Learning from Optimal Caching for Content Delivery

Gang Yan
SUNY-Binghamton University
Binghamton, NY, USA
gyan2@binghamton.edu

Jian Li
SUNY-Binghamton University
Binghamton, NY, USA
lij@binghamton.edu

Don Towsley
University of Massachusetts
Amherst, MA, USA
towsley@cs.umass.edu

ABSTRACT

Content delivery networks (CDNs) distribute much of today's Internet traffic by caching and serving users' contents requested. A major goal of a CDN is to improve hit probabilities of its caches, thereby reducing WAN traffic and user-perceived latency. In this paper, we develop a new approach for caching in CDNs that learns from optimal caching for decision making. To attain this goal, we first propose *HRO* to compute the upper bound on optimal caching in an online manner, and then leverage HRO to inform future content admission and eviction. We call this new cache design *LHR*. We show that LHR is efficient since it includes a detection mechanism for model update, an auto-tuned threshold-based model for content admission with a simple eviction rule. We have implemented an LHR simulator as well as a prototype within an Apache Traffic Server and the Caffeine, respectively. Our experimental results using four production CDN traces show that LHR consistently outperforms state of the arts with an increase in hit probability of up to 9% and a reduction in WAN traffic of up to 15% compared to a typical production CDN cache. Our evaluation of the LHR prototype shows that it only imposes a moderate overhead and can be deployed on today's CDN servers.

CCS CONCEPTS

• **Theory of computation → Caching and paging algorithms**;
• **Computing methodologies → Machine learning**.

ACM Reference Format:
Gang Yan, Jian Li, and Don Towsley. 2021. Learning from Optimal Caching for Content Delivery. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany*. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3485983.3494855

1 INTRODUCTION

Caches are pervasive in networked systems, and play a critical role in end-to-end application performance. For example, content delivery networks (CDNs) deploy hundreds of thousands of servers across the world to serve user requests. If the requested content resides in servers near the user, a *cache hit* occurs and the user promptly receives the content. Otherwise a *cache miss* occurs and the response time degrades dramatically. Consequently, there has been a renewed focus on increasing cache hit probabilities, particularly for content delivery [12, 20, 21, 34, 46, 64].

Algorithms that determine what contents to cache in each CDN server play a key role in achieving a high hit probability. Upon a content request, the CDN server makes *cache admission* and *cache eviction* decisions. Cache admission determines whether to cache the content while cache eviction decides what content to evict when the cache is full. Indeed, there is a plethora of work focusing on improving caching performance by proposing admission and eviction algorithms. However, major CDNs today still employ the classic Least Recently Used (LRU) [22] or its variants for content caching [46]. In spite of decades of studies on caching, including a recent trend on using machine learning (ML) techniques, *the fundamental limitation of existing caching algorithms remains*. We still observe a large gap between hit probabilities of state-of-the-art caching algorithms (SOTAs) and the Bélády's offline MIN algorithm [9].

One way to understand such a large gap in caching design is by studying the performance of the offline optimum algorithm (OPT). The classic such algorithm is Bélády's algorithm [9]. Although OPT cannot be implemented in production systems since it assumes exact knowledge of future requests, it can provide guidance on designing practical caching algorithms as well as bounding their performance. For example, if OPT can achieve a hit probability of x%, then any online caching algorithm can achieve *at most* x%. Our approach to improving the performance of CDN caches is to use lessons learned from OPT to guide our design of a practical learning augmented online caching algorithm.

Limitations of existing algorithms. We begin in Section 2 by showing the fundamental limitations of existing state-of-the-art caching algorithms. We measure hit probabilities using four production traces with hundreds of millions of requests across cache sizes from 64GB to 1,024GB. We find that there remains a gap of 15%-25% between state of the arts and the Bélády algorithm.

Optimal caching. Having demonstrated such a large gap, we turn to the question: *how much further hit probability can be improved*. To provide a principled answer to this question, an *upper bound* on maximum achievable hit probability[1] has been widely used as a performance metric. For equal size contents, it is well known that OPT is the Bélády algorithm. However, content sizes often vary widely in production CDNs from a few bytes [50] to several gigabytes [34]. Unfortunately, computing OPT for variable content sizes is known to be NP-hard [19]. Existing bounds widely used by practitioners, e.g., PFOO [11] are *offline* and can be fragile (Section 2).

System designers often have no access to the exact request trace but can estimate statistical properties of the content request process such as the inter-request time distribution. Thus we address the

CoNEXT '21, December 7–10, 2021, Virtual Event, Germany

ACM ISBN 978-1-4503-9098-9/21/12…$15.00
https://doi.org/10.1145/3485983.3494855

[1] For the notation abuse, we also call the maximum achievable hit probability for online state-of-the-art caching algorithms as OPT.

Dataset	*CDN-A*	*CDN-B*	*CDN-C*	*Wiki*
Duration (Hours)	24	9.9	330	0.1
Unique contents	330,446	162,104	297,920	406,883
Total requests (Millions)	0.97	1	0.6	1
Total bytes requested (TB)	36.42	107.21	57.22	34.34
Unique bytes requested (GB)	8,242	10,832	29,094	27,618
Active bytes (GB)	1,183	2,180	286	1,642
Mean content size (MB)	25.5	68.4	100	69.5
Max content size (MB)	7,790	38,392	101	92,100

Table 1: Key characteristics of production traces used throughout our evaluation spanning different CDNs.

above challenges by designing a practical *online* upper bound on OPT using *the hazard rate density function* (Section 3). We call this upper bound *HRO*, which can be computed in *an online* manner for both *equal and variable sized* contents with limited knowledge on content arrivals and no look ahead option. We then quantify the gap between SOTAs and HRO, PFOO, and Bélády. We find that Bélády and PFOO are 4%-16% worse than HRO on average.

Learning to leverage optimal caching. We use HRO as our guide for the design of practical learning augmented online caching algorithms in Section 4. Specifically, we present a novel ML approach that is fundamentally different from SOTAs. Our approach is to leverage HRO into content caching, using ML to simultaneously find content to admit and evict based on recent content request patterns. A naive ML algorithm that imitates HRO would incur prohibitively high computational costs. To overcome this challenge, our key insight is that it suffices to leverage a lightweight supervised learning based model along with HRO to learn a content admission probability, which is used for both content admission and eviction. We call this new algorithm LHR (i.e., learning from HRO).

Upon a new request, LHR first extracts its features and trains a supervised learning model using HRO, which outputs an admission probability. LHR only admits a content whose admission probability is greater than a *threshold*. To set a proper threshold, we develop a simple *estimation algorithm* that *adaptively* adjusts the threshold value based on recent requests. We show that the hit probability of LHR using this *auto-tuned threshold* can be significantly larger than that for a fixed threshold (e.g., 0.5) that has been widely used in the literature (Sections 5.2.3 and 7.4).

Once admitting a content, LHR needs to decide what content to evict when the cache is full. Unlike prior approaches that use a small set of static or learned content features for eviction, we design a simple *eviction rule* in LHR that weights a larger set of features including the learned admission probability. The auto-tuned threshold and the simple eviction rule together give our system a much more accurate set of choices to aim for, which has one major consequence: it allows our system to quickly and accurately adapt to changes in the workload and provide better caching decisions compared to SOTAs (Sections 5.2.5 and 7.4).

Even with the above design insights for LHR, training a potential ML framework on CDN traces is often computationally intensive and time consuming. To reduce computational complexity, we consider a *sliding time window* over content requests. Naively training the learning model over every sliding window would still incur high computational cost. We instead update the model only when the request patterns in two consecutive time windows exhibit significant changes. To that end, we develop a lightweight *detection mechanism* and we observe that it leads to a dramatic decrease in implementation overhead (Sections 5.2.2 and 7.4).

These insights enable us to design LHR. We have implemented an LHR simulator as well as an LHR prototype within an Apache Traffic Server [3] and the Caffeine [47], respectively. Our extensive evaluations using four production CDN traces show that LHR consistently outperforms all candidate SOTAs with an increase in content hit probability of up to 9% and a reduction in WAN traffic of up to 15%. Furthermore, our prototype evaluation shows that LHR incurs comparable computational and memory overheads to other designs. Hence LHR can be deployed on today's CDN servers.

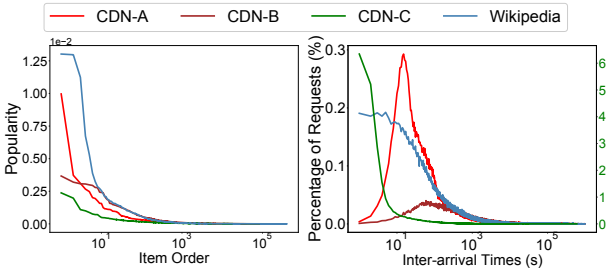
2 BACKGROUND AND MOTIVATION

Real world systems usually exhibit quirks and dynamic behaviors that are hard to capture with tractable assumptions. In this section, we quantify opportunities in such production systems, discuss limitations of existing SOTAs and offline bounds on OPT, as well as challenges to the design of practical online caching algorithms.

Real request traces are diverse. We consider production traces from four CDNs, three of which chose to remain anonymous. (1) CDN-A collected from several nodes in one continent serves a mixture of web and video traffic; (2) CDN-B captures mobile video behaviors collected from one live streaming system; (3) CDN-C contains user requests for specific contents collected from a local network; and (4) a Wikipedia (Wiki) trace collected on a west-coast node serving photos and other media content. Key trace characteristics are summarized in Table 1 and Figure 1.

The traces typically span several tens to hundreds of thousands of requests, and tens of thousands of contents with sizes varying from 10 KB to 10^4 MB. The total bytes requested are on the order of TBs; however, *the active bytes*[2] [40] are on average on the order of GBs. As a result, we choose the cache size in the range of 64 GB to 1,024 GB for different traces in our evaluations. For the sake of readability, we present the results using two cache sizes for each trace in the rest of the paper. Similar observations hold for other cache sizes. We conclude that content requests in production systems exhibit extreme *variability* in content popularity, content size, and inter-arrival time distributions. This makes cache management challenging for a CDN server and the need to design an online

[2]A content is said to be active at time t in 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.



(a) Content Popularity (b) Inter-arrival Time

Figure 1: Content popularity and inter-arrival time.

caching algorithm that can automatically adapt to these changes with a robust performance.

Limitations of practical algorithms and bounds. System designers usually take advantage of the aforementioned quirks and dynamics in production systems [16, 59] to build new algorithms that outperform classic ones. In this subsection, we quantify opportunities to improve hit probabilities of existing SOTAs and existing bounds on OPT.

Experimental setup. We implement a cache simulator including a wide range of SOTAs and only report on the seven best-performing algorithms including LRU, LRB [56], AdaptSize [12], etc in this paper (more details in Section 7.5). For comparison, we compute two state-of-the-art offline upper bounds on OPT, i.e., PFOO [11] and Bélády-size, a simple Bélády variant widely used by the community [34, 44, 55]. For each workload, we simulate a set of cache scenarios. For ease of exposition, we only present results for one scenario as shown in Figure 2, where the "best-performing" SOTA is the one with the highest content hit probability among all SOTAs in consideration.

Significant opportunities to improve hit probabilities and upper bounds on OPT. We observe a large gap between existing SOTAs and upper bounds on OPT. Two possible reasons behind this observation: the offline bounds (e.g., Bélády-size) widely used by practitioners offer no optimality guarantees, and there are still achievable gains on hit probability to be explored. We will answer yes to both these possibilities. We show that these upper bounds are in fact far from OPT, and hence give practitioners a false sense of complacency. As a result, existing learning-based algorithms that leverage OPT for decision making may suffer performance losses. Our interpretation is that upper bounds leveraged in these algorithms ignore many practical content features (e.g., content inter-arrival time) exhibited in production workloads, and hence are not tight on OPT for arbitrary traces.

We conclude that there is significant need to realize online caching algorithms that can automatically adapt to changes in content request patterns over time. Furthermore, there is still room to develop tight upper bound on maximum achievable cache hit probability in an online manner to gauge the potential effectiveness of caching algorithms. Developing such a practical online upper bound and leveraging it along with ML for CDN caching require us to overcome many challenges as we describe in Sections 3 and 5.

3 AN ONLINE UPPER BOUND ON OPT

Since existing bounds on OPT are offline, weak or computationally expensive, they are hard to leverage into learning augmented online

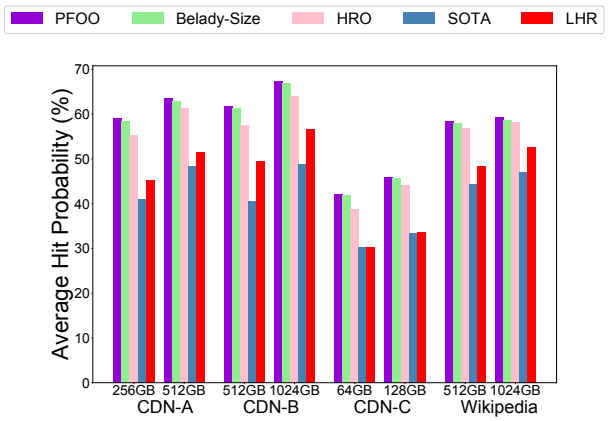


Figure 2: Simulated content hit probabilities of two widely used offline bounds, our online bound HRO, the "best-performing" SOTA, and LHR. There is a large gap of 15%-25% between SOTAs and the tighter offline bound. HRO is much tighter than the other two offline upper bounds. More details in Section 7.5.

caching designs. Recently, Panigraphy et al [53] proposed a hazard rate (HR) based upper bound on OPT that outperforms existing bounds. However, [53] requires the exact content request distribution, which is only available for synthetic workloads, to compute the bound. To tackle this challenge, we generalize results in [53] and develop a new online upper bound, HRO, with limited knowledge of the content request process, which can be easily leveraged into an ML model for online caching algorithm design. In the following, we first present the upper bound in [53] for completeness and then introduce HRO. For ease of readability, we relegate the performance analysis of HRO to Appendix A.1.

3.1 Theoretical Model

We denote successive times when content i is requested as $\{\tau_{ik}, k \in \mathbb{Z}\}$. Then the inter-request time (IRT) between the $(k-1)$-st and k-th requests to content i is defined as $X_{ik} = \tau_{ik} - \tau_{i(k-1)}$, for $k \geq 1$, and $\tau_{i0} = 0$ by convention. We consider $\{X_{ik}\}_{k\geq 1}$ to be a stationary point process with cumulative inter-arrival time distribution functions (c.d.f.) satisfying [6] $F_i(t) = \mathbb{P}(X_{ik} \leq t)$. Denote the corresponding density function as $f_i(t)$. The mean request rate μ_i for content i is then given by $\mu_i = \frac{1}{\mathbb{E}[X_{ik}]} = \frac{1}{\int_0^\infty (1-F_i(t))dt}$. The c.d.f. of the age associated with the inter-arrival time distribution for content i satisfying [6] $\hat{F}_i(t) = \mu_i \int_0^t (1 - F(x))dx$. The hazard rate function $\zeta_i(t)$ associated with $F_i(t)$ is defined as

$$\zeta_i(t) = \frac{f_i(t)}{1 - F_i(t)}, \quad t \in [0, F_i^{-1}(1)], \tag{1}$$

which represents the conditional probability of the occurrence of a content request at time t, given the realization of the request processes over the interval $[0, t)$.

Given the above definitions, a hazard rate based upper bound on OPT can be computed as follows [53]. Upon a request for content i at time t, contents are sorted according to their hazard rates at time t in decreasing order. Then the request is classified as a hit if content i is among the M contents with the largest hazard rates, where M

is the cache size. Such a hazard rate based ordering is analogous to the LFU policy since the hazard rate at time t determines the most popular contents at time t. The above hazard rate function is defined under the assumption that content sizes in the system are equal; however, the content sizes in production system usually vary significantly. To overcome this drawback, the hazard rate function by incorporating content size s_i is redefined as follows

$$\tilde{\zeta}_i(t) = \frac{f_i(t)}{(1 - F_i(t))s_i}, \quad t \in [0, F_i^{-1}(1)]. \tag{2}$$

3.2 From Theory to Practice

To compute the hazard rate and obtain the hazard rate based policy, the c.d.f. $F_i(t)$ is needed. This is straightforward for synthetic workloads; however, the c.d.f. is usually unknown and computationally expensive (e.g., kernel method) to obtain in production systems. We are therefore unlikely to find the exact hazard rate for production workloads. To overcome this challenge, our key insight is that it is sufficient to develop a technique that accurately approximates the hazard rate based upper bound on any trace, which we call HRO.

We consider a sliding time window[3], and only contents within the window are used to compute the IRTs. Given these IRTs, we approximate the request process as a Poisson process to compute an approximate c.d.f., which is used to update the approximated hazard rates in (2). Finally, requests for contents with the top M hazard rates are classified as cache hits. We show analytically (see Appendix A.1) and empirically (Figure 2) that HRO yields a tighter bound on *traces seen in practice* compared to SOTA bounds, and can be efficiently computed in polynomial time. Therefore, we believe these results demonstrate that HRO is both theoretically well-founded and practically useful. More importantly, we will show that it indeed leverages more accurate information about the content request process, and hence can be used to improve online algorithm performance.

4 LEARNING TO LEVERAGE HRO

In order to leverage HRO in the design of a learning augmented online caching algorithm, this section introduces LHR, which is a layered framework moving from detection to admission to eviction. We will discuss the design details of LHR in Section 5.

4.1 LHR Algorithm

To overcome the limitations of SOTAs, we leverage a lightweight supervised learning model (e.g., a XGBM [17] based model) along with HRO to learn a *per-content admission probability*. LHR admits a content if the learned admission probability is greater than an auto-tuned threshold. We then develop a simple *eviction rule* based on the learned probability for content eviction. Finally, to reduce computational costs, we propose a *detection mechanism* to determine when to update the ML model. We call our algorithm LHR, which leverages HRO to *simultaneously learn* content admission and eviction through a *unified* learning framework that enables detection, estimation, and training.

[3]Different from the sliding window in TCP, our sliding windows have no overlaps. For example, for a window of size W, the first window contains the first W-th requests, the second window contains the (W+1)-st to the 2W-th requests and so on.

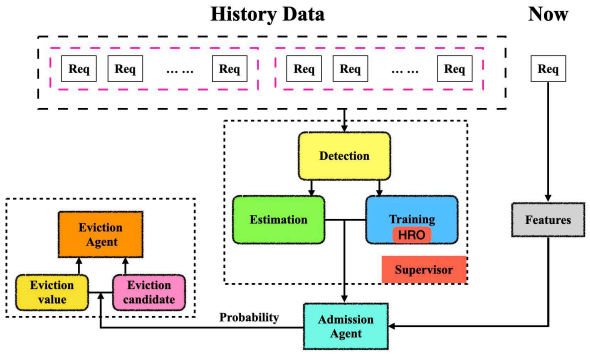


Figure 3: Framework of LHR.

Figure 3 shows the LHR framework, consisting of three components: a supervisor, an admission agent and an eviction agent. We consider a cache of size M and denote the set of cached contents as $\mathcal{M}$. LHR maintains a vector of size M to store the learned content admission probability, p_i of content i in the cache. Denote this vector as $\mathcal{L}$.

Upon a new request for content i, LHR first extracts its feature u_i and computes HRO using request information in the past sliding window. Then LHR outputs an admission probability p_i for content i via a XGBM based model (admission agent). This will be described in detail in Section 5.2.4. LHR compares p_i to an auto-tuned threshold δ for cache admission decision. Upon a cache hit (i.e., content i is already in the cache), we consider two cases: (i) $p_i \geq \delta$: we simply update its admission probability in $\mathcal{L}$; (ii) $p_i < \delta$: we not only update its value in $\mathcal{L}$ but also label it as an *eviction candidate* with value of p_i. Similarly, upon a cache miss, we consider two cases: (iii) $p_i \geq \delta$: content i is admitted, and its value p_i is inserted into $\mathcal{L}$; (iv) $p_i < \delta$: simply discard the request.

When the cache is full, LHR needs to decide what content to evict upon a cache admission (eviction agent). Given admission probability p_i, we develop a simple eviction rule that outputs an eviction probability q_i (eviction value), which is a function of p_i. See Section 5.2.5 for details. LHR evicts an eviction candidate with the smallest eviction value q_i.

4.2 Auto-tuned Threshold

LHR only admits a content whose admission probability is greater than a *threshold*. A smaller threshold moves the hit probability of LHR closer to the admit-all policy [40]. This will be particularly useful for traces where existing learning based algorithms do not perform as well as simple admit-all or admit-nothing models. A larger threshold makes LHR more tractable but may discard popular contents. It is thus important to find a proper threshold. To systematically choose a threshold for LHR, we design a simple *estimation algorithm* that adaptively adjusts the threshold δ based on recent requests. This is described in detail in Section 5.2.3.

4.3 Efficient Training

Even with a simple threshold-based policy for content admission, naively training LHR on arbitrary CDN traces may still be computationally intensive and time consuming. To reduce computational

costs, we first consider *a sliding time window* and use only history data within the window for training. Then a straightforward training process is to update model parameters at every sliding window. However, our key observation is that this is not necessary in production systems as it is time consuming with a high memory overhead. If requests in two consecutive sliding windows exhibit "similar" pattern, there is no need to update the model again. Then a natural question arises: *when should the model be updated?* We pose this problem as an *anomaly detection problem*, where an "anomaly" occurs when request patterns change significantly between two consecutive sliding windows. We propose a simple *detection mechanism* for LHR, which is built on the estimation of request distribution. This will be described in detail in Section 5.2.2.

Key take-aways. LHR has four important properties.

• ***Leveraging HRO:*** LHR leverages HRO through a simple supervised learning framework to make caching decisions in an online manner. This is possible since HRO can be efficiently updated in production systems in polynomial time.

• ***Auto-tuned threshold-based admission policy.*** Rather than using a fixed threshold (e.g., $\delta = 0.5$), we adaptively determine threshold value δ based on the request stream. This not only makes the algorithm adapt to dynamics in production workloads, but also reduces training error and hence improves model accuracy.

• ***Better eviction rule:*** Unlike prior approaches that only use a small set of content features for cache eviction, we develop a new eviction rule based on the learned admission probability p_i and other key content features (e.g., content size). This also improves learning accuracy of our model through making better eviction decisions.

• ***Efficient training via detection:*** Instead of training the ML model after each sliding window, we design a detection mechanism to determine when the model should be trained and updated. This property provides a dramatic reduction of computational cost.

5 DESIGN OF LHR CACHE

This section presents the design details of LHR, which uses machine learning to leverage HRO into online caching design. To make LHR a reality, we have to simultaneously address three previously unsolved problems:

• How to leverage HRO into an online caching design?

• How to design an ML approach that achieves hit probabilities close to those of OPT with small WAN traffic?

• How to make this practical?

Each of these problems pose several challenges, and it additionally requires us to balance their often-competing goals when addressing them simultaneously. We present a general ML framework to leverage HRO for learning content admission and eviction simultaneously as shown in Figure 4. There are two key design issues:

Historical data. To compute HRO and to train the decision model, it is of utmost importance to determine how much past information is needed. More historical data may lead to a more accurate model, but may also significantly increase resource overheads. As a result, we need to consider a tradeoff in production CDN servers due to hardware constraints.

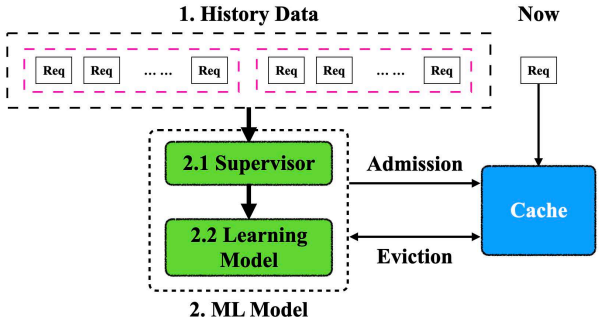


Figure 4: A general ML framework that leverages optimal caching for learning content admission and eviction simultaneously.

Learning framework. This involves how to select and design a supervised ML framework to achieve high hit probabilities. This includes (i) selection of content features; (ii) the design of a "supervisor" that can leverage HRO into cache decisions, determine when the ML model should be updated, as well as the admission and eviction rule; and (iii) an ML model to integrate all into caching decision making.

In the following, we describe the design decisions of LHR for each of these design issues. Our design is generally guided by the practical online upper bound on OPT, i.e., the HRO.

5.1 History Data

LHR records information of contents that have been previously requested. We represent each request as a vector including different content features that will be described in details later. In our LHR framework, history data are used for training the learning model for content admission, as well as for computing HRO.

The admission agent in our LHR framework takes content features as input and outputs an admission probability, which is further used in the eviction process. We consider a sliding window and only use data within the window for training and computing HRO. To reduce computational costs, we do not train the model during every sliding window but instead use a detection mechanism to decide when to retrain.

As a result, a proper window size is important to the performance of LHR. On one hand, the admission model and the eviction rule will not be accurate if the window size is too small. On the other hand, a large window size may increase memory overhead and result in more time required to process operations. In reality, CDN servers need to process millions of requests per second, and the window size can be large given the effective system computing capability. Since content sizes vary significantly in production traces, we consider the size of a window measured in the number of unique bytes seen over a number of requests. We evaluate the impact of the window size on four traces using two cache sizes as shown in Figure 5, where 2× means the unique bytes of content requests in the window is twice the cache size. Given the characteristics of these traces, we mimic real CDN systems with a window of content requests whose unique bytes is 4× of the cache size for all traces considered in this paper to achieve a reasonable tradeoff between hit probability, memory overhead and running time.

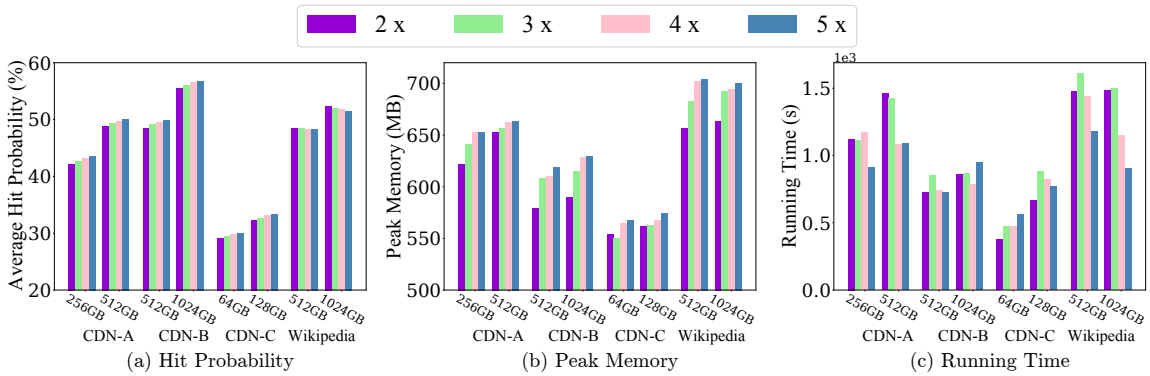


Figure 5: The impact of sliding window size.

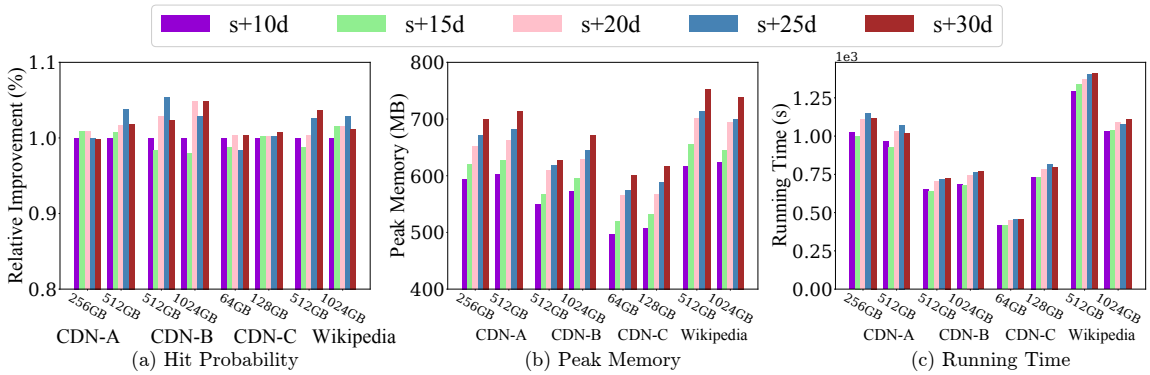


Figure 6: The impact of content features, where 's' and 'd' represent static and dynamic features, respectively. For example, '10d' means IRT_1 to IRT_{10} are included.

Finally, data is needed to start the training process at the beginning of a trace so as to build an initial model. Hence we use the first sliding window as training data, and then execute the caching algorithm from the second sliding window onwards.

5.2 LHR Framework

We describe the main components in the ML framework of LHR in this subsection. For each component, we describe our choice and design, and discuss the corresponding rationale.

5.2.1 ***Content features.*** The rationale for our feature selection is to provide a superset of information used by SOTAs. Different from those heuristic methods that only rely on one or a few features, we consider a set of *time-varying* features along with a set of *static* features.

• *Inter-request times.* Inspired by the design of HRO, we consider inter-request times (IRTs) – the times between consecutive requests to a content – as a time-varying feature. For example, IRT_1 represents the time since the last request, IRT_2 is the time between the content's previous two requests, and so on. We use IRTs as features since they provide useful information used by many SOTAs, including LRU with IRT_1 and LRU-K using IRT_1 to IRT_k.

• *Static Features.* Static features capture additional non-time varying content information such as content size and so on. These features are easy to obtain and consume little memory overhead. Static features also play a role in labeling different contents.

IRTs and static features provide a set of information used by SOTAs. We benefit from them by adding more features during model training. However, adding more features may increase system memory overhead, i.e., there is a tradeoff between adding more information to the model for making better decisions and the system overhead. Therefore, we determine the effectiveness of these features and decide how many IRTs to store. Figure 6 shows the marginal improvements from an accumulation of static features, and 10-30 IRTs, where the hit probability improvement is measured with respect to the case with 10 IRTs. As expected, more features improves hits with diminishing improvements and larger memory overhead and running time. We settle on 20 IRTs because this appears to guarantee good performance of our design across all traces. We denote the features of content i as u_i.

5.2.2 ***Detection mechanism.*** Although the content request process varies over time, it has been observed [5, 14, 30] in production systems that the request process is characterized by a Zipf distribution, where the popularity of the i-th most popular content follows $p_i = A/i^{\alpha}$ with A being the normalization factor and α is the Zipf parameter. If we know the value of α in each sliding window, then we only need to update our model if there is a significant change in

α between two consecutive windows, e.g., $|\alpha_{j+1} - \alpha_j| \geq \epsilon$, where j indicates the j-th time window and ϵ is a constant. This greatly reduces computational and memory overheads. Then we develop a Least-Squared-Method (LSM) based model to estimate the value of α in each sliding window. We first transform the Zipf popularity into $\log p_i = \log A - \alpha \log i$, and then directly use LSM to estimate its intercept $\log A$ and gradient α. This LSM-based method is well known to be lightweight with a complexity of $O(N)$, where N is the number of unique contents in one sliding window. Hence it can be easily implemented in production systems. We will show that our LSM based model is accurate in Section 7.4 as well as additional discussions on parameters in Appendix A.2.

5.2.3 ***Estimation algorithm.*** As observed in Section 2, realistic content requests generated by users in production systems are non-stationary, i.e., time-varying content popularities, variable content size and so on. Consequently, it is inappropriate to use a fixed threshold (e.g., $\delta = 0.5$ has been widely used) to decide whether to admit a content upon a request as discussed in Section 4. Hence we need to adaptively adjust the threshold δ based on recent requests.

Suppose that the threshold used in the current sliding window k is δ_k, and the corresponding hit probability is $h(\delta_k)$. We set a candidate threshold set Δ_k for sliding window k to be $\Delta_k = \{0, 0.5, \delta_k - 0.1, \delta_k + 0.1\}$. Note that we use a step size of 0.1 to update the threshold. It can be any value in $[0, 0.1, \cdots, 0.9, 1]$. From our experiments, we observe that our choice of δ_k provides good performance in terms of hit probability and training time. We compute the hit probability during the current sliding window k using the candidate thresholds in Δ_k. Suppose that $\hat{\delta}_k \in \Delta_k$ returns the largest hit probability $h(\hat{\delta}_k)$. To make the estimation robust, we use the threshold $\hat{\delta}_k$ in the next sliding window $k+1$ if and only if $h(\hat{\delta}_k) > h(\delta_k)$, and $|h(\hat{\delta}_k) - h(\delta_k)| > \beta$, where β is a constant. Otherwise, we do not update the threshold. The above two update rules help the learning algorithm avoid unnecessary updates and hence reduce training time and computational costs.

Furthermore, we observe that there is no need to use all requests in the current sliding window in the above estimation process. For instance, using only half of requests in each sliding window still provides a good performance (See Section 7).

5.2.4 ***Learning model.*** LHR uses an XGBoosting Machine (XGBM) [17] based model to learn content admission probabilities. As mentioned above, the input consists of 20 IRTs and static features, and the output is the learned admission probability for one content. To be more specific, we assume that there are N requests in the current sliding window t with features $\{u_i\}_{i=1}^N$. Denote the decisions $\{y_i\}_{i=1}^N$ on these requests from HRO, i.e., "optimal caching decision", which are used as content labels. Consider an XGBM based model with parameter $w(t-1)$ and represent the output of the model as $A(w(t-1), \cdot)$, i.e., the admission probability for each content. Then we train the model and update its parameter by leveraging HRO through solving the following optimization problem

$$\min_w \sum_{i=1}^{N} (A(w(t-1), u_i) - y_i)^2,$$

where we consider the mean squared error since it achieves the best performance in our experiments compared to other loss functions that we explored. The learned admission probability will be used in the four cases described in Section 4, i.e., compare with a threshold to determine whether to admit or not. Upon admission, then update the admission probability for the content in $\mathcal{L}$. This admission probability is also used in the eviction process, as illustrated in Figure 3.

Algorithm 1 LHR

```
Initialize the admission threshold as δ_0 = 0.5
for sliding windows k = 1, 2, ..., do
  Extract features of each requested content i and outputs an
  admission probability p_i (Section 5.2.4);
  if p_i ≥ δ_k then
    Cache content i and evict content j with q_j (Section 5.2.5);
  end if
  Estimate α_k using LSM (Section 5.2.2)
  if |α_k − α_{k−1}| ≥ ε then
    Update δ_{k+1} (Section 5.2.3)
    Re-train the learning model (Section 5.2.4)
  end if
end for
```

5.2.5 ***Eviction rule.*** A straightforward eviction rule is to use the learned probability p_i and evict the cached content i with the smallest p_i. This eviction rule is simple and the resulting LHR outperforms most SOTAs in some but not all production traces considered in the paper, in particular for traces whose content sizes vary significantly and exhibit stark variation in content inter-arrival times. Therefore, instead of just using one feature (e.g., p_i) for eviction, we design a new and simple eviction rule that includes a class of content features such as IRTs and content size.

To be more specific, we consider the output of p_i from the XGBM model and content features for content eviction. When a content eviction is needed, we sort the contents in the cache according to $q_i = \frac{p_i}{s_i} \times \frac{1}{\text{IRT}_1}$, and evict the content with the smallest q_i. Such an eviction rule not only captures content popularity, but also content size and the approximate number of requests for content i. For example, if the cached content has not been requested for a long time (i.e., a large IRT_1 value), then it should be evicted with a higher probability (i.e., a smaller q_i).

5.3 Putting This All Together

Combining all of the above components results in the design shown in Figure 3. LHR learns to cache using a sliding window of N requests through the content admission and eviction models as described above. The detection mechanism leads to a dramatic decrease in the computation overhead while HRO, the estimation algorithm and eviction rule all improve the learning accuracy (i.e., a higher hit probability). We summarize LHR in Algorithm 1.

6 IMPLEMENTATION

We have implemented a LHR prototype within an Apache Traffic Server (ATS) [3] and the Caffeine [47], respectively. An LHR cache simulator[4] has also been implemented for the sake of comparison

[4]The cache simulator along with all implementations in this paper are available at https://github.com/GYan58/lhr-work.

with a wide range of state-of-the-art caching algorithms. The implementations are written in C++, Java and Python3, respectively. For ease of presentation and due to space constraints, we relegate the implementation of Caffeine and the corresponding results to Appendix A.3.

6.1 LHR Prototype

ATS is a multi-threaded and event-based CDN caching server with a space-efficient in-memory lookup data structure as an index to the cache. A typical ATS configuration consists of a disk/SSD cache and a memory cache. To achieve high performance, ATS is accessed using asynchronous I/O.

Upon a new request, ATS implements the following steps:

• Step 1. Based on the URL, it looks up the caches to check whether the corresponding content is available.

• Step 2. If the requested content is already in the caches, it will check if the content is fresh[5]. (a) If the content is fresh, it directly sends the content to the user; (b) If the content is stale, it communicates with the origin server to revalidate the content. If it is still fresh, it directly sends it to user; otherwise, it re-fetches the content into the caches and delivers it to the user at the same time.

• Step 3. If the requested content is not in the caches. ATS will fetch the content from the origin server. Then it directly sends the content to the user and admits the content into the cache at the same time.

We implement LHR on top of ATS. To do so, we replace the lookup data structures for the ATS cache with the LHR architecture described in Section 4. The content admission and look-up processes are implemented asynchronously. The decision model is used in these two processes to make admission decisions and to update the corresponding values that are used in the eviction process. In particular, the eviction process is run by scheduling cache admissions in a lock-free queue. It implements eviction rule to select one eviction candidate when the cache is full. However, we do not have access to the flash abstraction layer (e.g., RIPQ [60]). Hence we emulate this layer, reading offsets randomly and writing sequentially to the disk. The memory cache is typically small which has little impact on hit probability [12], we keep this part of ATS unchanged. We implement the framework by only modifying about 100 lines of codes in ATS. The LHR framework library contains about 1,300 lines of codes.

6.2 LHR Simulator

We implement an LHR simulator that includes a wide range of classic and learning-based algorithms, as well as several existing upper bounds on OPT. We only report results for the seven best-performing algorithms including LRB [56], Hawkeye [36], LRU, LRU-4 [51], LFU-DA [4, 54], AdaptSize [12] and B-LRU[6]. Finally, our implementation benefits from existing caching simulators in the literature such as libCacheSim [45] and LRB simulators [56].

[5]The server could cache contents which were admitted into the cache long time ago (the time to judge whether a content is stale can be configured on ATS). When a user again requests these contents, ATS needs to communicate with the origin server to make sure they have not been changed or updated.

[6]Bloom Filter LRU (B-LRU) uses Bloom filter to prevent one-hit content from being admitted.

7 EVALUATION

In this section, we evaluate our LHR prototype. We also use simulation to compare LHR to a wide range of SOTAs using four production traces. Our results address the following questions:

• What is the benefit of using our LHR prototype compared to existing CDN production systems in terms of content hit probability, WAN traffic, and implementation overhead (Section 7.2)?

• What is the performance of LHR cache, in terms of content hit probability, WAN traffic, latency and throughput, compared to SOTAs on a wide range of production CDN traces under various settings (e.g., cache size) (Section 7.3)?

• How do the estimation algorithm and detection mechanism impact the performance of LHR (Section 7.4)?

• What is the gap between LHR and upper bounds on OPT (Section 7.5)?

• How well can LHR adapt to the workload changes (Section 7.6)?

For the sake of readability, some experimental results are relegated to Appendix A.3.

7.1 Methodology

Algorithm settings. Our evaluation uses the following default values. As discussed in Section 5.1, each sample is generated using a sliding window of content requests whose unique bytes is 4× of the cache size. We set $\beta = 0.2\%$ in the estimation algorithm. Their rationales are discussed earlier (see Sections 5.1 and 5.2.3).

Baselines. As LHR leverages HRO to learn both content admission and eviction, we compare it to a wide range of SOTAs. To improve readability, we only show the seven best-performing algorithms in the following figures.

Overhead. The metadata overhead varies across different algorithms. We deduct the corresponding overheads from the cache sizes in all experiments for all SOTAs for the sake of fairness. For example, for an algorithm with 2GB overhead on a 512GB cache, only 510GB is used for caching.

Performance evaluation. We evaluate the performance of these algorithms using the four production workloads described in Section 2 with different cache sizes, which are chosen based on the active bytes. All results are generated by running on Ubuntu 18.04 with an Intel(R) Core(TM) i5-10400HQ processor and a 8GB RAM.

7.2 LHR Prototype vs. ATS

We first compare our LHR prototype to the unmodified ATS with respect to hit probability, WAN traffic and implementation overhead as shown in Figure 7 and Table 2.

Hit probabilities. Figure 7 compares hit probabilities of LHR and an unmodified ATS using CDN-A, CDN-B, CDN-C and Wikipedia traces with a cache of 512GB, 1,024GB, 128GB and 1,024GB, respectively. LHR achieves a higher overall hit probability than ATS. Furthermore, LHR quickly outperforms ATS after obtaining five sliding windows of data, and LHR continues to improve its performance as it obtains more data.

Implementation overhead. We then compare the implementation overhead of our LHR prototype against the unmodified ATS. We measure the throughput, CPU and memory utility under the "max" experiments, as shown in Table 2. We observe that LHR has no measurable throughput overhead but the peak CPU utilization

		CDN-A		CDN-B		CDN-C		Wikipedia	
Metric	Experiment	LHR	ATS	LHR	ATS	LHR	ATS	LHR	ATS
Throughput (Gbps)	max	6.57	6.46	7.18	6.66	6.05	5.90	7.37	6.91
Peak CPU (%)	max	22.5	3.7	24.1	4.8	22.5	4.1	23.9	4.2
Peak Mem (GB)	max	2.8	2.2	2.5	2.3	2.1	1.7	1.9	1.7
P90 Latency (ms)	normal	224	245	241	253	274	276	232	256
P99 Latency (ms)	normal	304	305	324	325	322	322	305	314
Overall Latency (ms)	average	104	118	119	133	163	168	91	117
Traffic (Gbps)	average	1.41	1.47	2.52	2.69	3.47	3.61	2.12	2.37
Content Hit (%)	normal	48.92	41.68	49.92	39.04	27.46	25.96	44.96	36.64

Table 2: Resource usage for LHR and ATS in max (throughput-bound) and normal (production-speed) experiments.

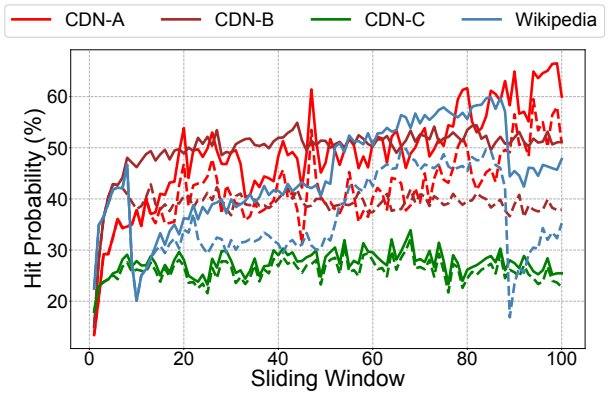


Figure 7: The content hit probabilities of LHR (solid lines) and unmodified ATS (dashed lines).

increases to 22.5% from 3.7% for ATS under CDN-A, 24.1% from 4.8% for ATS under CDN-B, 22.5% from 4.1% for ATS under CDN-C and 23.9% from 4.2% for ATS under Wikipedia. However, we note that most production servers, even at their busiest mode, have sufficient CPU headroom.

We replay our traces using their original timestamps and measure the corresponding number of content hits. We label these "normal" experiments as shown in Table 2. It is clear that LHR significantly increases content hit probability by 2%-11% over ATS. This improvement allows LHR to improve the 90-th percentile latency (P90 latency) by 2%-10%, the 99-th percentile latency (P99 latency) by 1%-5%, and overall average latency by 4%-23% compared to ATS. This further translates to a traffic reduction of 5%-12% over ATS. Finally, we measure peak memory overhead for all traces and cache sizes, we observe that LHR uses 0.2%-1.6% of the cache size to store metadata. As we show later, such a small loss in available caching space is more than offset by LHR's significant improvements in hit probability and WAN traffic.

We conclude from these experiments that LHR is a practical design for today's CDNs and can be easily implemented in existing production CDN servers with modest resource overhead.

7.3 LHR vs. SOTAs

We further compare LHR to a large number of SOTAs using four production traces across a wide range of cache sizes.

Hit probabilities and WAN traffic. Figure 8 shows the average content hit probability and the average WAN traffic of each candidate algorithm with different cache sizes. LHR *consistently* outperforms the best SOTAs (which varies in different settings). Overall, LHR improves hit probability by 2%-9% on average besides on the CDN-C trace, and reduces traffic by 5%-15% on average. Note that the improvement in the hit probability in CDN-C trace is not significant. Our interpretation is that most contents in CDN-C are only requested once, which leaves less room for LHR to learn the features, e.g., inter-request times.

Note that LHR improves hit probabilities and reduces traffic across all traces whereas none of the SOTAs does so. For example, Hawkeye is the best in CDN-B but among the worst with CDN-C. LFU-DA does well with CDN-A and CDN-C but not well with Wikipedia with 512 GB cache. Furthermore, LHR simultaneously achieves the largest hit probability and the least traffic while the large hit probabilities of SOTAs often do not translate into low traffic rates. For example, Hawkeye has one of the highest hit probabilities among SOTAs on CDN-A with 512 GB cache but exhibits one of the most traffic. These results indicate that existing SOTAs including recently developed learning-augmented caching algorithms perform well on certain workloads but poorly on others.

Memory overhead and running time. Figure 9 shows the memory overhead for training the learning-based caching algorithms. We observe that LHR requires less memory than LRB, which requires more space to store content features, but more memory than Hawkeye. Note that the amounts of memory required by all of these algorithms are much smaller than the cache size. We further characterize the running times of learning augmented caching algorithms, which include training time, and the time to process content admission and eviction. From Figure 9, we observe that LHR dramatically reduces running time compared to LRB. The reason is that LRB needs to use the ML model to predict the next request time for all cached contents upon each eviction. Further, it requires updating corresponding features for all contents before making the prediction. Neither is required in the design of LHR.

Latency and throughput. We further characterize latency (e.g., query time) and throughput of LHR on production traces. We assume the trace-based simulation is run in an ideal environment where (a) network transmission rate is 8 Gbps, i.e., each content can be transmitted at the rate of 8 Gbps. (b) latency is mainly affected by two factors: distance and content size. The larger the size, the slower the user receiving the complete content. We also

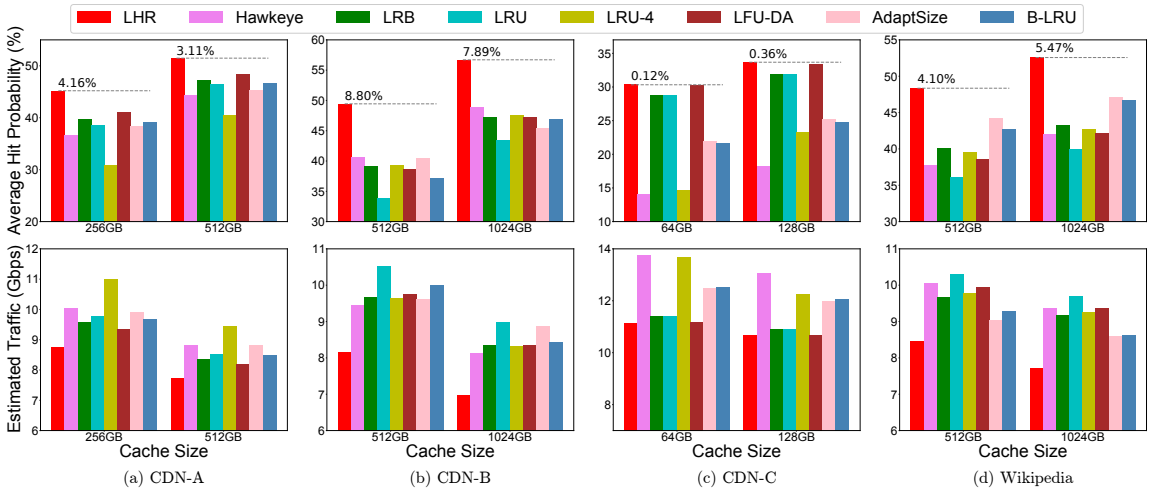


Figure 8: Comparison of average content hit probabilities and WAN traffic of LHR and SOTAs using production traces. It is clear that LHR consistently outperforms all candidate SOTAs in all traces with difference cache sizes.

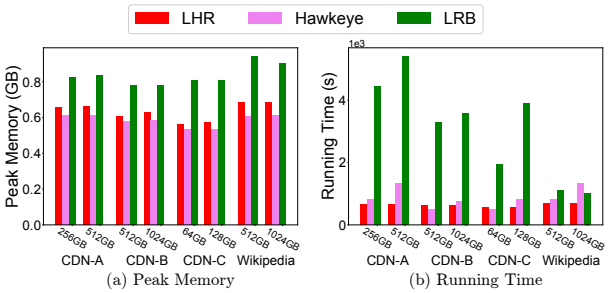


Figure 9: Comparison of peak memory and running time between LHR and SOTAs.

Metrics	Traces	LHR	Hawkeye	LRB	LRU
Latency	CDN-A	52.6	57.9	55.0	55.9
	CDN-B	50.2	58.0	59.6	63.3
	CDN-C	67.8	83.0	69.3	69.3
	Wikipedia	50.3	60.1	58.9	62.1
Throughput	CDN-A	8.11	7.24	7.70	7.56
	CDN-B	9.50	8.23	7.99	7.44
	CDN-C	5.45	3.06	5.22	5.22
	Wikipedia	8.47	6.91	7.09	6.58

Table 3: Estimated average latency (ms) and throughput (Gbps) for LHR, Hawkeye, LRB and LRU on a cache of 512GB, 1,024GB, 128GB, and 1,024GB for CDN-A, CDN-B, CDN-C, and Wikipedia, respectively.

take the running time of the ML model into account. The average latency and throughput of LRU (the default algorithm in ATS), LRB, Hawkeye, and LHR are given in Table 3. We observe that LHR consistently improves latency and throughput over SOTAs across all traces.

7.4 LHR: Estimation and Detection

To further understand where the improvements of LHR come from in Sections 7.2 and 7.3, we focus on understanding how our proposed techniques in LHR impact its performance. In other words, we validate the detection mechanism (Section 5.2.2) and the estimation algorithm (Section 5.2.3) using production traces. To that end, we consider two variants of LHR: D-LHR which is LHR using a fixed threshold $\delta = 0.5$ for content eviction; and N-LHR which is D-LHR without the detection mechanism. The average content hit probabilities, peak memory and running time of LHR, D-LHR and N-LHR are shown in Figure 10.

7.4.1 ***Impact of an auto-tuned threshold.*** We first validate the importance of including an estimation algorithm on improving learning accuracy by comparing LHR to D-LHR. From Figure 10(a), it is obvious that LHR significantly outperforms D-LHR in CDN-C with various cache sizes. Content hit probabilities are improved by 160% and 143%, respectively. While in other traces, LHR achieves similar performance as D-LHR. We find that the threshold value derived from our estimation algorithm is roughly 0.5 across all of the sliding windows in these traces, and hence D-LHR is almost the same as LHR.

From Figure 10(b), we observe that the improvement in content hit probability comes at the cost of a marginal increase in training time and a memory overhead of 1%-8%. However, the latter is much smaller than the available cache size. For a given cache size, although LHR might incur a little more memory overhead (leaving less cache space for contents), it achieves a much better hit probability compared to D-LHR.

7.4.2 ***Impact of the detection mechanism.*** Next we validate the importance of the detection mechanism in reducing computational cost (e.g., training time) by comparing D-LHR and N-LHR. From Figure 10(c), we observe that the detection mechanism in D-LHR consistently reduces training times over N-LHR by 15%-40%. More importantly, we observe from Figure 10(b) that this training time reduction causes no additional memory overhead, which further validates the lightweight nature of our detection mechanism.

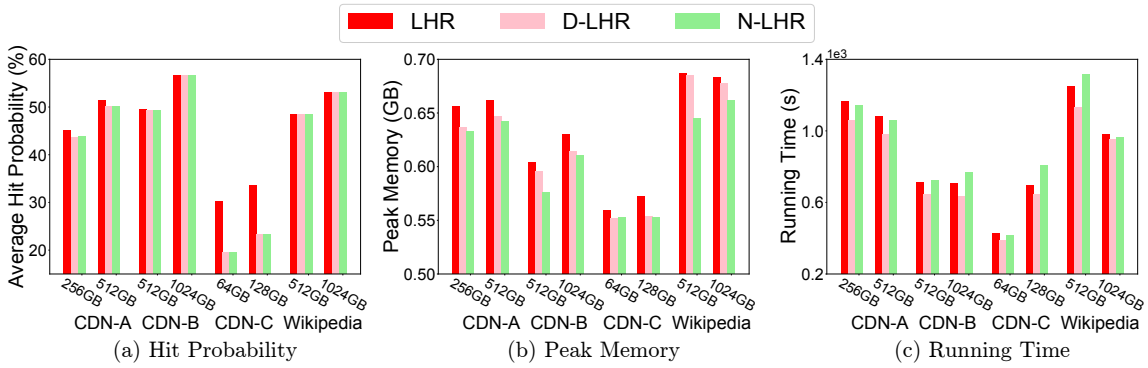


Figure 10: Comparison of average hit probabilities, peak memory and running time of LHR, D-LHR and N-LHR.

Furthermore, since the detection mechanism reduces the frequency at which the learning model is updated, it may suffer an accuracy loss. However, from Figure 10(a), we observe that the detection mechanism actually does not generally decease the hit probability.

7.4.3 ***Joint impact.*** Finally we characterize their joint impact on the design of LHR by comparing LHR and N-LHR. Again, from Figure 10(a), LHR consistently achieves a higher hit probability than N-LHR. As discussed above, we believe this benefits from the auto-tuned threshold, i.e., the estimation algorithm with a marginal increase in memory overhead. Finally, the training time has been improved on the CDN-B, CDC-C and Wikipedia datasets, i.e., LHR decreases the training time by 8%-20% on average compared to N-LHR. However, LHR takes a bit longer to train on CDN-A. Our interpretation is that there is a tradeoff between estimation and detection. For example, in CDN-A, the estimation of Zipf parameter α dominates detection overhead, while other traces, the detection mechanism benefits the design by reducing training time.

We conclude that our proposed detection and estimation algorithms are beneficial to LHR providing an improved hit probability at moderate memory overhead and training time in general. More importantly, it provides greater flexibility to the system. Together with the discussions in Section 7.3, this suggests that improvements of LHR come from the "accurate information" leverage from HRO, the detection mechanism and the estimation algorithm.

7.5 LHR vs. OPT

Given the above discussions, it is clear that LHR consistently outperforms SOTAs on considered traces. Now we compare LHR with several offline upper bounds that are widely used by the system communities and our online upper bound HRO. As shown in Figure 2 in Section 2, we observe that HRO provides a tighter online upper bound on OPT compared to existing offline upper bounds, and LHR indeed reduces the gap between SOTAs and the upper bounds on OPT of up to 30%. Furthermore, we observe that LHR is closer to HRO and other offline upper bounds. The remaining gap between LHR and HRO is mainly due to the errors in our model. We will explore the improvement in the learning model as a promising direction for future work.

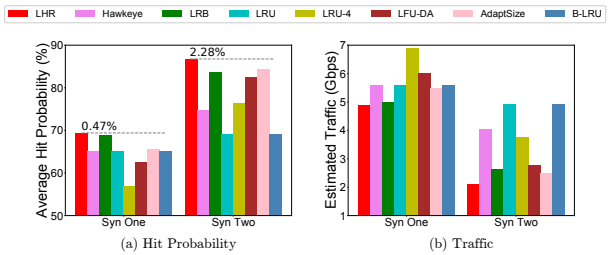


Figure 11: Responsiveness of LHR.

7.6 Responsiveness of LHR

Finally, we evaluate the responsiveness of LHR to workload changes to understand how well LHR and existing algorithms are able to track the arrival process. A simple and widely used model that possesses the desired changeability property is a Markov modulated request process. Under this model, each state corresponds to one content request process. Requests are drawn from the request distribution corresponding to the current state. In particular, we consider two cases named "Syn One" and "Syn Two".

Under "Syn One", we define a Markov chain $\{Y_l\}_{l\geq 0}$ with state space $\{0, 1\}$, each corresponding to one content request process (i.e., one content popularity distribution). We say $Y_l = 0$ if the system is at state 0 and the popularity follows one Zipf distribution, and $Y_l = 1$ if the system is at state 1 and follows another Zipf distribution. Without loss of generality, we consider two Zipf distributions over N distinct contents, one with increasing order of ranking, i.e., $p_i = A/i^{\alpha}$, the other with decreasing order of ranking, i.e., $p_j = A/(N - j + 1)^{\alpha}$, where $i, j \in \{1, \cdots, N\}$ and A is the normalization factor. We assume that if the Markov chain is in a particular state, a fixed number of requests r will be drawn according to the corresponding distribution, and a state transition occurs.

Under "Syn Two", we consider a Markov chain with state space $\{0, 1, 2\}$. The Zipf distribution corresponds to each state with an increasing order of ranking over contents, i.e., $p_i = A/i^{\alpha_l}$ but with α_l depending on the state. In particular, we consider $\alpha_0 = 0.7$, $\alpha_1 = 0.9$ and $\alpha_2 = 1.1$. The Markov chain starts from state 0, and then transitions to state 1, state 2, state 1 and back to state 0. In any particular state, a fixed number of requests r will be drawn according to the corresponding distribution.

We consider 1 million requests over $N = 1,000$ distinct contents and $r = 200,000$. The performance of LHR and existing algorithms is presented in Figure 11. We observe that the best performing SOTA under "Syn One" is LRB and under "Syn Two" is AdaptSize, while LHR consistently outperforms existing algorithms in both hit probability and traffic, i.e., LHR is adaptive to the changes in workloads.

8 RELATED WORK

Optimal caching and upper bound on hit probability. The optimal algorithm for equal sized contents is the Bélády [9], which is offline since it uses exact knowledge of future content requests. It has been widely used by the systems community as an upper bound on hit probability. In addition, LFU achieves the maximum hit probability when requests for equal sized contents follow the Independent Reference Model (IRM). Computing the optimal hit probability for variable contents is known to be NP-hard [19].

Results on bounding OPT has been proposed, for example, Bélády variants (e.g., Bélády-Size) are widely used as an upper bound on OPT. The few known results with variable content sizes include InfiniteCap [2], Flow-based offline optimal (FOO) and Practical FOO (PFOO) [11]. But, all these bounds are offline, i.e., they assume exact knowledge of future requests. Instead, we develop a practical online upper bound based on the recently proposed hazard rate based upper bound [53] that can be computed for both equal and variable sized contents in polynomial time, and hence can be easily leveraged into online caching algorithm design.

Conventional caching algorithms. Many previous works have focused on improving caching hit probabilities. We classify them by admission or eviction. The widely used admission algorithms include AdaptSize [12], TinyLFU [25] and SecondHit [46], and among others where static features such as content sizes are used for admission [1, 2, 15, 26, 57]. A large number of works proposed eviction algorithms from classic Least Recently Used (LRU) [22], RANDOM, FIFO, to more sophisticated ones that are more difficult to implement in practice, e.g., LRU-K [51], LFU-DA [4, 54], GDSF [18], A-LRU [43], ARC [48], CAR [7] and among others, where recency, frequency or their combinations are usually used for eviction decision [8, 13, 27, 33, 34, 37–39, 42].

Learning augmented caching algorithms. Recently, ML based caching algorithms have been proposed. On the one hand, some focus on learning content popularities for *content eviction* via deep neural networks (DNNs), e.g., DeepCache [49], FNN-Cache [29], PopCache [58] and PA-Cache [28], or by approximating or imitating offline optimal Bélády for content eviction, e.g., LFO [10], LRB [56] and Hawkeye [36] which we use them as SOTAs for comparisons. Though Hawkeye was designed for hardware cache, its idea of applying Bélády to history data with prefetching can be implemented in CDNs. Note that LFO also learns from heuristic OPT but it performs even worse than some conventional algorithms on production traces and hence are not included in the top seven algorithms presented in Section 7. On the other hand, some algorithms learn to decide whether or not to admit a content upon a request (i.e., *content admission*) via reinforcement learning (RL), e.g., RL-Cache [40], CACA [32], RL-Bélády [63] and among others [41, 61, 65].

There are mainly three key limitations of these ML based algorithms. (1) Non-robust performance, i.e., good performance for some access systems and poor for others. On the one hand, DNNs require the entire training dataset to be available for learning a fine-tuned but might outdated prediction model with high computational complexity, which makes it difficult to adopt DNNs based methods at production CDNs. On the other hand, RL-based caching has been shown to perform suboptimally compared to simple heuristics; however, rewards (cache hits) manifest with large delays, which prevents timely feedback to the learning algorithm and introduces notable complexity. (2) Still a big gap in hit rates between heuristic algorithms and the offline optimum [56]. (3) 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 propose LHR using a unified model to learn content admission and eviction simultaneously that can consistently outperform all candidate SOTAs with improved hit probabilities and reduced WAN traffic.

9 CONCLUSION

In this paper, we designed, implemented and evaluated LHR. We showed that LHR is a practical ML-based CDN cache design that consistently outperforms state of the arts over four production CDN traces with both an increase in content hit probability and a reduction in WAN traffic. To bridge the gap with optimal caching (OPT), we proposed HRO and then leveraged HRO to learn the content admission probability, which is used for both content admission and eviction. To improve learning accuracy and reduce computational costs of ML caching, we proposed a simple estimation algorithm, a new eviction rule and a lightweight detection mechanism. We showed that LHR's implementation is practical and deployable in today's CDN servers.

ACKNOWLEDGEMENTS

We would like to thank our anonymous reviewers for their valuable feedback. We would like to thank our shepherd Georgios Smaragdakis for his insightful comments that improved the quality of the paper immensely. This work was supported in part by NSF grants CRII-CNS-2104880 and CNS-1617437. This work was also supported in part by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Solar Energy Technologies Office Award Number DE-EE0009341, and the U.S. ARL and the U.K. MoD under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the NSF, ARL, the U.S. Government, MoD or the U.K. Government.

REFERENCES

[1] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. 1996. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. of ACM SIGCOMM*.
[2] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A Fox. 1995. *Caching Proxies: Limitations and Potentials.* Technical Report. Virginia Polytechnic Institute & State University.
[3] Apache Traffic Server. 2020. https://trafficserver.apache.org/.

[4] 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.

[5] Martin F Arlitt and Carey L Williamson. 1997. Internet Web Servers: Workload Characterization and Performance Implications. *IEEE/ACM Transactions on Networking* 5, 5 (1997), 631–645.

[6] F. Baccelli and P. Brémaud. 2013. *Elements of Queueing Theory: Palm Martingale Calculus and Stochastic Recurrences*. Vol. 26. Springer Science & Business Media.

[7] Sorav Bansal and Dharmendra S Modha. 2004. CAR: Clock with Adaptive Replacement. In *Proc. of USENIX FAST*.

[8] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. 2018. LHD: Improving cache hit rate by maximizing hit density. In *Proc. of USENIX NSDI*.

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

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

[11] Daniel S Berger, Nathan Beckmann, and Mor Harchol-Balter. 2018. Practical Bounds on Optimal Caching with Variable Object Sizes. *Proceedings of the ACM on Measurement and Analysis of Computing Systems* 2, 2 (2018), 1–38.

[12] 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*.

[13] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. 2017. Hyperbolic Caching: Flexible Caching for Web Applications. In *Proc. of USENIX ATC*.

[14] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web Caching and Zipf-like Distributions: Evidence and Implications. In *Proc. of IEEE INFOCOM*.

[15] Pei Cao and Sandy Irani. 1997. Cost-Aware WWW Proxy Caching Algorithms. In *USENIX symposium on internet technologies and systems*, Vol. 12. 193–206.

[16] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. 2015. End-User Mapping: Next Generation Request Routing for Content Delivery. *Proc. of ACM SIGCOMM* (2015).

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

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

[19] Marek Chrobak, Gerhard J Woeginger, Kazuhisa Makino, and Haifeng Xu. 2012. Caching is Hard—Even in the Fault Model. *Algorithmica* 63, 4 (2012), 781–794.

[20] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic Cloud Caching. In *Proc. of USENIX HotCloud*.

[21] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2016. Cliffhanger: Scaling Performance Cliffs in Web Memory Caches. In *Proc. of USENIX NSDI*.

[22] Edward Grady Coffman and Peter J Denning. 1973. *Operating Systems Theory*. Prentice-Hall Englewood Cliffs, NJ.

[23] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. 2018. Adaptive Software Cache Management. In *Proc. of ACM Middleware*.

[24] Gil Einziger, Ohad Eytan, Roy Friedman, and Benjamin Manes. 2021. Lightweight Robust Size Aware Cache Management. *arXiv preprint arXiv:2105.08770* (2021).

[25] 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.

[26] Bin Fan, David G Andersen, and Michael Kaminsky. 2013. MemC3: Compact and Concurrent MemCache with Dumber Caching and Smarter Hashing. In *Proc. of USENIX NSDI*.

[27] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. 2011. Small Cache, Big Effect: Provable Load Balancing for Randomly Partitioned Cluster Services. In *Proc. of ACM SoCC*.

[28] Qilin Fan, Xiuhua Li, Jian Li, Qiang He, Kai Wang, and Junhao Wen. 2021. PA-Cache: Evolving Learning-Based Popularity-Aware Content Caching in Edge Networks. *IEEE Transactions on Network and Service Management* 18, 2 (2021), 1746–1757.

[29] 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.

[30] Phillipa Gill, Martin Arlitt, Zongpeng Li, and Anirban Mahanti. 2007. Youtube Traffic Characterization: A View from the Edge. In *Proc. of ACM IMC*.

[31] Michael T Goodrich and Roberto Tamassia. 2006. *Algorithm Design: Foundation, Analysis and Internet Examples*. John Wiley & Sons.

[32] 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*.

[33] Xiameng Hu, Xiaolin Wang, Yechen Li, Lan Zhou, Yingwei Luo, Chen Ding, Song Jiang, and Zhenlin Wang. 2015. LAMA: Optimized Locality-aware Memory Allocation for Key-value Cache. In *Proc. of USENIX ATC*.

[34] Qi Huang, Ken Birman, Robbert Van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. 2013. An Analysis of Facebook Photo Caching. In *Proc. of ACM SOSP*.

[35] Hiroaki Ishii, Toshihide Ibaraki, and Hisashi Mine. 1977. Fractional Knapsack Problems. *Mathematical Programming* 13, 1 (1977), 255–271.

[36] Akanksha Jain and Calvin Lin. 2016. Back to the Future: Leveraging Belady's Algorithm for Improved Cache Replacement. In *Proc. of ACM/IEEE ISCA*.

[37] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proc. of USENIX ATC*.

[38] Song Jiang and Xiaodong Zhang. 2002. LIRS: An Efficient Low Inter-Reference Recency Set Replacement Policy to Improve Buffer Cache Performance. *Proc. of ACM SIGMETRICS* (2002).

[39] Shudong Jin and Azer Bestavros. 2000. Popularity-Aware Greedy Dual-Size Web Proxy Caching Algorithms. In *Proc. of IEEE ICDCS*.

[40] Vadim Kirilin, Aditya Sundarrajan, Sergey Gorinsky, and Ramesh K Sitaraman. 2020. RL-Cache: Learning-based Cache Admission for Content Delivery. *IEEE Journal on Selected Areas in Communications* 38, 10 (2020), 2372–2385.

[41] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. 2017. Harvesting Randomness to Optimize Distributed Systems. In *Proc. of ACM HotNets*.

[42] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 1999. On the Existence of A Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies. In *Proc. of ACM SIGMETRICS*.

[43] 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.

[44] Suoheng Li, Jie Xu, Mihaela Van Der Schaar, and Weiping Li. 2016. Popularity-Driven Content Caching. In *Proc. of IEEE INFOCOM*.

[45] limCacheSim. [n.d.]. https://github.com/1a1a11a/libCacheSim.

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

[47] Ben Manes. 2016. Caffeine: A High Performance Caching Library for Java 8. https://github.com/ben-manes/caffeine.

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

[49] 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*.

[50] Rajesh Nishtala, Hans Fugal, Steven Grimm, Marc Kwiatkowski, Herman Lee, Harry C Li, Ryan McElroy, Mike Paleczny, Daniel Peek, Paul Saab, et al. 2013. Scaling Memcache at Facebook. In *Proc. of USENIX NSDI*.

[51] 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.

[52] Nitish K Panigrahy, Jian Li, and Don Towsley. 2018. Network Cache Design under Stationary Requests: Exact Analysis and Poisson Approximation. In *Proc. of IEEE MASCOTS*.

[53] Nitish K. Panigrahy, Philippe Nain, Giovanni Neglia, and Don Towsley. 2020. A New Upper Bound on Cache Hit Probability for Non-Anticipative Caching Policies. In *Proc. of IFIP Performance*.

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

[55] Samta Shukla and Alhussein A Abouzeid. 2016. Optimal Device-Aware Caching. *IEEE Transactions on Mobile Computing* 16, 7 (2016), 1994–2007.

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

[57] David Starobinski and David Tse. 2001. Probabilistic Methods for Web Caching. *Performance evaluation* 46, 2-3 (2001), 125–137.

[58] 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 LCN*.

[59] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. 2017. Footprint Descriptors: Theory and Practice of Cache Provisioning in A Global CDN. In *Proc. of ACM CoNEXT*.

[60] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced Photo Caching on Flash for Facebook. In *Proc. of USENIX FAST*.

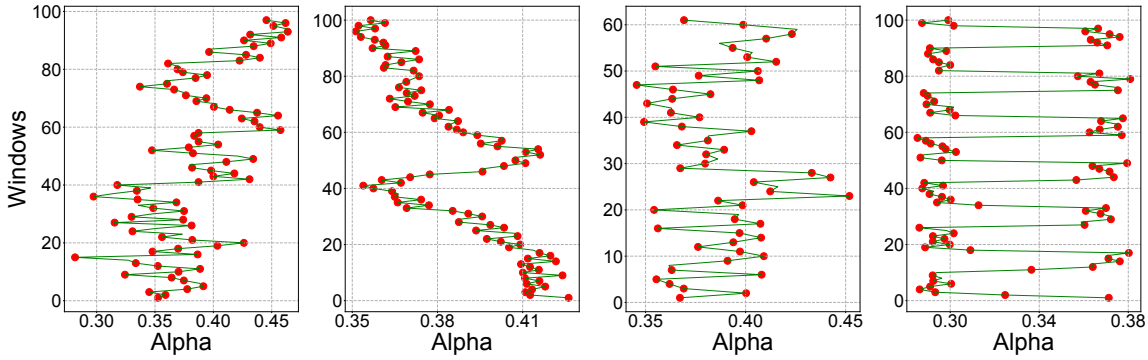
[61] Haonan Wang, Hao He, Mohammad Alizadeh, and Hongzi Mao. 2019. Learning Caching Policies with Subsampling. In *NeurIPS Machine Learning for Systems Workshop*.

[62] G.V. Weinberg. 2016. Kullback Leibler Divergence and the Pareto Exponential Approximation. *SpringerPlus 5* (2016).

[63] Gang Yan and Jian Li. 2020. RL-Bélády: A Unified Learning Framework for Content Caching. In *Proc. of ACM Multimedia*.

[64] Juncheng Yang, Yao Yue, and Rashmi Vinayak. 2020. A Large Scale Analysis of Hundreds of In-memory Cache Clusters at Twitter. In *Proc. of USENIX OSDI*.

[65] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. 2018. A Deep Reinforcement Learning-based Framework for Content Caching. In *Proc. of IEEE CISS*.



(a) CDN-A (b) CDN-B (c) CDN-C (d) Wikipedia

Figure 12: Performance of detection mechanism.

A APPENDIX

A.1 Analysis of *HRO*

We consider a set of N distinct contents, where the successive times when content i is requested are denoted as $\{\tau_{ik}, k \in \mathbb{Z}\}$. Denote $\{\tau_k, k \in \mathbb{Z}\}$ as the superposition of N such processes. A hazard rate (HR) based policy [53] was shown to be an upper bound for all non-anticipative caching policies. However, its computation requires exact knowledge of request distributions, i.e., the c.d.f.. We generalize this result and develop a new online upper bound without such exact knowledge (Section 3). We approximate the request process as a Poisson process to compute the c.d.f., which is an accurate approximation for the point process [52, 62] under the assumption that the number of requests in each sliding window is large as in our system. As in [53], it can be shown that such an approximated HR based policy is still an upper bound when taking content size into consideration as in (2). Specifically, consider any caching policy π and let $H_k = 1$ if the k-th request is a hit and $H_k = 0$ otherwise under policy π. Denote $N_k = \sum_{k=1}^{K} H_k$ as the number of hits during the first K requests and M_k as the set of cached contents. Then it can be easily shown [53] that $\mathbb{E}[H_k] = \frac{\sum_{i\in M_k} \zeta_i(\tau_k)}{\sum_i^N \zeta_i(\tau_k)}$, which can be maximized by solving a 0-1 knapsack problem:

$$\max \sum_{i=1}^{N} x_i \zeta_i(\tau_k), \text{ s.t. } \sum_{i=1}^{N} s_i x_i \le M, \ x_i \in \{0,1\}.$$

This problem is NP-hard and the solution to the corresponding relaxed problem serves as an upper bound [31, 35]. Thus, an upper bound can be found on $\mathbb{E}[H_k]$ by ordering contents at time τ_k according to $\tilde{\zeta}_i(t)$ in (2).

Proposition A.1. *The hit probability achieved by the sized hazard rate (i.e., $\tilde{\zeta}_i(t)$ in (2)) based policy is an upper bound on that of all non-anticipative caching policies.*

Proof. Following the definition of HRO, we have $\sum_{i\in M_k^{\text{HRO}}} \tilde{\zeta}_i(t) \ge \sum_{i\in M_k^{\pi}} \tilde{\zeta}_i(t)$ for $\forall \pi$. Then by the definition of $\mathbb{E}[H_k]$, we directly have that $\mathbb{E}[H_k^{\text{HRO}}] \ge \mathbb{E}[H_k^{\pi}]$. Summing over k, we have that $\mathbb{E}[N_k^{\text{HRO}}] \ge \mathbb{E}[N_k^{\pi}]$. □

A.2 Additional Results on LHR Design

We measure the accuracy of our proposed LSM based model. We first generate synthetic datasets including 10 million requests for 10,000 contents where the requests follow Zipf distribution. We vary the Zipf parameter every 100,000 requests which leads to different synthetic datasets. We test the performance of our algorithm using four synthetic datasets, where the miss detection occurs only 3 times on average, i.e., with 97% detection accuracy when we set $\epsilon = 0.002$. Similarly, we test its performance using production traces, as shown in Figure 12. In Figure 12, the red box is the sliding window detected by our LSM model where our XGBM based learning model should be updated and trained. We observe that our detection mechanism can accurately detect 99% of the significant change of request patterns between two consecutive sliding windows.

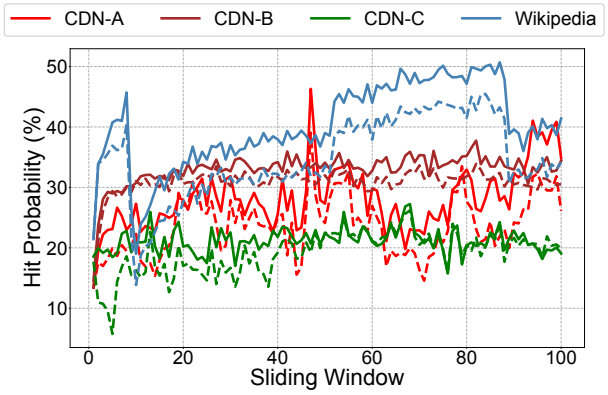


Figure 13: The content hit probabilities of LHR (solid lines) and Caffeine (dashed lines).

A.3 LHR Prototype vs. Caffeine

Similar to our LHR prototype in ATS in Section 6, we also implement and compare LHR with the caching benchmark Caffeine caching [47]. Caffeine is an in-memory cache using a Google Guava inspired API and the baseline policy is the state-of-the-art W-TinyLFU [23–25]. Different from ATS, the implementations in Caffeine are written

		CDN-A		CDN-B		CDN-C		Wikipedia	
Metric	Experiment	LHR	Caffeine	LHR	Caffeine	LHR	Caffeine	LHR	Caffeine
Throughput (Gbps)	max	5.45	5.42	6.54	6.43	6.75	6.51	5.56	5.24
Peak CPU (%)	max	21.6	17.2	22.1	18.7	25.7	23.3	25.3	20.1
Peak Mem (GB)	max	3.5	3.3	3.6	3.3	3.6	3.3	3.4	3.3
P90 Latency (ms)	normal	324.1	324.3	425.1	420.3	401.2	405.2	272.6	272.4
P99 Latency (ms)	normal	363.1	360.7	525.5	525.5	444.1	443.3	331.7	329.5
Overall Latency (ms)	average	223.5	231.3	248.0	251.4	316.1	322.7	163.0	172.5
Traffic (Gbps)	average	2.19	2.21	3.59	3.60	4.66	4.78	1.40	1.46
Content Hit (%)	normal	28.23	23.61	33.01	30.85	21.19	19.17	40.24	34.76

Table 4: Resource usage for LHR and Caffeine in max (throughput-bound) and normal (production-speed) experiments.

in Java. Similar to Section 7.2, we compare LHR and Caffeine in terms of hit probabilities and implementation overhead.

The hit probability comparison of LHR and Caffeine using CDN-A, CDN-B, CDN-C and Wikipedia traces with a cache of 64GB, 128GB, 16GB and 128GB are presented in Figure 13. We observe that LHR achieves a higher overall hit probability than Caffeine. We further compare the implementation overhead as shown in Table 4. Again, we observe that LHR has no measurable throughput overhead with a slightly increased peak CPU utilization compared to Caffeine. However, most production servers have sufficient CPU headroom even at their busiest mode. Similar to the settings in Section 7.2, it is clear that LHR outperforms Caffeine in terms of hit probability by 2%-6%. It is interesting to observe that LHR slightly increases the P90 and P99 latency but reduce the overall latency by 2%-6%. Finally, we observe that both LHR and. Caffeine uses a small cache size to store metadata.