefore a storage with size 700 GB can store all the videos that have been requested in the data trace. In addition, we observe that only 876 videos were requested two or more

Figure 5: Number of new user sessions every 5 minutes.
times, and the total size of these videos is around 43.8 GB. In order to evaluate different caching and prefetching algorithms in the context of limited cache, we should use a relatively small cache.

To this end, we first limit the cache size, i.e., C, to 1 GB to compare the performance of different algorithms when the cache size is small. We then increase it to 10 GB to show the impact of the cache size to the performance of different approaches in terms of the byte-hit ratio, the user perceived quality, and the server load.

6.2 Byte-hit Ratio

To measure the effectiveness of the prefetching algorithms, the byte-hit ratio is used to observe how much of the video data is served by proxies instead of the content server.

Fig. 6(a) plots the byte-hit ratios of different approaches when C = 1 GB. The figure shows that iPac obtains up to 84% and 8× performance gains over AP, LRU and PC, respectively. When the arrival rate of user requests is relatively low (i.e., from 2 AM to 1 PM), both iPac and AP work very well such that byte-hit ratios higher than 0.8 can be achieved. In this case, iPac obtains 8× performance gains over LRU and PC. When the arrival rate of user requests is high (e.g., from 1 PM to 12 AM), iPac can still achieve a byte-hit ratio higher than 0.5, which is 5× higher than LRU and PC. We observe that iPac improves the performance by nearly 84% compared to AP, when the number of user requests is large. This is due to the fact that AP sends too many prefetch requests which compete for the proxy-server bandwidth with cache-miss requests. When C = 10 GB, the byte-hit ratios of iPac and AP are slightly improved, as shown in Fig. 6(b). However, the byte-hit ratio of LRU or PC is not improved effectively. This is due to the diversity of user request patterns, since video players can switch between different bitrates.

We observe that iPac can significantly improve the byte-hit ratio even when only the next video segment is considered for prefetching (i.e., iPac-1). It is worth noting that even an optimal prefetching algorithm cannot achieve a byte-hit ratio of 1 due to both the limited proxy-server bandwidth and the fact that the prefetched video segments may not be actually used.

Fig. 6(a) and Fig. 6(b) show that iPac-2 outperforms iPac-1, especially when the arrival rate of user requests is large. Note that iPac-1 only prefetches the next video segment for each user request; the time constraint\(^4\) of each prefetch request in iPac-1 is T. In comparison, the time constraint of every prefetch request in iPac-2 is 2T, so iPac-2 can achieve a better byte-hit ratio, especially when the number of user requests is large. However, increasing the prefetch window size (i.e., \(\alpha\)) may not always result in a higher byte-hit ratio. This is because the video players may change their bitrates, and thus the prefetched video segments may be useless. To understand this, let \(p\) be the probability that a video player will change the bitrate at the next request. The probability that at least one prefetched segment is useless is \(1 − (1 − p)^\alpha\), which increases as \(\alpha\) increases.

\(\text{Footnote:} \text{The time constraint of a prefetch request means the} \text{prefetched content should be retrieved before it is requested by a cache-miss request.}\)

---

\(^4\)The time constraint of a prefetch request means the prefetched content should be retrieved before it is requested by a cache-miss request.
It is worth remarking that iPac can achieve at least 50% of the performance of any optimal approach, as shown in Fig. 6(a) and Fig. 6(b). This is consistent with Theorem 1 in Section 4, which provides a worst-case guarantee for the proposed online prefetching algorithm with iPac.

6.3 User Perceived Quality

According to the measurements in [10], a user’s perceived video quality can be linked to the throughput of the user session. We use the average per-user throughput (APUT), defined as the average throughput of a user session to measure the QoS of different approaches. Note that the number of users during 12AM and 12PM is relatively small in the trace, we compare the APUT of different approaches during 1PM and 11PM. For iPac, we set the prefetch windows size to 2, since iPac-2 achieves the best byte-hit ratio as shown in Fig. 6(a) and Fig. 6(b).

Fig. 7(a) plots the CDF of APUT with different approaches when $C = 1$ GB. The figure shows that the median APUT of iPac is about 1100 kbps, which is 50% higher than the median APUT of LRU and PC, where the median APUT is about 722 kbps. In addition, iPac achieves a 31% performance gain over AP, where the median APUT is about 834 kbps. When the cache size is increased to 10 GB, as shown in Fig. 7(b), iPac achieves higher performance gains over LRU, PC, and AP. This is attributed to the higher byte-hit ratio that iPac achieves compared to LRU, PC, and AP. Therefore, the access latency with iPac is reduced, and the measured throughput at the client side is increased accordingly.

6.4 Server Load

Fig. 9 shows the number of requests (both prefetch requests and cache-miss requests) that are sent to the content server by the different approaches every 5 minutes. The prefetching window size of iPac and AP is 2. We see that the prefetching approaches (i.e., iPac and AP) incur additional load at the content server. This is the price to pay for higher byte-hit ratios and better user perceived quality. We observe that iPac achieves a 26% performance gain over AP in terms of server load. Because iPac achieves a higher byte-hit ratio than AP. This results in the fact that more requests can be satisfied by the proxy instead of content servers. In practice, suitable load balancing mechanisms should be applied to scale to a large number of users.

7. RELATED WORK

Recently, an increasing amount of video content is delivered over HTTP instead of classic streaming protocols (e.g., RTSP and RTMP) because HTTP-based video streaming does not require specialized server software and is easier to deploy. Besides, it is not a problem to deliver HTTP data through firewalls or NATs. More recent studies [19], [26] have focused on client-side rate adaptation algorithms for HTTP-based adaptive streaming since the client side is considered to be in the best position to measure the available bandwidth and respond to the changing network conditions. There is a considerable amount of prior work on metrics
that measure the perceived video quality [4] and how these metrics impact the user engagement [10], and user viewing patterns [21].

To offer better perceived video quality, a large number of caching strategies have been developed for video content delivery in the last decade. Prefix caching strategies have been proposed in [23], [24] where only the initial portion of each video or video segment is cached to reduce the initial play-out latency for users. However, these caching-only strategies cannot automatically ensure continuous streaming delivery to users. To this end, several prefetching strategies [7], [16] have been studied to reduce the transmission delay jitter.

We differ from all of the above approaches in that (i) we investigate large-scale HTTP-based adaptive streaming applications where user request patterns are unpredictable and the number of to-be-requested video segments can be huge, and (ii) we propose a 0.5-competitive online prefetching algorithm to maximize the byte-hit ratio with respect to the limited proxy-server bandwidth and the real-time constraints of video content. The problem is NP-hard [9], [15], and several offline approximation algorithms can be found in [14], [11]. It has been proved in [25] that the best competitive ratio that any offline algorithm can achieve is $1 - e^{-1} \approx 0.63$.

8. CONCLUSIONS

In this paper, we present iPac for HTTP-based adaptive streaming applications. To provide QoS guarantees to users, the byte-hit ratio at proxies should be maximized to reduce the access latency. We formulate the problem of maximizing byte-hit ratio in the context of limited proxy-bandwidth as a submodular maximization with knapsack constraints problem, which is NP-hard and the best approximation ratio of any offline greedy algorithm can achieve is $1 - e^{-1}$. Since offline algorithms cannot be applied to real-time applications, we propose an online prefetching algorithm with a competitive ratio of 0.5 which, to the best of our knowledge, is the best lower bound so far. We evaluate the performance of iPac based on a real-world implementation. Our results show that iPac can achieve a substantial performance gain over the state-of-the-art solutions. It is worth noting that the practicality of the proposed iPac proxy is high such that it is compatible with existing HTTP-based adaptive streaming implementations, and it can be easily applied to general real-time massive content delivery applications without any modification to content servers and video players.

9. ACKNOWLEDGMENTS

This study has been supported by the research grant for ADSC’s Human-Centered Cyber-physical Systems (HCICS) from Singapore Agency for Science, Technology and Research (A*STAR). The study is also supported by the Singapore National Research Foundation under its International Research Centre @ Singapore Funding Initiative and administered by the IDM Programme Office through the Centre of Social Media Innovations for Communities (COSMIC).

10. REFERENCES


