RL-Bélády: A Unified Learning Framework for Content Caching

Gang Yan
Binghamton University, State University of New York
gyan2@binghamton.edu

Jian Li
Binghamton University, State University of New York
lij@binghamton.edu

ABSTRACT

Content streaming is the dominant application in today's Internet, which is typically distributed via content delivery networks (CDNs). CDNs usually use caching as a means to reduce user access latency so as to enable faster content downloads. Typical analysis of caching systems either focuses on *content admission*, which decides whether to cache a content, or *content eviction* to decide which content to evict when the cache is full. This paper instead proposes a novel framework that can simultaneously learn both content admission and content eviction for caching in CDNs. To attain this goal, we first put forward a lightweight architecture for content next request time prediction. We then leverage reinforcement learning (RL) along with the prediction to learn the time-varying content popularities for content admission, and develop a simple threshold-based model for content eviction. We call this new algorithm RL-Bélády (RLB). In addition, we address several key challenges to design learning-based caching algorithms, including how to guarantee lightweight training and prediction with both content eviction and admission in consideration, limit memory overhead, reduce randomness and improve robustness in RL stochastic optimization. Our evaluation results using 3 production CDN datasets show that RLB can consistently outperform state-of-the-art methods with dramatically reduced running time and modest overhead.

CCS CONCEPTS

• **Information systems** → **Multimedia streaming**.

KEYWORDS

Cache Admission; Cache Eviction; Content Delivery Networks; Machine Learning; Bélády

ACM Reference Format:
Gang Yan and Jian Li. 2020. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proceedings of the 28th ACM International Conference on Multimedia (MM '20), October 12–16, 2020, Seattle, WA, USA*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3394171.3413524

1 INTRODUCTION

Recent years have seen an explosion of Internet traffic generated by multimedia applications including images, audio and video contents. Much of these contents are delivered by content delivery networks (CDNs) [9, 17], which consist of a large network of caches. The fundamental idea behind caching is to improve the end user's quality of experience by making content available at a location close to the end-user. If the requested content is available at a nearby cache, the user experiences a faster response time than it is served by the backend server. One major goal of a CDN is to decrease the user-perceived latency by maximizing the server's hit rate [13, 26–28], which is defined as the fraction of requests served from caches.

Upon a content request, the CDN cache server needs to make a two-fold decision: *content admission* and *content eviction*. Content admission determines whether to cache the requested content, while content eviction decides which content to evict when the cache is full. On the one hand, a large number of sophisticated eviction algorithms that may not be able to implement in practice have been proposed, e.g., LRU-K [25], A-LRU [19], LFU-DA [1, 30], Greedy-Dual-Size-Frequency (GDSF) [1, 8] and ARC [22]. However, Least-Recently-Used (LRU) and its variants (e.g., Segmented LRU [23]) are still the state-of-the-art eviction methods used in today's CDN cache servers. On the other hand, content admission has become an important technique to improve the performance of LRU caches by blocking some unpopular contents based on the request patterns. For example, size-based content admission only admits contents with sizes below a threshold, e.g., AdaptSize [4], while frequency-based content admission only admits contents with frequencies above a threshold, e.g., TinyLFU [10].

Recently, machine learning (ML) based caching algorithms have been proposed to either learn content popularities for content eviction, e.g., DeepCache [24], FNN-Cache [12], PA-Cache [11] and PopCache [32], or learn to decide whether or not to admit a content upon a request, e.g., RL-Cache [18] and CACA [15]. The drawback of these algorithms is that they typically have a good performance for some access systems and poorly for others. Furthermore, there is a still a big gap in hit rates between these state-of-the-art algorithms and the Bélády's offline MIN algorithm [2] on a number of production datasets [3, 31]. Finally, existing ML based approaches can only learn content admission or content eviction independently. However, content admission and eviction processes are interrelated and their performance have an impact on each other.

In this paper, we address these drawbacks and bridge the gap by proposing a novel ML approach with content admission and eviction in consideration at the same time. To the best of our knowledge, this is one of the first work on enabling learning of content admission and eviction simultaneously. Different from previous approaches that require complex content managements or optimize eviction policies, our goal is to design a lightweight and practicable ML-based cache operator to provide content admission and eviction services for existing CDN servers. To attain this goal, we design RL-Bélády (RLB) by first putting forward a lightweight architecture for content next request time prediction, and then leverage reinforcement learning (RL) along with the prediction to learn the

Dataset	*YouTube* [33]	*Bilibili* [5]	*Iqiyi* [20]
Duration (Hours)	330	18.7	9.9
Unique contents	297,920	4,852	162,104
Total requests (Millions)	0.6	1	1
Total bytes requested (TB)	57.22	16.79	107.21
Unique bytes requested (GB)	29,094	83	10,832
Mean Content Size (MB)	100	17.6	68.4
Max Content Size (MB)	101	17.7	38,392

Table 1: Characteristics of production datasets.

time-varying content popularities for content admission, and develop a simple threshold-based model for content eviction.

Upon a new request, RLB first extracts its features as the input of the prediction model, which outputs the predicted next request for the content. To decide whether to admit a content, our admission model trains a feedforward neural network (FNN) to compute an admission probability. Only a content with an admission probability greater than 0.5 is admitted. Training FNN on CDN datasets is computationally intensive and may suffer performance degradation due to the randomness caused by the stochastic optimization for policy selection. To design a proper and lightweight training model with stable performance, we develop two key techniques: limited stochastic optimization (LSO) and experts decision model (EDM). These techniques allow our model to quickly adapt to the changes in the datasets and provide a stable and better admission decision compared to state-of-the-art methods. Furthermore, they can also dramatically reduce the computational cost and training time.

Once admitting a content, RLB needs to decide which content to evict when the cache is full. The offline Bélády's algorithm always evicts the content with the furthest next request time. However, it is computationally expensive for a ML model to mimic its behaviors. To overcome this, our eviction model relaxes Bélády with a threshold to evict a content whose predicted next request time is beyond the threshold rather than the furthest one. However, predicting next request time for all cached contents at each time still incurs a high computational cost. Moreover, the real world request processes are well-known to be non-stationary [6, 33], hence an auto-tuning threshold value will be desired. To deal with these challenges, we design a model-free update rule for next request time prediction by taking advantage of the accuracy of our prediction model along with the lightweight LRU. We also develop a bootstrap based method for auto-tuning the threshold to make our eviction model be adaptive to time-varying requests.

We perform an evaluation on three real-world datasets collected from production CDNs. Although real-world datasets exhibit non-stationary request patterns, some may be very bursty (e.g., most contents have only be requested once in a short time scale), which makes the ML model hard to learn its request patterns. To capture this, we define a quality metric for real-world datasets, the *stability* that characterizes non-stationary request patterns. We show that RLB can outperform or at least match state-of-the-art methods in production datasets that exhibit different level of stabilities. In particular, RLB achieves a higher hit rate compared to existing ML-based methods with a dramatically reduced running time.

The rest of the paper is organized as follows. Section 2 analyzes challenges observed from production datasets and existing ML methods, and explain why we need to have a unified model that can learn both content admission and eviction. Section 3 describes the RLB and its design details are shown in Section 4. Evaluation results are given in Section 5. We conclude our paper in Section 6.

2 MOTIVATION AND INSIGHTS

In this section, we discuss the challenges in CDNs and state-of-the-art methods, and the opportunities to improve the CDN performance in a new dimension.

2.1 Challenges and Insights from Datasets

We consider three production datasets: (i) *YouTube* [33], which contains trace data about user requests for specific *YouTube* content collected from a campus network; (ii) *Bilibili* [5], which contains real HTTP request traces from Video-on-Demand service; and (iii) *Iqiyi* [20], which contains mobile video behaviors. The characteristics of these three datasets are summarized in Table 1.

To make better decisions on content admission and eviction, a caching algorithm needs to quickly learn which contents are the most popular. In other words, a good algorithm should be adaptive to the changes on popularity that might happen frequently. To further quantify the variability of these datasets, we plot their characteristics in terms of content popularity, content size, active bytes, inter-arrival time and stability.

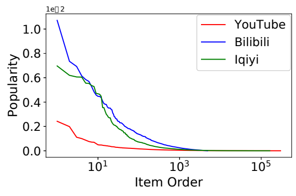


Figure 1: Content Popularity.

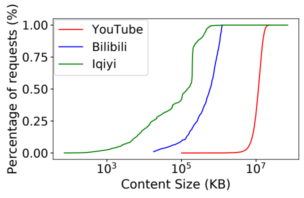


Figure 2: Content Size Distribution.

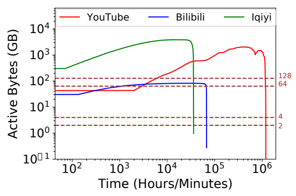


Figure 3: Active Bytes.

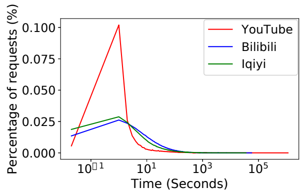
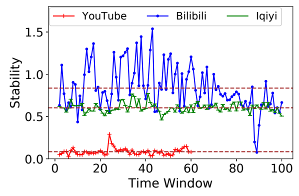


Figure 4: Inter-arrival times.

Content Popularity. Figure 1 shows that 80% of the requests are for less than 30% of the contents for all three datasets, which indicates that only a small portion of contents needs to be cached to achieve high hit rates.

Content Size. Figure 2 reveals the content size can vary significantly in these three datasets. In particular, the variation for YouTube dataset can be up to three orders of decimal magnitude.

Active Bytes. To decide a proper cache size for achieving higher hit rates, we use the metric of *active bytes* [18]. A content is said to be *active* at time t of a trace if t lies between the first and the last requests for the content. The total size of active contents at time t is defined as active bytes. Figure 3 plots active bytes for all three datasets. For each dataset, we choose two cache sizes to test caching algorithms. In particular, Bilibili with 2GB and 4GB caches, and YouTube and Iqiyi with 64GB and 128GB caches.



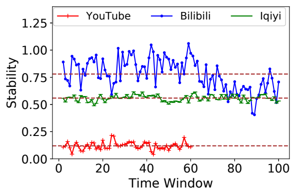


Figure 5: Stability for YouTube, Bilibili and Iqiyi with different cache sizes. *(Left)*: 64GB, 2GB and 64GB; *(Right)*: 128GB, 4GB and 128GB.

Inter-arrival Times. To quantify the non-stationary request process, Figure 4 depicts the distribution of content inter-arrival times in three datasets. As shown, the requested contents in YouTube have much higher variations than that of Bilibili and Iqiyi since YouTube contents have smaller inter-arrival times, i.e., most contents are only requested over a shorter time period. We can further see such variability though the *stability* metric discussed below.

Stability. To quantify changes in content requests, we introduce a notion of stability. We divide the dataset using a window of K consecutive requests and slide the window along the dataset by K requests at a time. The dataset is said to be more *stable* if contents requested in consecutive window do not change too much. To capture these properties, we first define the *inheritance rate* as $IH_{i+1} = |C_i \cap C_{i+1}|/|C_{i+1}|$, where C_i and C_{i+1} are the sets of cached contents in the i-th and $(i+1)$-th sliding windows, respectively; $C_i \cap C_{i+1}$ denotes the common contents of these two sets; and $|C|$ denotes the cardinality of set C. Since the content size is not uniformly distributed, inheritance rate is proportional to the cache size, which can be observed from numerical evaluations, and we omit it here due to space constraints. To eliminate the cache size impact, we use the hit rate which also depends on cache size. Putting them together, we define the stability as

$$S_{i+1} = \frac{|C_i \cap C_{i+1}|}{|C_{i+1}| \times H_{i+1}},$$

where H_{i+1} is the hit rates in the $(i+1)$-th window. *Stability* is relevant because it captures the request changes in the dataset. A stable trace usually has a relatively larger stability value since the number of common contents between two consecutive windows is large. Figure 5 plots the stability of these three datasets using LRU as a benchmark with $K = 10^4$. We can observe that Bilibili and Iqiyi have much larger values than YouTube, i.e., Bilibili and Iqiyi datasets are relatively more stable than YouTube, which is consistent with the observations from inter-arrival time distributions in Figure 4.

Key Observations. We observe extreme variability in content popularity and size distributions, and inter-arrival time distribution in three production datasets. This makes the cache management challenging in production settings. However, even for datasets with high variability, it may exhibit more *stable* request pattern. We later show that our proposed algorithms can take advantage of such *stability* to be able to adapt to the varying popularity, size and inter-arrival time characteristics to achieve higher hit rates.

2.2 Challenges of Existing Caching Algorithms

We validate the performance of two novel ML caching algorithms, *Learning Relaxed Bélády (LRB)* [31] for content eviction and *RL-Cache* [18] for content admission, using the above three production datasets. We discuss the challenges and insights obtained from these state-of-the-art methods. We also use two state-of-the-art non-ML based algorithms, LRU and AdaptSize [4] for comparison. The dynamics of hit rates with different cache sizes are shown in Figures 6, 7 and 8. We only present results for YouTube with 2GB cache, Bilibili and Iqiyi with 64 GB cache due to space constraints. Similar trends can be observed for other cache sizes.

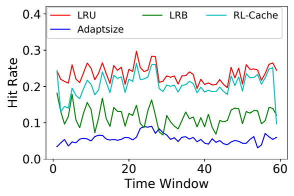


Figure 6: Dynamic hit rates for YouTube.

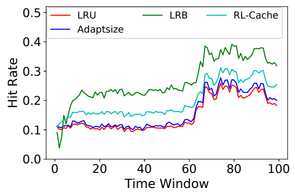


Figure 7: Dynamic hit rates for Bilibili.

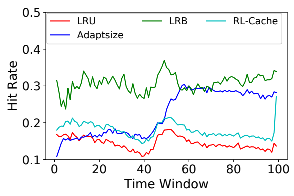


Figure 8: Dynamic hit rates for Iqiyi.

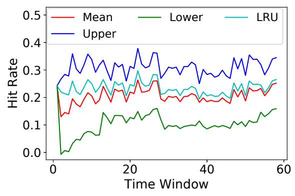


Figure 9: Randomness of RL-Cache for YouTube.

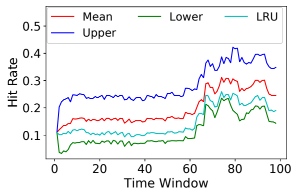


Figure 10: Randomness of RL-Cache for Bilibili.

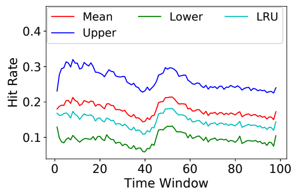


Figure 11: Randomness of RL-Cache for Iqiyi.

Learning Relaxed Bélády (LRB) [31]. LRB approximates the offline Bélády's algorithm [2] by randomly evicting a content whose *predicted* next request time is beyond a *pre-defined threshold*. LRB involves two key components: a Gradient Boosting Machine [14] (GBM) based model for content next request time prediction, and a pre-defined threshold to determine whether to evict the predicted contents. LRB has been shown to achieve a better performance compared to several state-of-the-art methods on CDN traces collected from Wikipedia, web and video traffics. However, the GBM model has to be frequently adopted in the caching system to update next request time for all cached contents. This is usually *time consuming* as it needs to update a large set of features for prediction. Furthermore, the threshold needs to be properly pre-defined, otherwise, it may lead to performance degradation.

From Figures 7 and 8, we observe that LRB can quickly learn the time-varying content popularities and adapt its learning strategy for a higher hit rates for Bilibili and Iqiyi datasets, which have higher stabilities. However, it performs poorly in YouTube dataset as shown in Figure 6, which exhibits very low stability in the requests.

RL-Cache [18]. RL-Cache uses model-free RL to decide whether to admit a requested content by directly searching for the optimal policy for each content via stochastic optimization. The training algorithm stops when the weights of neural network (NN) converge. However, although the policy selected by the stochastic optimization process would decrease hit rates, it is still being used to update NN weights. This will lead to a bad admission model due to the *randomness* from stochastic optimization.

We demonstrate the randomness in RL-Cache by plotting hit rate dynamics for all three datasets as shown in Figures 9, 10 and 11. For the sake of simplicity, we only use the first 20 windows and average over 50 runs. The upper and lower bounds correspond to the 95% confidence interval. We observe that RL-Cache can improve hit rates but may also lead to poor performance due to the randomness. Furthermore, the training of NNs is offline and computationally expensive. From the analysis in Section 2.1, requests in production systems may not be stable for a long time, and an effective admission model should be trained faster to catch the trends of the time-varying requests.

Key Observations. Learning-based caching admission or eviction algorithms can improve the performance over conventional methods. However, they may be computationally expensive in training (for both LRB and RL-Cache), need to manually tune parameters (for LRB), and have performance degradation due to non-stability in real datasets and randomness in algorithm design itself (for both LRB and RL-Cache). Therefore, there is still a big room for improving learning-based caching algorithm design from the perspective of both content admission and content eviction.

2.3 Desirable Properties and Requirements

From the above analysis, it is clear that we have to take the following key metrics into consideration when to design a learning-based caching algorithm and to deploy it on production systems.

- ***Hit rates.*** Achieve higher hit rates as the major goal of a CDN is to maximize the hit rates of its caches.
- ***Fast learning rate.*** Be able to quickly learn and adapt to the changes of requests [19, 29].
- ***Memory overhead.*** Require low memory overhead as large quantities are not available [4].
- ***No TPUs or GPUs requirement.*** Require no TPUs or GPUs which are not available in current production systems [16, 21].

3 RL-BÉLÁDY ALGORITHM

This section introduces the RL-Bélády (RLB) algorithm, which simultaneously makes content admission and content eviction decisions upon a new content request. We will discuss the design details of RLB in next section.

To overcome the issues in state-of-the-art learning-based caching algorithms, we first put forward a lightweight prediction model for the content next request time prediction. We then leverage reinforcement learning (RL) along with the prediction information to learn the time-varying content popularities for content admission. We also develop a simple threshold-based model for content eviction. We call our algorithm RL-Belady, which enable learning for both content admission and eviction at the same time.

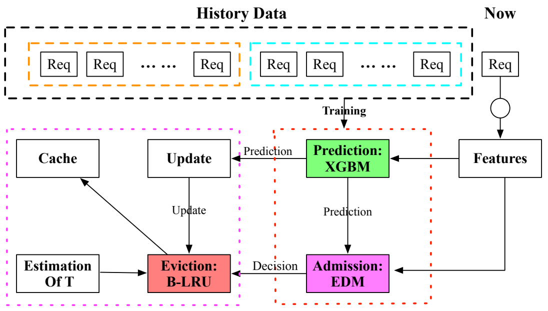


Figure 12: Architecture of RLB.

As shown in Figure 12, the architecture of RLB consists of three parts: prediction, admission and eviction. We consider a cache of size M, and denote the set of cached contents as $\mathcal{M}$. Analogous to the Bélády's algorithm, RLB maintains a vector of size M to keep the predicted next request time, NR_i of content i in the cache.

Upon a new request for content i, RLB first extracts its features u_i, and then outputs NR_i for content i via a prediction model (a XGBM based model). This will be described in details in Sections 4.2 and 4.4. If this request results in a cache hit, then RLB simply updates NR_i for all contents $i \in \mathcal{M}$ in the cache (see Section 4.6 for the update rules). Otherwise, our admission model (an EDM based model) takes the features as input and outputs an admission probability (see Section 4.5). If the admission probability is greater than 0.5, then content i is admitted and inserted into the cache. Otherwise, RLB does not admit content i into the cache.

Upon an admission, RLB needs to decide which content to evict when the cache is full. Based on predicted $\mathrm{NR}_i, \forall i \in \mathcal{M}$, our eviction model divide the cached contents into two sets based on a threshold T, whose value can be dynamically tuned (See Section 4.6). The first set contains contents whose predicted next request time is below T, and we call this $\mathcal{S}_1$. The second set contains contents whose predicted next request time is beyond T, and we call it $\mathcal{S}_2$. If $\mathcal{S}_2$ is not empty, RLB randomly evicts a content from $\mathcal{S}_2$. Otherwise, RLB evicts a content from $\mathcal{S}_1$ with the furtherest next request time. We call our eviction model that integrates the prediction as B-LRU[1].

4 SYSTEM DESIGN

We describe the design details of RLB in this section. To achieve this, we need to address two previously unsolved issues:

- How to efficiently train a unified ML model with learning of content admission and eviction simultaneously?
- How to guarantee an improved hit rates with respect to state-of-the-art methods with a low computational complexity?

There are several challenges in each of this issue and it requires additional effort to tradeoff some often-competing goals when we address them at the same time. Furthermore, there are two additional design issues for learning-based caching algorithm design:

History Data. To design a ML caching model, it is important to determine how much past information that can be used to train the model. Though the model will be more accurate if more history data is available, we need to limit this information in real cache system due to hardware constraints.

[1] B-LRU can be treated as RLB using admitting all as the content admission policy. See its performance in Section 5.

Learning Architecture. This will involve how to select and design ML architecture to achieve high hit rates, and select features, the prediction goal, and integrate them into decision making.

4.1 History Data

RLB records historical data including the information of contents that has been previously requested. For each request, it can be represented as a vector using content features, this will be described in details later. History data are used for training admission models in our framework.

Our framework takes content features as input and output a predicted content next request time for the eviction model, and a binary variable (i.e., admit or not) for the admission model. We consider a sliding time window and only history data within the window will be used for training and prediction.

A proper window size is important to the performance of our algorithm. On the one hand, the eviction and admission models will not be accurate if the window size is too small. On the other hand, if the window size is too large, it may increase the memory overhead and needs more time to process operations. In reality, CDN system should be able to process millions of requests per seconds, and the window size can be large given the effective system computing capability. To mimic the real CDN systems, we choose a window size of $10K$ for all datasets considered in this paper.

To warm up the training process at the beginning of a trace, we need data, which is not available at the beginning, to initial the model. To overcome this, we skip the first sliding window and use it as training data, and then implement algorithms from the second sliding window.

4.2 Content Features

Prediction Model. Three classes of features are selected: deltas, exponential decayed counters and static features.

▷ *Deltas.* Deltas capture the inter-request times between consecutive requests for a content. For example, Delta_1 represents the time since last request, Delta_2 is the time between the content's previous two requests, and so on. Deltas have been used as features in many algorithms, including the LRU with Delta_1.

▷ *Exponential Decayed Counters (EDCs).* EDCs approximate per-content request count over longer time periods. Each EDC_i is initialized to 0. When there is a request to content i, we first update Delta_1 and then $C_i = 1 + C_i \times 2^{-\text{Delta}_1/2^{9+i}}$. Note that C_i will not be updated until another request arrives for i.

▷ *Static Features.* This captures non-time varying content features, which includes content size and time duration of videos and so on. This type of feature plays a role in labeling different items.

These features provide a set of information used in state-of-the-art methods. We benefit from them by adding more features in the model training. However, adding more features may increase the system memory overhead, i.e., there is a tradeoff between adding more information to the model for decision making and the system overhead. We set the number of deltas and EDCs to be 32 and 10, respectively. Though it is similar to the features used in LRB, we use these features for next request time prediction rather than for eviction, and also they are used in a different way for model training (See Section 4.5). We denote these features as u_i for content i.

Admission Model. We only use 5 features for the admission model, which include the output from the prediction model, and the values of Delta_1 to Delta_4. The admission model takes the output of prediction model as one element of input and the output contains important information about one item, such as size. Then there is no need to add static features into admission model. We denote these features as u'_i for content i.

4.3 Training Data

To train our prediction and admission models, labeled data are needed. As for prediction model, inspired by LRB, for a content, RLB will wait until it is requested again. Once this occurs, the "future" information is available, and we will use this to determine the label. If the content will not be requested again after a long time, we will simply set its label as ∞.

For the admission model, we consider a sliding window of size K requests. The training data consists of a feature set $\{u'_i, i = 1, 2, \ldots, K\}$ and an optimal policy under the current window. Stochastic optimization (SO) is usually used to find this policy (a set of probabilities to admit K contents). The search process contains several repetitions. In particular, for the q-th search, a random variable $v_i \in [0, 1]$ is generated for each content request i. If $v_i \leq p_i(u'_i, w)$, then admit it and the policy for i now is 1. Otherwise the corresponding policy is 0. After all requests in the sliding window have been processed, we have a policy P_q with a corresponding hit rate H_q. Repeat this process for several times, and choose the policy with the highest hit rate as the optimal policy for the current window. However, there are some limitations for such SO approaches as described in Section 2.2. In this paper, we propose an improved approach called limited stochastic optimization (LSO) and will be described in details in Section 4.5.

4.4 Prediction Part

RLB uses the XGBoosting Machine (XGBM) [7] model for content next request time prediction. As mentioned above, the input to the prediction model consists of 32 deltas, 10 EDCs and static features, and the output is the predicted next request time for one content. To be more specific, the prediction model outputs the log value of the next request time. The output of the prediction model will be used in the admission and eviction decision parts, as illustrated in Figure 12. For the admission part, it is used as one of the features to the FNN. For the eviction part, it is used in two cases: (i) upon a cache hit, i.e., the requested content is in the cache, then update the next request time by the new one; (ii) upon a cache miss, i.e., the requested content is not in the cache. If the admission model decides to cache the content, then the prediction time will be used as a decision variable for content eviction.

4.5 Admission Decision Part

Content admission can be naturally formulated as a RL problem where contents in the cache define the system state, the admission of a content corresponds to an action and the hit rate is the reward of such admission. Similar to RL-Cache, we use RL to train our admission model. In order to decide whether to admit a content i upon request, our admission model uses a feedforward neural network (FNN) to compute an admission probability $p_i(u'_i, w) \in$

$[0, 1]$ as a function of content features u'_i and FNN weights w. If $p_i(u'_i, w) \geq 0.5$ then admit the requested content, otherwise not.

Training. To achieve this, we need to efficiently train the FNN with a manageable overhead to quickly learn time-varying requests so as to apply it to content eviction. We train the FNN using the Monte Carlo approach. We traverse the sequence of requests with a sliding window. Starting from the first request, the window slides forward by K requests every time to the end of the dataset. As discussed in Section 2.2, similar training framework has been used in RL-Cache, however, it suffers two major challenges: (i) *bad admission decision* due to the randomness by using stochastic optimization (SO) for searching the optimal policy for each content; (ii) *offline training with large overhead*. RL-Cache updates each training step until the FNN weights converge, which is computationally expensive and not necessary. To overcome these challenges with a stable and improved performance in hit rates and a lightweight training process, we develop two key techniques in the training: limited stochastic optimization (LSO) and experts decision model (EDM).

▷***Limited Stochastic Optimization (LSO)*** is the improvement of the SO based on historical data. In this paper, LSO and SO are used to directly find the optimal policy for contents in a sliding window based on the outputs of admission model. These outputs are the probabilities to cache contents, however, the SO model can not accurately compute such probabilities. Some popular contents that should be cached to improve hit rates may be assigned low admission probabilities, while some unpopular contents may be falsely assigned high admission probabilities. To overcome this, we take advantage of the trace real next request time, that is, NR_i here is the real next request time for item i. At time t, for each content i with NR_i, we compute the number of contents fall in $[t, \mathrm{NR}_i]$. A content j requested at time $t' > t$ is said to be in $[t, \mathrm{NR}_i]$ if $t' + \mathrm{NR}_j \leq \mathrm{NR}_i$. We denote the number of contents in $[t, \mathrm{NR}_i]$ as L_i. Hence we obtain a set $\mathcal{L} = \{L_1, \cdots, L_n\}$ for all contents. Denote the 5% and 95% points in $\mathcal{L}$ as Z_1 and Z_2, respectively. If $L_i < Z_1$ ($L_i > Z_2$) then content i is treated as a popular (unpopular) content, and the policy is 1 (0). For other cases, LSO reduces to SO. The idea behind LSO is that if L_i is large for content i, which means a large number of other contents will be requested before the next request for i, thus it is not necessary to cache i now. Similar argument holds for a small value of L_i.

▷***Experts Decision Model (EDM)*** represents a combination of Φ models where each model E_ϕ is an expert for $\phi \in \Phi$. When a content request arrives, each expert makes an admission decision, and a final decision to admit the requested content or not depends on the combination of all decisions from Φ. Different from a single expert model, in EDM, the decision from the $(\phi + 1)$-th expert is made based on the ϕ-th expert through LSO.

For each sliding window, the training process of the admission model consists of the iteration of following sequence of sampling, judging and learning steps, as shown in the dashed red box in Figure 13[2].

• ***Sampling.*** Let E be the current model with weights w and u'_i be the features of i-th request in current window, where $1 \leq i \leq K$. Since the admission decision can affect caching performance

[2]The training of the prediction model via XGBM (shown in the dashed purple box) is straightforward. Note that both models can be trained at the same time.

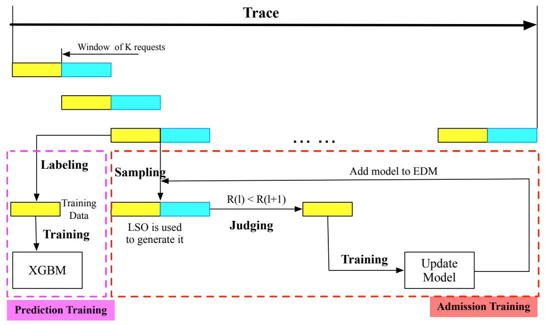


Figure 13: Training process of RLB.

beyond the current window, we extend the current window with additional L requests. As a result, we have $K+L$ requests to simulate the admission model and compute the hit rates over the extended window. For each request i, the EDM outputs a probability $E(u'_i, w)$ to admit the content. For all $K+L$ requests, we generate one training sample by using LSO.

• ***Judging.*** Inspired by the RL framework, we discount the hit rates of the additional L request for each sliding window. More specific, define a binary variable h_i as the outcome of the i-th request, and a cache hit corresponds to $h_i = 1$, and a miss with $h_i = 0$. Thus the hit rate for the l-th training is

$$R_l = \frac{\sum_{i=1}^{K} h_i + \sum_{i=K+1}^{K+L} \gamma^{i-K} h_i}{K + L}.$$

If $R_{l-1} < R_l$ holds, pass the training sample to next step.

• ***Learning.*** Directly use this training sample to retrain the FNN model $F_{nn}(l)$ to get a new $F_{nn}(l + 1)$. Then add the latest recent model into EDM and remove the least recent model from EDM. Suppose that EDM is a five expert models with $|\Phi| = 5$, then remove E_1 and add $F_{nn}(l + 1)$ into it and change their orders according to the time sequence.

Note that to improve the training efficiency, we only process the training steps with a few iterations (e.g., 5 iterations in our experiments) rather than to wait until weights are converged for each sliding window.

4.6 Eviction Decision Part

As described in Section 3, the admission part in RLB depends on the predicted next request time NR_i for each content i from the prediction model as well as a threshold T. Upon a request arrival, RLB needs to update NR_i $\forall i \in \mathcal{M}$ and the threshold T to decide which content to evict. As mentioned earlier, the idea behind our eviction model is similar to LRB. However, our algorithms differs and improves LRB and LRU from the following two perspectives:

Updating Rule. The updating rule in LRB requires to apply the Gradient Boosting Machines [14] (GBM) to all contents in the cache at each time. This is a model-based update, which is time consuming and heavy in memory overhead. To overcome this, we take advantage of the prediction accuracy of XGBM and the lightweight LRU. The update rule is given as

$$\mathrm{NR}_i(t) = \max\{\mathrm{NR}_i, \mathrm{LR}_i(t)\},$$

where $\mathrm{NR}_i(t)$ is the updated next request for content i at time t, $\mathrm{LR}_i(t)$ is the time since last request for i and NR_i is the initial predicted next request for i. Such an update can greatly reduce the algorithm complexity since there is no need to update the prediction every time if the prediction model has a good accuracy, and a content in the cache should be evicted without being requested for a long time. This is a model-free update and a key part of our eviction model in RLB.

Auto-tuning Threshold. To make the eviction part be adaptive to the variabilities in requests, an auto-tuning threshold T is desired. Further, rather than randomly evicting a content whose next request time is beyond T, a content i whose next request time $\mathrm{NR}_i(t)$ is further beyond T should be evicted with a higher probability compared to content j whose $\mathrm{NR}_j(t)$ is closer to T, i.e., $\mathrm{NR}_i(t) > \mathrm{NR}_j(t) > T$. However, such a probability is unknown and may depend on the predictor accuracy. To that end, we estimate such a threshold using a statistical bootstrap method. More specifically, we record the next-request time of all evicted contents in a eviction set E_v and use bootstrap based method to figure out the 95% point of T as the eviction threshold. If $\mathrm{NR}_i(t) > T$, then content i will be evicted with probability at least 95%. The idea behind is that T is the 95% percentile of E_v so that $\mathrm{NR}_i(t) > T$ holds, then we also have $\mathbb{P}(\mathrm{NR}_i(t) > T) \geq 0.95$. If $\mathrm{NR}_i(t)$ is much lower than the minimum value of E_v, then it is reasonable to keep it in cache and in this case $\mathbb{P}(\mathrm{NR}_i(t) > T) = 0$. To reduce the influence from the least latest evicted next-request time, E_v is updated dynamically. In other words, remove these least latest from E_v and add the latest time into it.

4.7 Putting Them Together

Putting all parts of our decision models in the RLB architecture, we have the complete design of RLB as illustrated in Figure 12. RLB learns to cache contents using a sliding window of K requests through the prediction, admission and eviction models as described above. These processes repeat for each sliding window. However, we observe from our experiments that it is not necessary to update admission model in each sliding window. The model trained in one window can still achieve a good performance in several following windows. This can significantly reduce the computational complexity and memory overhead.

5 EXPRIMENT EVALUATION

In this section, we evaluate the performance of RLB using both synthetic and production datasets, and compare with it to several state-of-the-art methods.

5.1 Methodology

Neural Network and XGBM. In NNs, we use two activation functions: ReLU and Sigmoid, with the optimizer *Adam*. For XGBM, we set the number of trees as 50 and the learning rate as 0.1. The loss functions for these two are both the mean-squared errors (MSE).

Algorithm Settings. Our evaluation uses the following default values for RLB parameters. Each sample is generated using a sliding window size is $K = 10\mathrm{K}$ requests following by $L = 10\mathrm{K}$ extra requests. The EDM has 5 expert models. The policy searched by LSO and SO is set to be 0.2 or 0.8 rather than 0 or 1. This will increase the probability of evicting unpopular contents from the cache.

Baselines. As our RLB learns both content admission and content eviction, we compare it with five state-of-art eviction algorithms: (1) GDSF, an improvement of Greedy-Dual-Size algorithm proposed for Web caches; (2) LFU-DA, least-frequently used eviction with dynamic aging; (3) LRU-K [25], evicting content with the oldest K-th reference in the past; (4) AdaptSize, a Markov cache model that adapts caching parameters to the changing request patterns; and (5) the LRB cache. In particular, we also compare with B-LRU (see Section 3).

Performance Evaluation. We evaluate the performance of these algorithms using the three datasets we described in Section 2.1 with different cache sizes, which are chosen based on the active bytes. We also consider a synthetic dataset generated by Zipf distribution. All results are generated by running on Ubuntu 18.04 with an Intel(R) Core(TM) i7-6700HQ processor and a 8GB RAM.

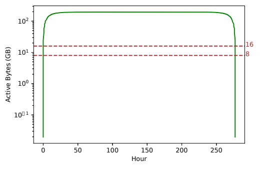


Figure 14: Active bytes.

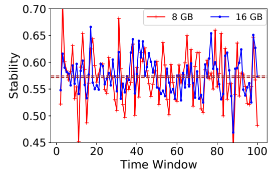


Figure 15: Stability.

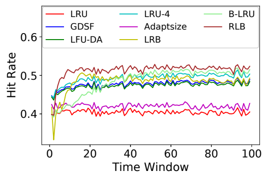


Figure 16: Dynamics of hit rates for synthetic datasets.

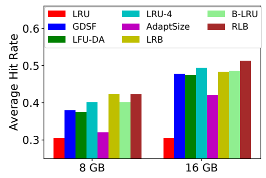


Figure 17: Average hit rates for synthetic datasets.

5.2 Evaluation on Synthetic Datasets

We generate a synthetic dataset for consideration, where requests are sampled from a Zipf distribution with parameter $\alpha = 0.8$. The content size is generated by Poisson distribution with a mean size of 20MB. We also use other values for α and observe similar performance. Hence are omitted here due to space constraints.

Based on the active bytes shown in Figure 14, we choose the cache sizes to be 8GB and 16GB. The stability is presented in Figure 15. Since this dataset is generated from a Zipf distribution, it is already relatively stable, i.e., it should have a relatively high stability value. This is indeed observed in Figure 15, which further validates the correctness of our stability metric.

Figure 16 shows the dynamic hit rates of different caching algorithms on 16GB caches. Similar observations can be made on 8 GB caches and hence are omitted here due to space constraints. Their average performance is illustrated in Figure 17. We observe that in both cases, our RLB can outperforms all candidate state-of-the-art methods. In particular, RLB improves the hit rates by up to 10% compared to LRB for a 16GB cache, and reduce the corresponding training time by up to 50%, as shown in Figure 24. More interestingly, our eviction only algorithm B-LRU can still outperform LRB on 16GB cache with a dramatically reduced running time.

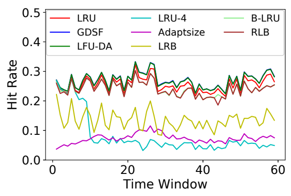


Figure 18: Dynamics of hit rates for YouTube dataset.

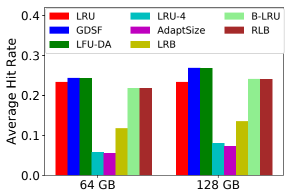


Figure 19: Average hit rates for YouTube dataset.

5.3 Evaluation on Real Datasets

We further compare the performance of different caching algorithms using the three production datasets. The dynamics of hit rates for YouTube, Bilibili and Iqiyi datasets are shown in Figures 18, 20 and 22, respectively. We only show the dynamics with one cache size, e.g., YouTube with 128 GB cache, Bilibili with 4 GB cache and Iqiyi with 128 GB cache. Similar observations can be made for YouTube with 64 GB cache, Bilibili with 2 GB cache and Iqiyi with 64 GB cache, and hence are omitted here due to space constraints. The average hit rates are presented in Figures 19, 21 and 23. Finally, the running times and memory overhead for YouTube with 128 GB cache, Bilibili with 4 GB cache and Iqiyi with 128 GB cache are given in Figures 24 and 25, respectively.

From the stability analysis in Section 2.1, we see that YouTube dataset exhibits an unstable request pattern, where most contents have only been requested once in a short time scale (see Figure 4) during in the whole datasets. In contrast, both Bilibili and Iqiyi datasets have relatively stable request patterns.

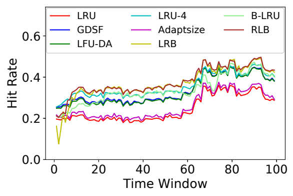


Figure 20: Dynamics of hit rates for Bilibili dataset.

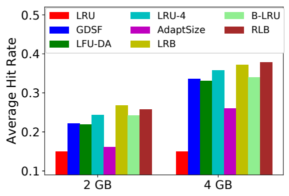


Figure 21: Average hit rates for Bilibili dataset.

For the stable Bilibili and Iqiyi datasets, RLB can outperform the best state-of-the-art methods (LRB in most cases) by up to 5% in hit rates. In particular, RLB can dramatically reduce the running time compared to LRB, which is a desirable property when applied to production systems. For the unstable YouTube dataset, RLB does not always perform better as shown in Figure 19. The reason is that YouTube dataset in our consideration has a very low mean stability (See Figure 5), with less information can be learned from the history. However, RLB can still significantly outperform LRB in both average hit rates and running time. If there are some abrupt changes in the datasets, which are usually the cases, RLB can still achieve a good performance. More interestingly, our eviction only algorithm B-LRU can significantly improve hit rates over LRB even on unstable datasets with a dramatically reduced running time.

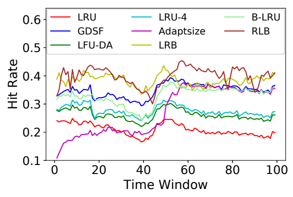


Figure 22: Dynamics of hit rates for Iqiyi dataset.

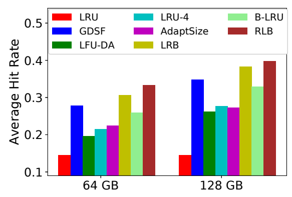


Figure 23: Average hit rates for Iqiyi dataset.

Note that the EDM model in our proposed RLB algorithm only needs to update itself at the first few sliding windows during the training process. Only when the trace patten changes fast, it will update itself again. This is due to the nature of our proposed EDM model that guarantees a stable and robust performance, which can significantly reduce the running time for RLB. To improve the efficiency of the state-of-art-method LRB, we use batch method to update its quantities.

Finally, Figure 25 presents the memory overhead for training three learning-based algorithms. On the one hand, we observe that RLB indeed requires more memory overhead compared to LRB. However, the value is much smaller than the available cache size. For a given cache size, though RLB costs memory overhead (leaving less cache space for contents), it can still achieve a better hit rate compared to LRB with a dramatically reduced running time. For non-stable YouTube dataset, B-LRU costs even less memory overhead than LRB with a higher hit rate and shorter running time. Similar trends are observed for Iqiyi and hence are omitted here.

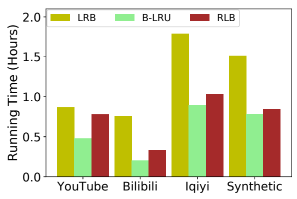


Figure 24: Running time for all traces.

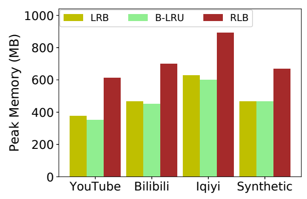


Figure 25: Peak memory for all traces.

6 CONCLUSION

In this paper, we designed and evaluated RL-Bélády (RLB) algorithm. RLB is one of the first work on learning content admission and eviction at the same time. It consists a lightweight architecture for content next request time prediction, an RL enabled architecture for content admission and a simple threshold-based method for content eviction. We extensively evaluated RLB using 3 production datasets and compared it with several state-of-the-art methods. We showed that RLB can outperform or at least match state-of-the-art methods. Particularly, RLB achieves a higher hit rate compared to existing ML-based methods with a dramatically reduced running time and a modest memory overhead.

REFERENCES

[1] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. 2000. Evaluating Content Management Techniques for Web Proxy Caches. *ACM SIGMETRICS Performance Evaluation Review* 27, 4 (2000), 3–11.

[2] Laszlo A. Bélády. 1966. A Study of Replacement Algorithms for A Virtual-Storage Computer. *IBM Systems journal* 5, 2 (1966), 78–101.

[3] Daniel S Berger. 2018. Towards Lightweight and Robust Machine Learning for CDN Caching. In *Proc. of ACM HotNets*.

[4] Daniel S Berger, Ramesh K Sitaraman, and Mor Harchol-Balter. 2017. AdaptSize: Orchestrating the Hot Object Memory Cache in a Content Delivery Network. In *Proc. of USENIX NSDI*.

[5] Bilibili. [n.d.]. https://www.bilibili.com.

[6] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. 2009. Analyzing the Video Popularity Characteristics of Large-Scale User Generated Content Systems. *IEEE/ACM Transactions on networking* 17, 5 (2009), 1357–1370.

[7] Tianqi Chen and Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System. In *Proc. of ACM SIGKDD*.

[8] Ludmila Cherkasova. 1998. *Improving WWW Proxies Performance with Greedy-Dual-Size-Frequency Caching policy*. Hewlett-Packard Laboratories.

[9] VNI Cisco. 2015. Cisco Visual Networking Index: Forecast and Methodology 2014–2019 White Paper. *Cisco, Tech. Rep* (2015).

[10] Gil Einziger, Roy Friedman, and Ben Manes. 2017. TinyLFU: A Highly Efficient Cache Admission Policy. *ACM Transactions on Storage (ToS)* 13, 4 (2017), 1–31.

[11] Qilin Fan, Jian Li, Xiuhua Li, Qiang He, Shu Fu, and Sen Wang. 2020. PA-Cache: Learning-based Popularity-Aware Content Caching in Edge Networks. *arXiv preprint arXiv:2002.08805* (2020).

[12] Vladyslav Fedchenko, Giovanni Neglia, and Bruno Ribeiro. 2019. Feedforward Neural Networks for Caching: Enough or Too Much? *ACM SIGMETRICS Performance Evaluation Review* 46, 3 (2019), 139–142.

[13] N Choungmo Fofack, Philippe Nain, Giovanni Neglia, and Don Towsley. 2012. Analysis of TTL-based Cache Networks. In *Proc. of IEEE VALUETOOLS*.

[14] Jerome H Friedman. 2001. Greedy Function Approximation: A Gradient Boosting Machine. *Annals of statistics* (2001), 1189–1232.

[15] Yu Guan, Xinggong Zhang, and Zongming Guo. 2019. CACA: Learning-based Content-Aware Cache Admission for Video Content in Edge Caching. In *Proc. of ACM Multimedia*.

[16] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. 2014. Trade-Offs in Optimizing the Cache Deployments of CDNs. In *Proc. of IEEE INFOCOM*.

[17] W. Jiang, S. Ioannidis, L. Massoulié, and F. Picconi. 2012. Orchestrating Massively Distributed CDNs. In *Proc. of ACM CoNEXT*.

[18] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. 2019. RL-Cache: Learning-based Cache Admission for Content Delivery. In *Proc. of Workshop on Network Meets AI & ML*.

[19] Jian Li, Srinivas Shakkottai, John CS Lui, and Vijay Subramanian. 2018. Accurate Learning or Fast Mixing? Dynamic Adaptability of Caching Algorithms. *IEEE Journal on Selected Areas in Communications* 36, 6 (2018), 1314–1330.

[20] Ge Ma, Zhi Wang, Miao Zhang, Jiahui Ye, Minghua Chen, and Wenwu Zhu. 2017. Understanding Performance of Edge Content Caching for Mobile Video Streaming. *IEEE Journal on Selected Areas in Communications* 35, 5 (2017), 1076–1089.

[21] Bruce M Maggs and Ramesh K Sitaraman. 2015. Algorithmic Nuggets in Content Delivery. *ACM SIGCOMM Computer Communication Review* 45, 3 (2015), 52–66.

[22] Nimrod Megiddo and Dharmendra S Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache.. In *Proc. of USENIX FAST*.

[23] Kathlene Morales and Byeong Kil Lee. 2012. Fixed Segmented LRU Cache Replacement Scheme with Selective Caching. In *Proc. of IEEE IPCCC*.

[24] Arvind Narayanan, Saurabh Verma, Eman Ramadan, Pariya Babaie, and Zhi-Li Zhang. 2018. Deepcache: A Deep Learning Based Framework for Content Caching. In *Proc. of Workshop on Network Meets AI & ML*.

[25] Elizabeth J O'neil, Patrick E O'neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. *Acm Sigmod Record* 22, 2 (1993), 297–306.

[26] Nitish K Panigrahy, Jian Li, and Don Towsley. 2017. Hit Rate vs. Hit Probability Based Cache Utility Maximization. *ACM SIGMETRICS Performance Evaluation Review* 45, 2 (2017), 21–23.

[27] Nitish K Panigrahy, Jian Li, Don Towsley, and Christopher V Hollot. 2020. Network Cache Design under Stationary Requests: Exact Analysis and Poisson Approximation. *Computer Networks* (2020), 107379.

[28] Nitish K Panigrahy, Jian Li, Faheem Zafari, Don Towsley, and Paul Yu. 2017. Optimizing Timer-based Policies for General Cache Networks. *arXiv preprint arXiv:1711.03941* (2017).

[29] Georgios S Paschos, George Iosifidis, Meixia Tao, Don Towsley, and Giuseppe Caire. 2018. The Role of Caching in Future Communication Systems and Networks. *IEEE Journal on Selected Areas in Communications* 36, 6 (2018), 1111–1125.

[30] Ketan Shah, Anirban Mitra, and Dhruv Matani. 2010. An O(1) Algorithm for Implementing the LFU Cache Eviction Scheme. *no* 1 (2010), 1–8.

[31] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. 2020. Learning Relaxed Belady for Content Distribution Network Caching. In *Proc. of USENIX NSDI*.

[32] Kalika Suksomboon, Saran Tarnoi, Yusheng Ji, Michihiro Koibuchi, Kensuke Fukuda, Shunji Abe, Nakamura Motonori, Michihiro Aoki, Shigeo Urushidani, and Shigeki Yamada. 2013. PopCache: Cache More or Less Based on Content Popularity for Information-Centric Networking. In *Proc. of IEEE Conference on Local Computer Networks*.

[33] Michael Zink, Kyoungwon Suh, Yu Gu, and Jim Kurose. 2008. Watch Global, Cache Local: YouTube Network Traffic at A Campus Network: Measurements and Implications. In *Multimedia Computing and Networking 2008*, Vol. 6818. 681805.