Towards Latency Awareness for Content Delivery Network Caching

Gang Yan and Jian Li, *SUNY-Binghamton University*

https://www.usenix.org/conference/atc22/presentation/yan-gang

This paper is included in the Proceedings of the 2022 USENIX Annual Technical Conference.

July 11–13, 2022 • Carlsbad, CA, USA

978-1-939133-29-8

Open access to the Proceedings of the 2022 USENIX Annual Technical Conference is sponsored by

Towards Latency Awareness for Content Delivery Network Caching

Gang Yan
SUNY-Binghamton University

Jian Li
SUNY-Binghamton University

Abstract

Caches are pervasively used in content delivery networks (CDNs) to serve requests close to users and thus reduce content access latency. However, designing latency-optimal caches are challenging in the presence of *delayed hits*, which occur in high-throughput systems when multiple requests for the same content occur before the content is fetched from the remote server. In this paper, we propose a novel timer-based mechanism that provably optimizes the mean caching latency, providing a theoretical basis for the understanding and design of latency-aware (LA) caching that is fundamental to content delivery in latency-sensitive systems. Our timer-based model is able to derive a simple ranking function which quickly informs us the priority of a content for our goal to minimize latency. Based on that we propose a lightweight latency-aware caching algorithm named LA-Cache. We have implemented a prototype within Apache Traffic Server, a popular CDN server. The latency achieved by our implementations agrees closely with theoretical predictions of our model. Our experimental results using production traces show that LA-Cache consistently reduces latencies by 5%-15% compared to state-of-the-art methods depending on the backend RTTs.

1 Introduction

Content delivery networks (CDNs) carry more than 50% of today's Internet traffic [17] by caching a variety of contents such as videos, music, software downloads, etc. and delivering thousands of millions of user requests each day. CDNs deploy hundreds of thousands of servers across the world to serve user requests. If the requested content is available in the server near the user, a cache hit occurs and the user experiences a quicker response with a lower latency. Otherwise, a cache miss occurs and the requested content has to be fetched from the remote server with a dramatically increased latency. As a result, there has been a renewed focus on increasing cache hits [11, 16, 31, 40], which can significantly improve the content delivery of the Internet.

The recent trends of improving caching efficiency in terms of *maximizing caching hits* mostly focus on designing different content caching algorithms, including but not limited to GDSF [15], ARC [44], CAR [6], LHD [7], A-LRU [37], AdaptSize [11], CACA [29], LRB [53], RL-Cache [35], RL-Bélády [58], DeepCache [45] and LHR [59]. However, most of these algorithms assume that the user-perceived latency upon a cache hit is *negligible* (i.e., zero delay). Though this assumption has been widely used in the caching literature, some recent efforts start linking it to the potential performance degradation when *minimizing the end-user latency* in the presence of *delayed hits* [27, 55]. Notably, the latest series of works [4, 42] reveals that *a delayed hit* can occur in real-world systems, especially in high-throughput systems when multiple requests to the same content occur before the requested content is fetched from the remote server. As a result, the aforementioned caching algorithms fail to minimizing user-perceived latency in the presence of delayed hit since they are designed under the assumption that delayed hits does not exist. See Section 2 for more detail.

Despite the insightful findings in [4, 42], there remains a major *gap* between the delayed hits observation and the goal of efficient *online* latency-aware caching algorithm design. This is due to the fact that delayed hits were *only* identified and overcome through a hard *offline* optimization problem assuming that all contents are the *same size*, and the fetching latency from remote server upon a cache miss is *uniform* across different contents. However, it is well-known that content sizes often vary widely in production CDNs from a few bytes [46] to several gigabytes [31]. Additionally, the fetching latencies upon cache misses in production systems may vary over time due to the network conditions (e.g., bandwidth), and differ across content sizes since large contents often require longer fetching latencies (e.g., multiple RTTs). These facts introduce an additional layer of complexity in the design of *online* algorithms for minimizing latency in the presence of delayed hits, which largely remain elusive in the literature.

This paper closes this gap by developing a novel and lightweight timer-based mechanism to account for the impact

of delayed hits for contents with variable sizes and different fetching latencies that provably optimizes the mean caching latency. This provides a theoretical basis for the understanding and design of latency-aware caching that is fundamental to content delivery in latency-sensitive systems. In this approach, each content is associated with a timer indicating the fetching latency between the cache and remote server, which can be variable across different contents and systems. Upon a cache miss, all requests (i.e., delayed hits) arriving at the cache during a certain time period dictated by its timer suffer a corresponding latency before these requests are truly served. This approach is able to explicitly characterize the expected average latency of a caching system in the presence of delayed hits. This further enables us to derive a simple ranking function which can quickly prioritize contents for the purpose of minimizing latency.

This paper makes the following research contributions:

- To the best of our knowledge, our proposed timer-based model is the first to provide a theoretical basis for understanding the impact of delayed hits in latency-sensitive caching systems in an online manner. This enables us to design a lightweight online latency-aware caching algorithm which can capture the variable fetching latency and different content sizes in real-world systems.
- Using this timer-based model, we explicitly characterize the mean latency of each content in the presence of delayed hits and derive a simple ranking function to prioritize contents so as to minimize the mean latency. We then propose a lightweight latency-aware caching algorithm named LA-Cache.
- We have implemented the LA-Cache prototype within Apache Traffic Server, a popular CDN server, and evaluate the performance of LA-Cache using production traces. Our empirical results are in close alignment with our theoretical model predictions. Furthermore, we show that LA-Cache consistently outperforms conventional caching algorithms by reducing the latency by 5%-15% depending on the backend RTTs.

The rest of the paper is organized as follows. We introduce the motivations and opportunities in designing latency-aware caching in Section 2. We present the model for delayed hits and explicitly characterize the mean latency in Section 3. We propose the latency-aware algorithm LA-Cache in Section 4, and present its prototype design in Section 5. Evaluation results are shown in Section 6. We discuss related work in Section 7 and conclude the paper in Section 8. Additional results are presented in the supplementary material.

2 Background and Motivation

We begin by motivating the existing of delayed hits in latency-sensitive systems and showing the fundamental limitations of existing algorithms.

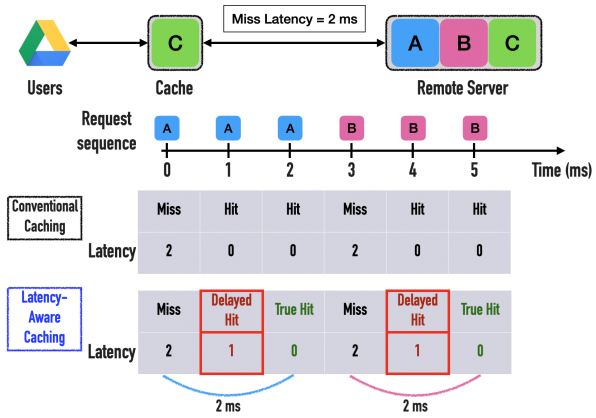


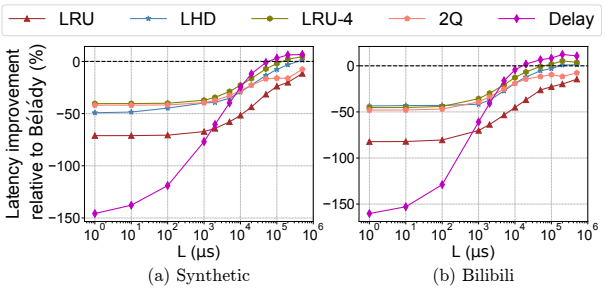
Figure 1: A motivating example for latency-aware caching where delayed hits occur. Suppose that the remote server stores all three contents named A, B, and C, the cache size is 1, and the miss latency (i.e., the Round-trip Time (RTT)) between the cache and the remote server is 2 ms. We also have a request sequence of A,A,A,B,B,B, ... with a new request arriving at the cache every 1 ms.

2.1 A Motivating Example

As a motivating example, we consider a basic delayed hit scenario in real-world systems as shown in Figure 1. Upon the first request to content 'A', a cache miss occurs and the cache must fetch content 'A' from the remote server with a latency of 2 ms. The next two requests to content 'A' can be directly served from the cache and experience a latency corresponding to a cache hit, *which is assumed to be zero in conventional caching algorithms.* Similar process happens for the requests to content 'B'. Since it takes 2 ms to fetch content 'A' from the remote server, *how is it possible for the second request to content 'A' that arrived just 1 ms after the miss was served with a zero latency?* Clearly, something is wrong.

Contrast to the ideal assumption in conventional caching algorithm design, the following *actually happens*. Upon the first request to content 'A', a cache miss is claimed and a fetch is triggered with a latency of 2 ms. Since the RTT is 2 ms, content 'A' will only arrive in the cache at time t=2 ms. As a result, the request at t=1 ms is queued behind the original miss, and must wait (at least) 1 ms to be served. At time t=2 ms, content 'A' arrives at the cache and all queued requests (including the one just arrived) are resolved. Similar process happens for the requests to content 'B'. These requests (e.g., requests to content 'A' at t=1 ms and to content 'B' at t= 4 ms) are called *delayed hits* since they neither suffer the latency of a cache miss[1] nor that of a true cache hit (e.g., requests to

[1] We interchangeably use the terms of "miss latency", "fetching latency"



(a) Synthetic (b) Bilibili

Figure 2: Conventional caching algorithms fail to minimize latency: As the fetching latency (the value of L) increases, conventional caching algorithms outperform the Bélády's offline MIN algorithm in term of mean latency on a 256GB cache. For example, LRU-4 outperforms Bélády when L is greater than 50 ms in the Bilibili trace.

content 'A' at t=2 ms and to content 'B' at t= 5 ms).

In general, delayed hits occur in *high-throughput systems* when multiple requests to the same content occur before the requested content is fetched from the remote server [4, 27, 42, 55]. *This phenomenon has become increasing perceptible due to the growing ratio between system throughputs and latencies in a wide range of systems.* For example, the wide-area latency between a CDN forward proxy and a central data center is only marginally improving, while newer technologies boast order-of-magnitude throughput improvement with the network links moving from 10 Gbps to 400 Gbps [24]. The fundamental problem is that latencies are closer and closer to limits imposed by the speed of light, while throughputs keep growing unboundedly. Hence, minimizing the impact of delayed hits is a key performance objective in latency-aware caching systems.

2.2 Limitations of Existing Algorithms

The above example indicates that conventional caching algorithms fail to capture the impact of delayed hits, which can be significant especially in systems with high latency to the remote server. One reason contributes to this failure is that conventional caching algorithms are designed to maximize cache hits under the assumption that all cache hits result in zero delay, i.e., they equally treat delayed hits and true hits. To that end, the latency measured by conventional caching algorithms significantly *underestimate true latency* in the presence of delayed hits. Some so-called "cache hits" will experience latencies closer to the high latency of a cache miss than the low latency of a true hit in practice.

As a consequence of this discrepancy, conventional caching algorithms fail to minimize latency although some caches were deployed for this purpose, which were actually treated equivalently to maximize cache hits regardless of delayed hits.

and "RTT" in this paper.

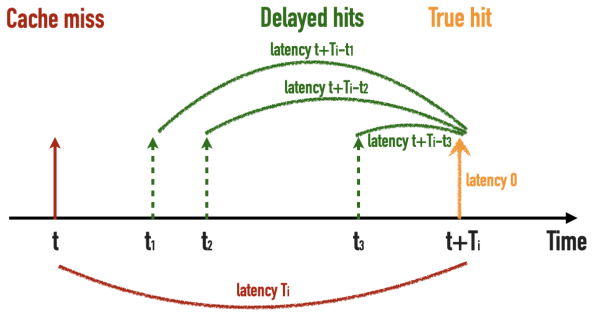


Figure 3: The illustration of our proposed timer-based model for a particular content i in the presence of delayed hits.

For example, the Bélády's offline MIN algorithm[2] [8] is the well-known optimal algorithm in maximizing cache hits and minimizing latency when all contents are of the same size and delayed hits are treated as true hits. However, Bélády is no longer optimal in minimizing latency in the presence of delayed hits, as illustrated in Figure 2 (More details in Section 6). Therefore, *the goal of maximizing cache hits for most conventional caching algorithms and the goal of minimizing latency are not equivalent, in the presence of delayed hits*[3]. *We need to design new online algorithms for latency-sensitive caching systems.*

3 Model for Latency Minimization

We consider the problem of minimizing latency for content delivery given delayed hits in latency-sensitive systems. In particular, we aim at designing a latency-aware caching policy that minimizes *the mean latency of all requests*. For ease of exposition, we denote the latency to fetch a particular content i from the remote server as L_i, which can vary across contents and servers. Therefore, the latency is (a) L_i upon a cache miss, (b) between 0 and L_i upon a delayed hit, and (c) 0 upon a true cache hit.

3.1 Modeling Delayed Hits

Our key insight is to build a novel connection between delayed hits and the timer-based caching policy. Figure 3 shows such a novel connection for a particular content i. More specifically, each content i has a timer value L_i which represents the latency to retrieve the content from the remote server upon a cache miss. In other words, upon a cache miss at time t, a request is sent to the remote server and content i will be

[2]The Bélády's offline MIN algorithm always evicts the content with the furthest next request.

[3]It has been shown [42] that the latency objective is not antimonotone for caching problems with delayed hits. In other words, a caching algorithm that improves average caching latency under delayed hits might even lower the true hit rate. Hence optimizing caching latency is fundamentally different from optimizing cache hits.

fetched and inserted into the cache after L_i time slots, i.e., at time $t+L_i$. The timer counts down and until the timer expires, any new requests for content i during $[t,t+L_i)$ are called *delayed hits*. These requests are queued behind the original cache miss occurred at time t, and resolved at time $t+L_i$ with a corresponding latency of $t+L_i-t'$ for the request occurred at time t'. The request arrives at time $t+L_i$ is satisfied immediately with a latency of 0, and hence is called the true hit.

3.2 Mean Latency of All Requests

Suppose that the requests for content i arrive at the cache according to a Poisson process[4] with rate λ_i. Let D_i be the expected aggregated latency experienced by the arrival of requests for content i (i.e., delayed hits) upon a cache miss. The probability of not finding content i in the cache is the cache miss probability m_i, which satisfies $m_i = \frac{1}{1+\lambda_i L_i}$ derived in the context of timer-based caches [25].

Proposition 1. *The expected latency experienced by the request arrival of content i is* $\left(1+\frac{1}{1+\lambda_i L_i}\right)\frac{L_i}{2}$.

Proof. We first compute the expected aggregated latency for content i. Upon a cache miss, the request is sent to the remote server and content i is fetched and inserted into the cache after L_i time slots. Thus, the latency of a cache miss is L_i. A delayed hit occurs for every arrival of content i during time $[t,t+L_i)$ since the content i has not been inserted into the cache yet, and the corresponding latency for request at time $t' \in [t,t+L_i)$ is $t+L_i-t'$. Thus *the expected aggregated latency by all delayed hits* in the interval of $[t,t+L_i)$ is

$$D_i := \int_t^{t+L_i} \lambda_i (t+L_i-x)dx = \frac{\lambda_i L_i^2}{2}. \tag{1}$$

Since the requests for content i follow a Poisson process, the expected number of requests, i.e., the expected number of delayed hits is $\lambda_i L_i$. Hence, the expected latency for each delayed hit is $\frac{\lambda_i L_i^2}{2}/\lambda_i L_i = \frac{L_i}{2}$. Then *the mean latency experienced by the requests of content i* is a weighted sum of the latency from the cache miss and the latency from delayed hits, which satisfies

$$\bar{D}_i = m_i L_i + (1-m_i)\frac{L_i}{2} = \left(1+\frac{1}{1+\lambda_i L_i}\right)\frac{L_i}{2}. \tag{2}$$

□

[4]Poisson arrivals are widely used in the literature, e.g., [32, 36, 38, 41, 43]. However, our model holds for general stationary process [5] at the cost of complicated notations [25, 26]. We relax the Poisson arrivals assumption for our algorithm design, implementation and empirical evaluation.

Corollary 1. *The mean latency experienced by all requests to N distinct contents satisfies*

$$\bar{D} := \sum_{i=1}^{N} \lambda_i \bar{D}_i = \sum_{i=1}^{N} \frac{\lambda_i L_i}{2}\left(1+\frac{1}{1+\lambda_i L_i}\right). \tag{3}$$

Remark 1. *The main advantage of our proposed timer-based model for delayed hits is that it can be easily deployed in latency-sensitive systems to provide a precise and theoretically-validated latency. Although real-world systems do not have strict Poisson arrivals for any content (see Section 6), we will show in Section 6 that our proposed theoretical model with Poisson assumptions works well in practice.*

Remark 2. *Timer-based caches have been extensively studied in the community [9, 14, 23, 25, 26, 33, 48–50] which are used to store frequently requested contents in computer systems. In a timer-based cache, a timer value is set when a content is first cached and evict the content when the timer expires. While our timer-based delayed hits model serves a different purpose, some of the theoretical analysis of timer caches directly apply (e.g., the decoupling nature of contents in timer cache analysis). This novel connection between timer-based delayed hits and traditional timer-based caches allows us to bring to bear the analytical work done in the conventional caching domain into latency-sensitive systems.*

4 Latency-Aware Cache

Our novel and lightweight timer-based model for delayed hits provides us an opportunity to design a *latency-aware* (LA) caching policy that achieves optimal latency for any given request sequence with variable content fetching latency from remote server.

Having characterized the mean latency experienced by all requests (see Corollary 1), we turn to derive a simple *ranking function* that can quickly prioritize contents so as to minimize latency[5]. Ranking function has been widely used in the design of conventional caching algorithms. For example, the classic Least Recently Used (LRU) [18] (or its variants) which are employed in major CDNs today, is based on a ranking function of content request recency, while Least Frequency Used (LFU) ranks contents by how frequently they have been

[5]We build a novel connection between delayed hits and timer-based caches for the sake of characterizing the mean latency of each content in the presence of delayed hits. See Remark 2. In particular, the fetching latency upon a cache miss is analogous to a timer. Need to mention that we are not considering the conventional timer-based caches, where each content is decoupled by the timer. Instead, our timer-based model serves a different purpose. For our latency-aware caching policy design, all contents are coupled by the cache capacity constraint and hence a ranking function (which is derived using the timer-based model, see equation (4)) is needed to make caching decisions. We use the term "timer-based model" since we can bring some analytical results from traditional timer-based cache domain into latency-minimization analysis, such as the expression of m_i.

requested. However, all these ranking functions prioritize contents for maximizing cache hits whereas we seek a ranking function so as to minimizing latency.

Our ranking function is inspired by the theoretically-sounded latency derived from the timer-based model for delayed hits, which in particular the following two intuitions. First, we consider the metric of aggregated latency computed in (1), which is the sum of latency due to a cache miss and any delay hits in the next L_i time slots which occur before the corresponding content is fetched and inserted into the cache. Intuitively, a content with a higher latency cost increases the average latency more than a content with a lower latency cost, and hence should be prioritized. Second, we consider the metric of mean latency for each request of a content, which is computed in (2). It is clear that *a burst of requests* to a content can contribute more to the average latency than a sparse of requests to a content. Following these two intuitions, we derive a ranking function based on both the aggregated latency and the mean latency for each content i as

$$\tilde{f}(i) = \frac{\text{Aggregated latency}}{\text{Mean latency for each request}} = \frac{D_i}{\bar{D}_i} = \frac{\lambda_i L_i (1+\lambda_i L_i)}{2+\lambda_i L_i}. \tag{4}$$

Now we introduce the *latency-aware* policy based on the above ranking function, abbreviated as LA-Cache. LA-Cache always places in the cache the C contents with the largest value of their corresponding ranking functions, i.e., if $\tilde{f}(1) \geq \tilde{f}(2) \geq \cdots \geq \tilde{f}(N)$, then contents $1, 2, \cdots, C$ are cached given that the cache size is C.

Proposition 2. *LA-Cache achieves the minimum mean latency experienced by all requests compared to any other online policies.*

Proof. This is clear from the definition of LA-Cache policy since both the ranking function (4) and the expected latency for each content i in (2) (resp. (3)) are monotonically increasing in $\lambda_i L_i$. □

Remark 3. *Given (3) (resp. (2)), it can be easily shown that $\bar{D}$ (resp. $\bar{D}_i$) is increasing in $\lambda_i L_i$, which can be interpreted as the expected number of delayed hits upon a cache miss for content i. Since our ranking function (4) is also increasing in $\lambda_i L_i$, it is clear that LA-Cache prioritizes caching bursty contents (We will formally define and evaluate the burstiness of a content using a burstiness measure [34] in Section 6). Intuitively this is correct since a busty content usually refers to a large number of requests in a shorter time period. This will result in a larger aggregated latency for the content upon a cache miss. As a result, prioritizing such a content can reduce the latency.*

4.1 From Theory to Practice

The above ranking function is defined under the assumption that the content sizes in the system are equal. However, the content sizes in production system usually vary significantly from a few bytes [46] to several gigabytes [31]. To overcome this drawback, we redefine the ranking function by incorporating the content size s_i, satisfying

$$f(i) = \frac{\tilde{f}(i)}{s_i} = \frac{\lambda_i L_i (1+\lambda_i L_i)}{(2+\lambda_i L_i) s_i}. \tag{5}$$

For the notation abuse, we call the policy using this refined ranking function as LA-Cache in the rest of the paper.

To compute the ranking function and obtain the ranking function based policy LA-Cache, the content arrival rate λ_i is needed[6]. This is straightforward for synthetic workload; however, the content arrival rate is usually unknown and varying over time in real-world systems. To this end, we use estimation techniques to approximate the request rates. For any content i, let X_j^i denote the random variable corresponding to the inter-arrival times for the requests for content i, and $\bar{X}^i$ be its mean. We can approximate the mean inter-arrival time as $\hat{\bar{X}}^i = \sum_{j=1}^{K} X_j^i / K$. It can be easily shown that $\hat{\bar{X}}^i$ is an unbiased estimator of $1/\lambda_i$. However, keep tracking of all X_j^i's for each content j from the very beginning will increase the overhead. As it is well-known that content request processes in production systems are highly dynamic and non-stationary, we further consider a sliding time window, and only use the X_j^i's within the window to estimate $\hat{\bar{X}}^i$. In Section 6, we will use this estimator to compute the ranking function for evaluating our algorithm.

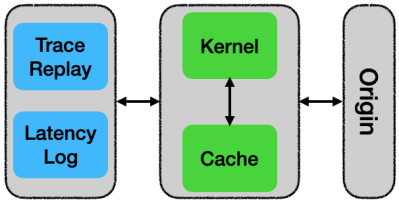


Figure 4: The architecture of our ATS prototype.

5 Implementation

We implement the LA-Cache prototype within Apache Traffic Server (ATS) [2], a popular CDN server. An LA-Cache cache simulator has also been implemented for the sake of comparison with a wide range of state-of-the-art caching algorithms. The two implementations are written in C++.

5.1 LA-Cache Prototype

ATS is a multi-threaded and event-based CDN caching server with a space-efficient in-memory lookup data structure as

[6] Again, L_i indicates the content retrieval latency upon a cache miss (related to RTTs of the system). In our experiments, we evaluate the impact of its value on the system performance. See Section 6.

	Description
LRU	Recency-based heuristic.
LRU-K	Recency-based heuristic. Evict content with the oldest K-th reference in the past.
LHD [7]	Using a ranking function of the content expected hit density.
2Q [52]	Manage caching decisions through a FIFO queue and a LRU queue.
Delay	An offline heuristic knows future request latency caused by evicting a content from cache now.
LRU-MAD [4]	Calculate content average latency from history and then combine with LHD as a ranking function.
LHD-MAD [4]	Calculate content average latency from history and then combine with LRU as a ranking function.

Table 1: Overview of state-of-the-art caching algorithms.

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/Os. The overview of our LA-Cache prototype is presented in Figure 4.

Upon a new request, ATS implements the following steps. Based on the URL, it looks up the local caches to check whether the corresponding content is available. If the requested content is already in the caches (i.e., a true hit), then the request is immediately satisfied by replaying a response back to the user. Otherwise, the request is sent to the kernel, which mains the request history received from users, i.e., a separate queue for each cache miss (see Section 1). If the current request belongs to one queue, then it will be added to the queue (i.e., a delayed hit). Otherwise, the kernel sets up a new queue for the content (i.e., a miss), and the request is forwarded to the original remote server. To deal with real traces, the requests are sent to a proxy sever in the recorded order (via the trace replayer). All users and the master sever communicate with each other by the TCP protocol.

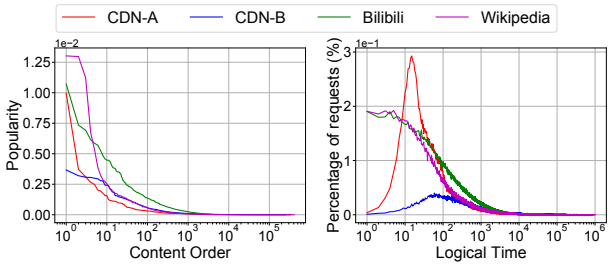
We implement LA-Cache on top of ATS. To do so, we replace the lookup data structures for ATS cache with the LA-Cache described in Section 4. The content admission[7] and look-up processes can be implemented asynchronously. These two processes are used to update parameters so as to make eviction decision[8]. 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. But as for the flash abstraction layer which is very important in production system (i.e., we have no access, e.g., RIPQ [56]), we only emulate the workings due to some difficulties, reading offsets randomly and writing sequentially to the disk. Since the memory cache is usually small which has little impact on hit probability [11], we keep this part of ATS unchanged. In summary, we implement the framework by only modifying about 100 lines of codes in ATS. The LA-Cache framework library contains about 600 lines of codes.

[7] Content admission decides whether to cache the content upon a cache miss.

[8] Eviction process determines which content to evict when the cache is full.

Dataset	*CDN-A*	*CDN-B*	*Bilibili*	*Wiki*
Duration (Hours)	24	9.9	18.7	0.1
Unique contents	330,446	162,104	4,852	407,919
Total requests (Millions)	0.97	1	1	1
Mean content size (MB)	25.5	68.4	563.5	69.8
Max content size (MB)	7,790	38,392	565.8	3,840

Table 2: Key characteristics of the production traces used throughout our evaluation spanning different CDNs.



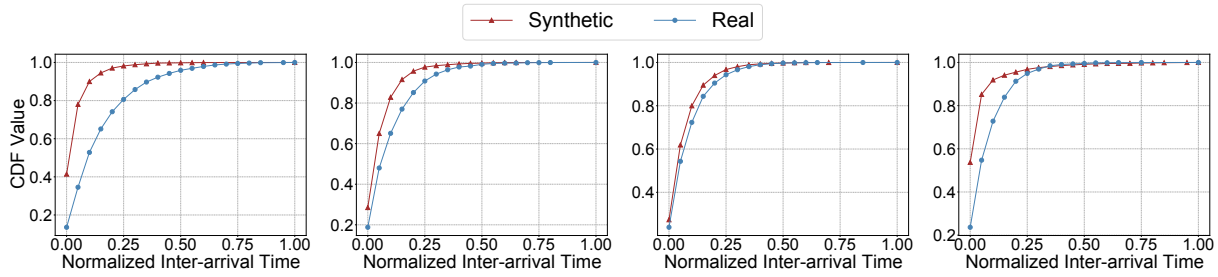
(a) Popularity (b) Inter-arrival Time

Figure 5: The production traces used in our evaluation comes from different CDNs and thus exhibit different request patterns.

5.2 LA-Cache Simulator

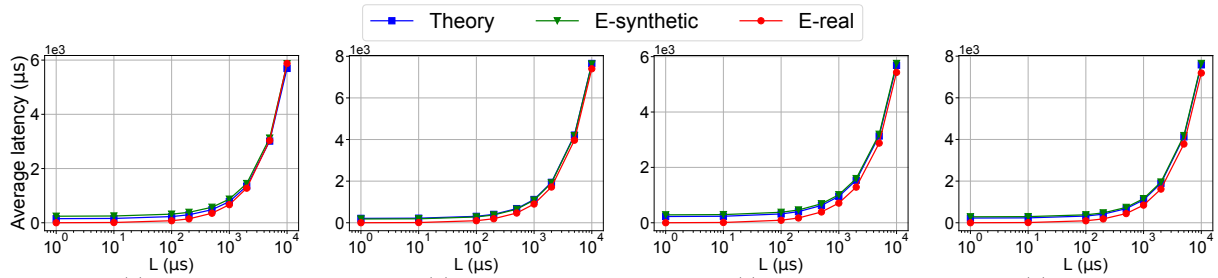
We implement an LA-Cache simulator that includes a wide range of conventional caching algorithms. For ease of exposition, we only report the results for the "best-performing" algorithms as summarized in Table 1. Finally, our implementation benefits from existing caching simulators such as libCacheSim [39] and LRB simulators [53].

Availability. The code for the prototype design, the cache simulator as well as all evaluations in Section 6 are available at https://github.com/GYan58/la-cache.



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

Figure 6: CDF of inter-arrival times for real trace and synthetic trace for two representative contents.



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

Figure 7: Comparison between theoretical and empirical results of mean latency for production traces. The mean latency graph for the *Theory*, the *E-real* and the *E-synthetic* is slightly offset for ease of visualization.

6 Evaluation

In this section, we evaluate our LA-Cache prototype. We also conduct simulations to compare LA-Cache to a wide range of state-of-the-art algorithms using production traces. Our results address the following questions:

- How accurate is our timer-based model for delayed hits given that real-world systems do not have strict Poisson arrivals (Section 6.2)?
- What is the benefit of using our LA-Cache prototype compared to existing CDN production systems in terms of latency and implementation overhead (Section 6.3)?
- What is the performance of LA-Cache compared to state-of-the-art algorithms on a wide range of production CDN traces under various cache settings (e.g., different fetching latency and cache sizes) (Section 6.4)?

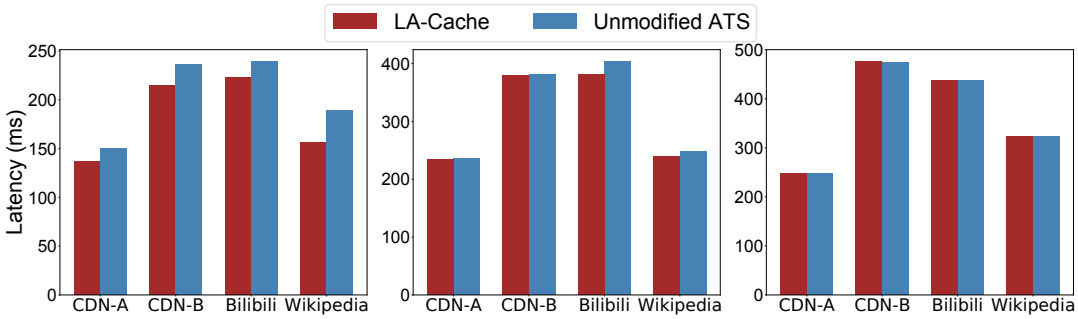
6.1 Methodology

Traces. We consider production traces from four CDNs, two 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) Bilibili [12, 29], collected from a Video-on-Demand (VoD) provider with millions of HTTP requests; and (4) a Wikipedia (Wiki) trace [53] collected on a west-coast node serving photos and other media content. We summarize the trace characteristics in Table 2 and present two key distributions of these traces in Figure 5. The traces typically span several tens to hundreds of thousands of requests, and tens of thousands of contents with sizes varying from 10KB to 10^4MB. The total bytes requested are on the order of TBs; however, the active bytes[9] are on average on the order of GBs. As a result, we choose the cache size in the range of 128GB to 1,024GB for different traces in our evaluation. For ease of readability, we only present results using a 256GB cache and a 512GB cache for each trace in the rest of this section. Similar observations hold for other cache sizes and hence are omitted. Finally, the average inter-request time is 6.5 ms for CDN-A, 9.1 ms for CDN-B, 1.1 ms for Bilibili and 5.4 ms for Wikipedia.

Baselines. We compare LA-Cache with a wide range of state-of-the-art algorithms. For ease of exposition, we only show the few "best-performing" algorithms (see Table 1) in the following figures.

Performance evaluation. We evaluate the performance of these algorithms using four production workloads described above with different fetching latencies and cache sizes. All results are generated by running on Ubuntu 18.04 with an Intel(R) Core(TM) i7-6700HQ processor and a 8GB RAM.

[9] A content is said to be active at time t in a trace if t lies between its first and last requests. The total size of active contents at time t is defined as active bytes [35].



(a) Average Latency (b) P90 Latency (c) P99 Latency

Figure 8: The latencies of LA-Cache and unmodified ATS using a 256GB cache.

		CDN-A		CDN-B		Bilibili		Wikipedia	
Metric	Experiment	LA-Cache	ATS	LA-Cache	ATS	LA-Cache	ATS	LA-Cache	ATS
Throughput (Gbps)	max	8.64	8.28	11.16	10.73	11.93	11.88	9.98	9.36
Overall CPU (%)	average	27.4	2.8	28.2	3.7	27.1	2.8	28.5	4.0
Peak Mem (GB)	max	2.6	2.3	2.7	2.5	2.3	2.2	2.3	2.2
P90 Latency (ms)	normal	234.2	235.6	378.8	381.6	390.7	403.7	239.0	247.3
P99 Latency (ms)	normal	247.6	248.2	474.1	474.6	435.5	436.5	323.7	324.0
Overall Latency (ms)	average	**137.2**	150.6	**215.0**	236.7	**223.5**	239.4	**156.4**	188.9

Table 3: Resource usage for LA-Cache and ATS in max (throughput-bound) and normal (production-speed) experiments.

6.2 Accuracy of Timer-based Model

We first show that our proposed timer-based model for delayed hits (see Sections 3 and 4) is accurate.

Non-Poisson arrivals in production traces. We first show that production traces do not have strict Poisson arrivals for any content. To that end, we generate a synthetic trace based on the real trace, where each content follows Poisson arrivals with the same average arrival rate as in the corresponding trace. We analyze the distribution of the inter-arrival times for the corresponding contents from the real and synthetic traces. It is clearly shown in Figure 6 that they are visibly different. Similarly trends hold for other contents in all production traces considered in this paper, i.e., production traces do not have strict Poisson arrivals for any content.

Comparison between theoretical and empirical results. We now show that despite the fact that production traces may not be strictly Poisson, our proposed timer-based model with Poisson assumption works well in practice. To this end, we compare the theoretically computed average latency (calculated using Equation (3)) to the empirically computed latency. In particular, we compare three results: (i) *Theory*: theoretical latency for the trace; (ii) *E-real*: empirical latency for the trace; and (iii) *E-synthetic*: empirical latency for the synthetic Poisson trace with same content arrival rates as the trace as described earlier. Figure 7 compares the curves for all three cases in four production traces. We observe that the theoretical latency matches very well with the empirical latency for the synthetic trace while the empirical latency for the real trace only differs slightly with the other two.

6.3 Latency Reduction of LA-Cache Prototype

We first compare our LA-Cache prototype to the ATS production systems in mean latency and implementation overhead as shown in Figure 8 and Table 3. The average RTT is 200 ms.

Latencies. Figure 8 compares the mean latency, the 90-th percentile latency (P90 latency) and the 99-th percentile latency (P99 latency) of LA-Cache and unmodified ATS using four production traces with a 256GB cache. LA-Cache consistently reduces the latency compared to ATS by 5%-20% on average[10].

Implementation overhead. We then compare the implementation overhead of our LA-Cache prototype against unmodified ATS. We measure the throughput, CPU and memory utility under the "max" experiments, as shown in Table 3. We see that LA-Cache has no measurable throughput overhead but the peak CPU utilization increases to 27.4% from 2.8% for ATS under CDN-A, 28.2% from 3.7% for ATS under

[10]P90 (resp. P99) latency is the value of top 10% (resp. 1%) latency. Though P90 (resp. P99) latencies of LA-Cache are not significantly better than ATS, it only means that the largest latency values are similar. More importantly, it is obvious that the mean latency, a key metric for real system, of LA-Cache significantly outperforms ATS, i.e., LA-Cache improves mean latency greatly for most content requests.

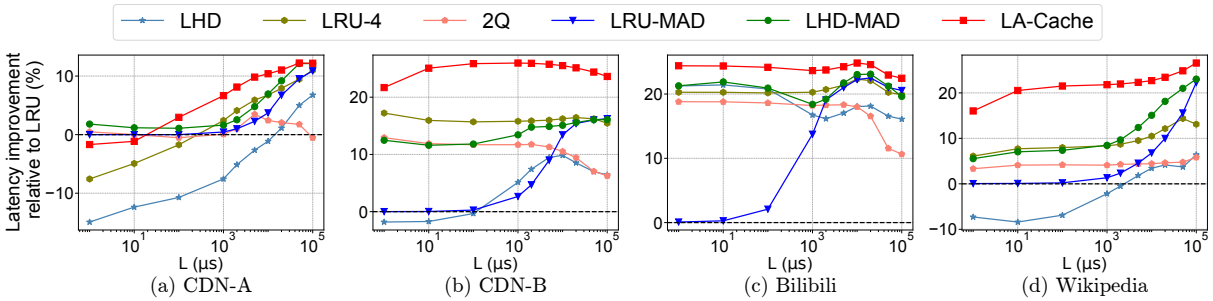


Figure 9: Comparison of mean latency improvement between LA-Cache and state-of-the-art algorithms relative to LRU using a 256GB cacahe.

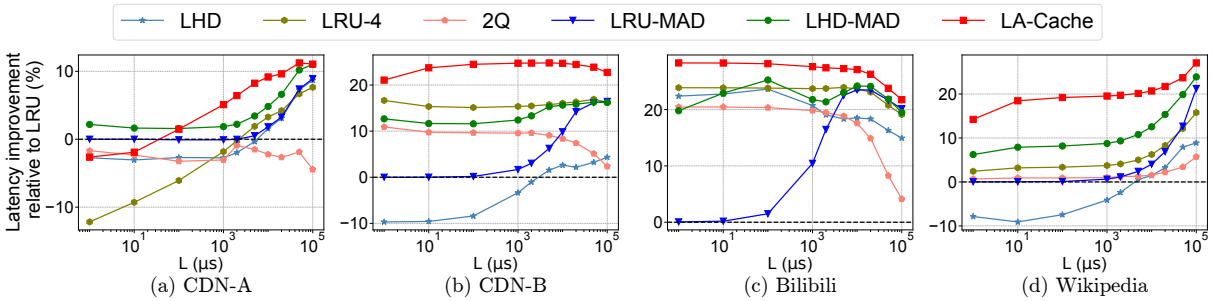


Figure 10: Comparison of mean latency improvement between LA-Cache and state-of-the-art algorithms relative to LRU using a 512GB cache.

CDN-B, 27.1% from 2.8% for ATS under Bilibili and 28.5% from 4.0% under Wikipedia. However, we note that most production servers, even at their busiest mode, have sufficient CPU headroom.

We replay our traces using its original timestamps and measure the latencies corresponding to cache misses, delayed hits and true hits. We call this "normal" experiments as shown in Table 3. It is clear that LA-Cache leads to significant latency reduction compared to ATS. More specifically, LA-Cache reduces the 90-th percentile latency (P90 latency) by 4%, the 99-th percentile latency (P99 latency) by 3%, and the overall average latency by 10% compared to ATS.

Finally, we measure the peak memory overhead for all traces and cache sizes, we observe that LA-Cache uses at most 1.1% of the cache size to store metadata. As we will show later, such a small loss in available caching space is more than offset by LA-Cache's significant latency reduction.

From our above experiments in ATS, we believe that LA-Cache is a practical design for today's CDNs and can be easily implemented in existing production CDN servers with modest resource overhead.

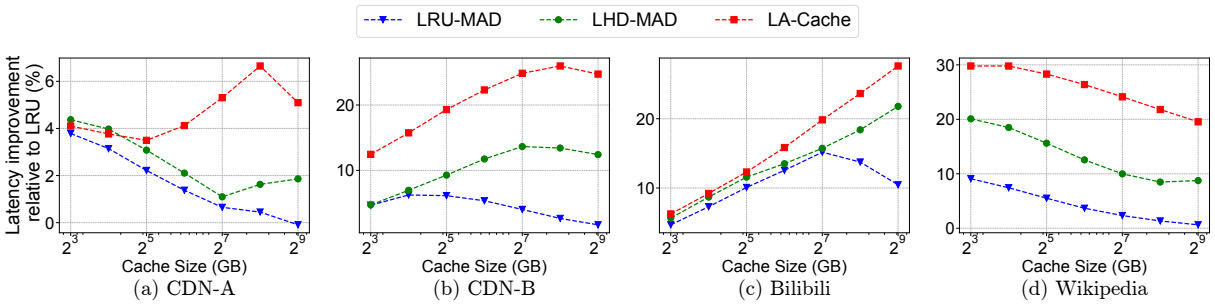
6.4 LA-Cache vs. State-of-the-art Algorithms

We further compare LA-Cache to a large number of state-of-the-art caching algorithms using four production traces with a wide range of fetching latencies and cache sizes.

Latency. Figures 9 and 10 compare the mean latency improvement of LA-Cache and state-of-the-art caching algorithms with respect to LRU with different fetching latencies using a 256GB and a 512GB cache, respectively. We choose LRU as the baseline since major CDNs today still employ LRU or its variants for content caching. The comparisons with respect to the offline Bélády are relegated to the supplementary material for ease of readability.

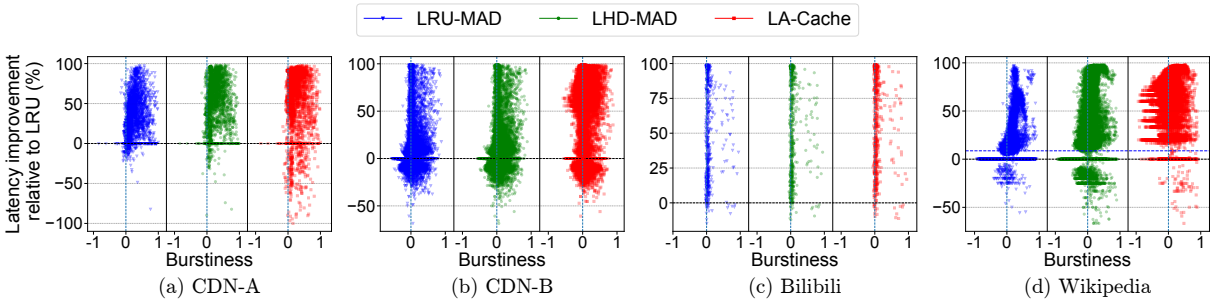
Our LA-Cache consistently outperforms the best state-of-the-art algorithms, i.e., "the best-performing" algorithms in Table 1. Overall, LA-Cache reduces the latency by 5%-15% on average. Note that LA-Cache is robust across all traces in latency reduction whereas no existing state-of-the-art algorithms could robustly reduce the latency across all traces. In particular, LA-Cache outperforms LHD-MAD and LRU-MAD, two recently proposed latency-aware caching algorithms [4]. Our interpretation is that our LA-Cache naturally offers a variable fetching latency for different contents as well as fully captures the varying content sizes whereas LHD-MAD or LRU-MAD are designed under the assumption that contents are of equal size, which is not the case in production systems (see Figure 5).

Impact of cache size. We further characterize the impact of cache size on the latency reduction of LA-Cache compared to state-of-the-art algorithms. Based on the results above,



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

Figure 11: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to LRU as a function of cache sizes.



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

Figure 12: Improvement from the burstiness of content requests relative to LRU using a 256GB cache.

we only focus on the comparison with two latency-aware algorithms LRU-MAD and LHD-MAD. We compute the latency reduction compared to LRU for all traces while using a fixed value of L=1 ms. From Figure 11, we observe that LA-Cache's improvement is between 4% and 30% compared to the widely deployed LRU. Finally, we see that LA-Cache outperforms LRU-MAD and LHD-MAD between 3% and 13% across different cache sizes.

Impact of burstiness. As motivated earlier in Section 2 as well as the design of our LA-Cache in Section 4, it is clear that a burst of requests to a content could contribute to the average latency more than a sparse of requests to a content. Now we turn to the question of *whether our intuition of burstiness indeed maps on the latency reduction of LA-Cache compared to state-of-the-art algorithms?*

To answer this question, we first need a measure to quantify the burstiness of a trace. A widely used metric is called the Goh-Barabasi score[11] [28]. However, it does not capture the impact of inter-arrival times, which play a significant role in delayed hits. To this end, we use a new burstiness measure [34] that not only captures the mean, variance of request sequences but also the inter-arrival times. The large the burstiness value, the busty the requests are. We refer interested readers to [34] for a detailed discussion[12].

Using this measure, we show that CDN-A has the largest burstiness value[13], which means that the inter-arrival times for most contents are completely random. On the other hand, CDN-B has the smallest burstiness value. These observations are consistently with the trace characteristics (e.g., the inter-arrival time distribution) in Figure 5, where about 25%-30% contents account for more than 80% requests in CDN-A, while the requests in CDN-B are more uniform.

We characterize the burstiness of each content and the corresponding latency improvement of this content due to the latency-aware caching algorithms in Figures 12 and 13, where we consider the setting as above with a fixed value of L=1 ms. We observe that bursty contents (with a large burstiness value) incur a lower latency compared to LRU in general. Furthermore, we indeed observe that LA-Cache prioritizes more bursty requests compared to other state-of-the-art algorithms, which contributes to the latency reduction of LA-Cache. As a result, the overall mean latency is reduced. For example,

[11]The value of Goh-Barabasi score is between -1 to 1, with -1 being a regular request sequence, 0 being a random request sequence and 1 being a bursty request sequence.

[12]The Goh-Barabasi score is a statistical measure of burstiness in a sequence of events and is defined as $B = \frac{r-1}{r+1}$, where $r = \sigma/\mu$ is the coefficient of variation, σ and μ denote the standard deviation and the mean of inter-arrival times, respectively. However, the behavior of this score may not be robust with respect to finite-size request sequence. [34] redefined the bustiness score as $\frac{\sqrt{n+1}r-\sqrt{n-1}}{(\sqrt{n+1}-2)r+\sqrt{n-1}}$ where n is the sample size (i.e., number of requests in the sequence). This new score has been shown to quantify the burstiness in the empirical dataset without finite-size effects.

[13]We compute a weighted average burstiness score over all requests in the trace using the new burstiness measure [34], where the weight for each content is proportional to the total number of requests for this content.

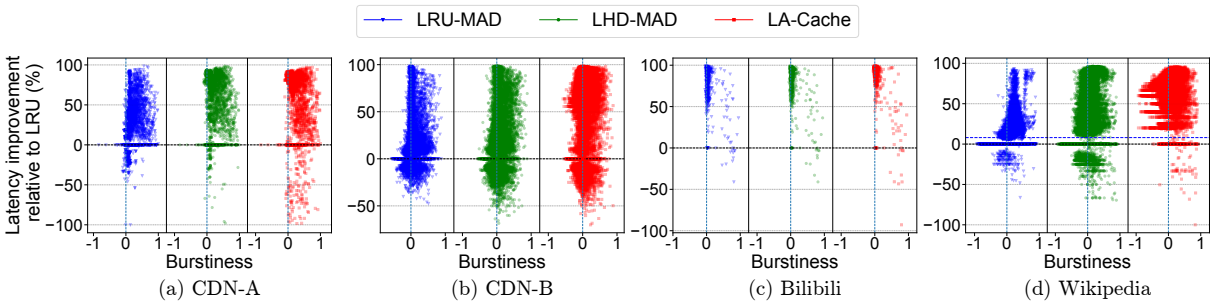


Figure 13: Improvement from the burstiness of content requests relative to LRU using a 512GB cache.

for Wiki in Figure 12 (d), it is obvious that LA-Cache prioritizes more bursty requests compared to LRU-MAD and LHD-MAD. This leads to a latency improvement of 22% for LA-Cache, while the improvements for LRU-MAD and LHD-MAD are 2% and 9%, respectively, as shown in Figure 9 ($L = 10^3 \mu s$). This phenomenon can also be observed when compared with the offline Bélády (see supplementary material). These observations further validate our intuitions on designing a ranking function to prioritize bursty contents so as to minimize latency (see Section 4).

7 Related Work

Caching algorithm design has been extensively studied over years. However, most of the previous works have focused on improving caching hit probabilities. We classify them by admission or eviction. The widely used admission algorithms include AdaptSize [11], TinyLFU [19] and SecondHit [40], and among others where static features such as content sizes are used for admission [1, 20]. A large number of works proposed eviction algorithms from classic Least Recently Used (LRU) [18], RANDOM, FIFO, to more sophisticated ones that are more difficult to implement in practice, e.g., LRU-K [47], LFU-DA [3, 51], GDSF [15], ARC [44], CAR [6] and among others, where recency, frequency or their combinations are usually used for eviction decision [7, 13, 30].

Recently, machine learning has been used for caching algorithm design. On the one hand, some focus on learning content popularities for content eviction via deep neural networks (DNNs), e.g., DeepCache [45], FNN-Cache [22], PopCache [54] and PA-Cache [21] or by approximating or imitating offline optimal Bélády for content eviction, e.g., LFO [10], LRB [53]. 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 [35], CACA [29], RL-Bélády [58] and among others [57, 60]. Again, most of these designs are focusing on improving cache hits rather than minimizing caching latency.

Closest to to our work is [4, 42]. In particular, [42] provides a lower bound on the performance of caching policies when delayed hits exist. [4] characterizes the impact of delayed hits and proposes an online approximation algorithm MAD based on a hard offline optimization problem. However, MAD fails to account for variable fetching latency and different content sizes, which are the cases in production CDNs and are both captured by our LA-Cache.

8 Conclusion

In this paper, we designed latency-aware caching in the presence of delayed hits, and proposed a novel timer-based mechanism to capture the impact of delayed hits which provably optimizes the mean caching latency. Furthermore, our model captured variable fetching latency and different content sizes, providing a theoretical basis for the understanding and design of latency-aware caching for content delivery in latency-sensitive systems. Using our timer-based model, we proposed a lightweight latency-aware caching algorithm LA-Cache. We implemented a LA-Cache prototype within Apache Traffic Sever. Using production traces, we showed that LA-Cache consistently outperformed state-of-the-art algorithms on latency reduction with modest resource overhead.

Acknowledgements

We would like to thank our anonymous reviewers for their valuable feedback. We would like to thank our shepherd's insightful comments that improved the quality of the paper immensely. This work was supported in part by the National Science Foundation (NSF) grants CRII-CNS-NeTS-2104880 and RINGS-2148309, and was supported in part by funds from OUSD R&E, NIST, and industry partners as specified in the Resilient & Intelligent NextG Systems (RINGS) program. 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. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding agencies.

References

[1] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. of ACM SIGCOMM*, 1996.

[2] Apache Traffic Server, 2020. https://trafficserver.apache.org/.

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

[4] Nirav Atre, Justine Sherry, Weina Wang, and Daniel S Berger. Caching with Delayed Hits. In *Proc. of ACM SIGCOMM*, 2020.

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

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

[7] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving Cache Hit Rate by Maximizing Hit Density. In *Proc. of USENIX NSDI*, 2018.

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

[9] D. Berger, P. Gland, S. Singla, and F. Ciucu. Exact Analysis of TTL Cache Networks. *Performance Evaluation*, 79:2–23, 2014.

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

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

[12] Bilibili. https://www.bilibili.com.

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

[14] H. Che, Y. Tung, and Z. Wang. Hierarchical Web Caching Systems: Modeling, Design and Experimental Results. *IEEE Journal on Selected Areas in Communications*, 20(7):1305–1314, 2002.

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

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

[17] Cisco. Cisco Visual Networking Index: Forecast and Trends 2022. *Cisco, Tech. Rep*, 2019.

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

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

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

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

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

[23] A. Ferragut, I. Rodríguez, and F. Paganini. Optimizing TTL Caches under Heavy-tailed Demands. In *Proc. of ACM SIGMETRICS*, 2016.

[24] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, et al. Azure Accelerated Networking: Smartnics in the Public Cloud. In *Proc. of USENIX NSDI*, 2018.

[25] N. C. Fofack, M. Dehghan, D. Towsley, M. Badov, and D. L. Goeckel. On the Performance of General Cache Networks. In *VALUETOOLS*, 2014.

[26] N. C. Fofack, P. Nain, G. Neglia, and D. Towsley. Analysis of TTL-based Cache Networks. In *VALUETOOLS*, 2012.

[27] Davy Genbrugge and Lieven Eeckhout. Memory Data Flow Modeling in Statistical Simulation for the Efficient Exploration of Microprocessor Design Spaces. *IEEE Transactions on Computers*, 57(1):41–54, 2007.

[28] K-I Goh and A-L Barabási. Burstiness and Memory in Complex Systems. *EPL (Europhysics Letters)*, 81(4):48002, 2008.

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

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

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

[32] Stratis Ioannidis and Edmund Yeh. Adaptive Caching Networks with Optimality Guarantees. *Proc. of ACM SIGMETRICS*, 2016.

[33] J. Jung, A. Berger, and H. Balakrishnan. Analysis of TTL-based Cache Networks. In *IEEE INFOCOM*, 2003.

[34] Eun-Kyeong Kim and Hang-Hyun Jo. Measuring Burstiness for Finite Event Sequences. *Physical Review E*, 94(3):032311, 2016.

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

[36] Jian Li, Truong Khoa Phan, Wei Koong Chai, Daphne Tuncer, George Pavlou, David Griffin, and Miguel Rio. DR-Cache: Distributed Resilient Caching with Latency Guarantees. In *Proc. of IEEE INFOCOM*, 2018.

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

[38] Yuanyuan Li and Stratis Ioannidis. Universally Stable Cache Networks. In *Proc. of IEEE INFOCOM*, 2020.

[39] limCacheSim. https://github.com/1a1a11a/libCacheSim.

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

[41] Milad Mahdian, Armin Moharrer, Stratis Ioannidis, and Edmund Yeh. Kelly Cache Networks. In *Proc. of IEEE INFOCOM*, 2019.

[42] Peter Manohar and Jalani Williams. Lower Bounds for Caching with Delayed Hits. *arXiv preprint arXiv:2006.00376*, 2020.

[43] Valentina Martina, Michele Garetto, and Emilio Leonardi. A Unified Approach to The Performance Analysis of Caching Systems. In *Proc. of IEEE INFOCOM*, 2014.

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

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

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

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

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

[49] Nitish K Panigrahy, Jian Li, Don Towsley, and Christopher V Hollot. Network Cache Design under Stationary Requests: Exact Analysis and Poisson Approximation. *Computer Networks*, 180:107379, 2020.

[50] Nitish K Panigrahy, Jian Li, Faheem Zafari, Don Towsley, and Paul Yu. A TTL-based Approach for Content Placement in Edge Networks. In *EAI International Conference on Performance Evaluation Methodologies and Tools*, pages 1–21. Springer, 2021.

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

[52] D Shasha and T Johnson. 2Q: A Low Overhead High Performance Buffer Management Replacement Algorithm. In *Proc. of VLDB*, 1994.

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

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

[55] Edward S Tam. *Improving Cache Performance via Active Management*. University of Michigan, 1999.

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

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

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

[59] Gang Yan, Jian Li, and Don Towsley. Learning from Optimal Caching for Content Delivery. In *Proc. of ACM CoNEXT*, 2021.

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

A Supplementary Material

A.1 Analysis of Real Request Traces

In this subsection, we provide the detailed analysis of four production traces used in this paper, as discussed in Section 6.

The content popularity distribution (Figure 5(a)) shows that most traces follow approximately a Zipf distribution with the Zipf parameter α between 0.56 and 1.24.

The content inter-arrival time distribution (Figure 5(b)) - the distribution of the time between two consecutive request arrivals - further distinguishes these traces. It is clear that the requested contents in CDN-A have the largest variations since CDN-A contents have the smallest inter-arrival times, i.e., most contents are only requested over a shorter time period. In contrast, CDN-B, Bilibili and Wikipedia serve millions of different customers and hence exhibit largely independent requests with random inter-request times, which is consistent with the content popularity distribution.

A.2 Comparison with Offline Optimum Bélády

We also compare the performance in term of latency reduction to the offline optimum Bélády. From Figures 14 and 15,

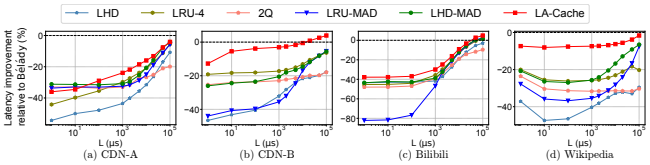


Figure 14: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to Bélády using a 256GB cache.

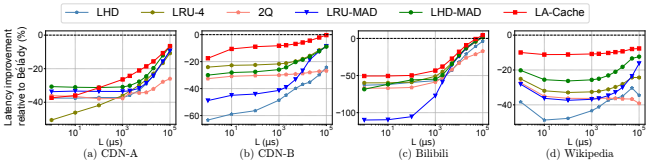


Figure 15: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to Bélády using a 512GB cache.

we observe again that LA-Cache outperforms other state-of-the-art algorithms, and importantly LA-Cache can outperform Bélády as the fetching latency increases. For example, LA-Cache outperforms Bélády when L is greater than 0.8 ms in CDN-B, and when L is greater than 60 ms in Bilibili.

Finally, we characterize the impact of cache size when compared with the offline optimum Bélády. From Figure 16, we observe that LA-Cache consistently outperforms LRU-MAD and LHD-MAD across a wide range of cache sizes. More interestingly, LA-Cache outperforms Bélády.

Impact of burstiness. Complementary to the results presented in Figure 12, the burstiness of each content and the corresponding latency improvement of this content due to the latency-aware caching algorithms with respect to the offline Bélády with 256GB and 512GB cache are presented in Figure 17 and Figure 18, respectively. Again, we observe that LA-Cache prioritizes more bursty requests compared to other state-of-the-art algorithms, which contributes to the latency reduction of LA-Cache.

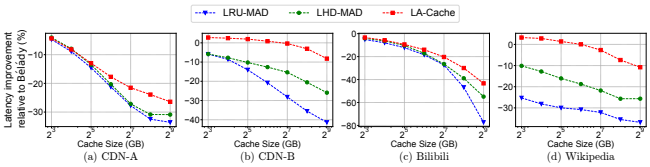


Figure 16: Comparison of the mean latency improvement between LA-Cache and state-of-the-art algorithms relative to Bélády as a function of cache sizes.

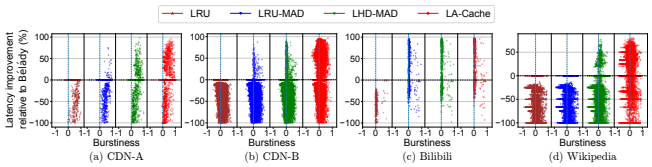


Figure 17: Improvement from the burstiness of content requests relative to the offline Béládyusing a 256GB cache.

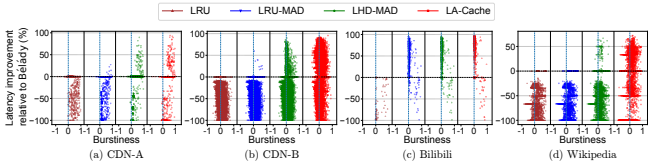


Figure 18: Improvement from the burstiness of content requests relative to the offline Bélády using a 512GB cache.

A.3 Hit Rate Comparison

Although we focus on designing latency-optimal caching in the presence of delayed hits, in which the goal of maximizing cache hits and the goal of minimizing latency are not equivalent (see Section 2.2), we argue that the latency improvements in turn also contribute to the cache hits performance. In most of existing works, the user-perceived latency upon cache hits is negligible, which is not true in the presence of "delayed hits" due to network latency. As a result, a content request results in three outcomes, i.e., miss, delayed hit, and true hit (see Introduction and Fig. 1). We observe that LA-Cache improves the true hit ratio up to 7% as shown in Figures 19 and 20, and "all hits" (true hit plus delayed hit) up to 9% as shown in Figures 21 and 22.

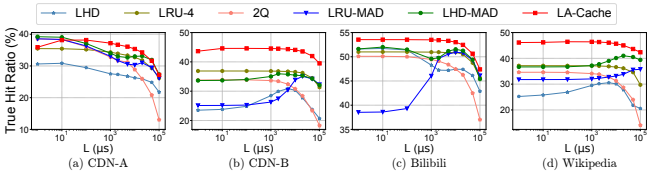


Figure 19: Comparison of true hits between LA-Cache and state-of-the-art algorithms using a 256GB cache.

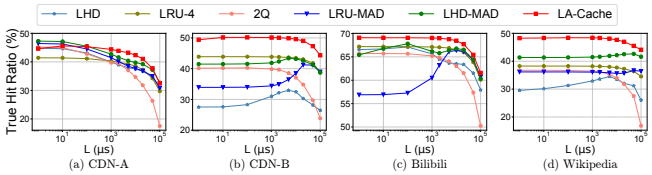


Figure 20: Comparison of true hits between LA-Cache and state-of-the-art algorithms using a 512GB cache.

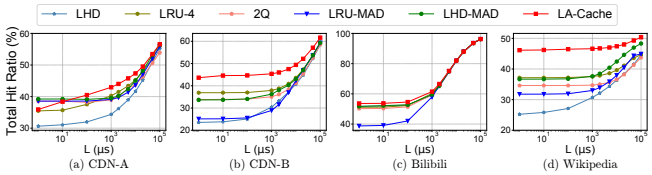


Figure 21: Comparison of all hits between LA-Cache and state-of-the-art algorithms using a 256GB cache.

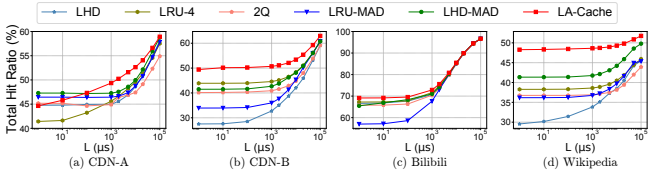


Figure 22: Comparison of all hits between LA-Cache and state-of-the-art algorithms using a 512GB cache.