Learning Relaxed Belady for Content Distribution Network Caching

Zhenyu Song
Princeton University

Daniel S. Berger
Microsoft Research & CMU

Kai Li
Princeton University

Wyatt Lloyd
Princeton University

Abstract

This paper presents a new approach for caching in CDNs that uses machine learning to approximate the Belady MIN (oracle) algorithm. To accomplish this complex task, we propose a CDN cache design called Learning Relaxed Belady (LRB) to mimic a Relaxed Belady algorithm, using the concept of Belady boundary. We also propose a metric called good decision ratio to help us make better design decisions. In addition, the paper addresses several challenges to build an end-to-end machine learning caching prototype, including how to gather training data, limit memory overhead, and have lightweight training and prediction.

We have implemented an LRB simulator and a prototype within Apache Traffic Server. Our simulation results with 6 production CDN traces show that LRB reduces WAN traffic compared to a typical production CDN cache design by 4–25%, and consistently outperform other state-of-the-art methods. Our evaluation of the LRB prototype shows its overhead is modest and it can be deployed on today's CDN servers.

1 Introduction

Content Distribution Networks (CDNs) deliver content through a network of caching servers to users to improve latency, bandwidth, and availability. CDNs delivered 56% of Internet traffic in 2017, with a predicted rise to 72% by 2022 [29]. Improving CDN caching servers can significantly improve the content delivery of the Internet.

A CDN cache sits between users and Wide-Area Networks (WANs). Whenever a user requests content that is not currently cached, the CDN must fetch this content across Internet Service Providers (ISPs) and WANs. While CDNs are paid for the bytes delivered to users, they need to pay for the bandwidth required to fetch cache misses. These bandwidth costs constitute an increasingly large factor in the operational cost of large CDNs [6, 14, 17, 41, 46, 66]. CDNs are thus seeking to minimize these costs, which are typically measured as *byte miss ratios*, i.e., the fraction of bytes requested by users that are not served from cache.

The algorithm that decides which objects are cached in each CDN server plays a key role in achieving a low byte miss ratio. Yet, the state-of-the-art algorithms used in CDN caching servers are heuristic-based variants of the Least-Recently-Used (LRU) algorithm (Section 2). The drawback of heuristics-based algorithms is that they typically work well for some access patterns and poorly for others. And, even with five decades of extensive study since caching was first proposed [84]—including using machine learning to adapt heuristics to different workloads—the fundamental limitation of heuristics remains. We still observe a large gap between the byte miss ratios of the state-of-the-art online cache replacement algorithms and Belady's offline MIN algorithm [16] on a range of production traces.

To bridge this gap, this paper presents a novel machine-learning (ML) approach that is fundamentally different from previous approaches to cache replacement. Our approach does not build on heuristics, nor try to optimize heuristics-based algorithms. Our approach is to approximate Belady's MIN (oracle) algorithm, using machine learning to find objects to evict based on past access patterns. Belady's algorithm always evicts the object with the furthest next request. A naive ML algorithm that imitates this behavior would incur prohibitively high computational cost. To overcome this challenge, our key insight is that it is sufficient to approximate a *relaxed Belady algorithm* that evicts an object whose next request is beyond a threshold but not necessarily the farthest in the future.

To set a proper threshold, we introduce the *Belady boundary*, the minimum time-to-next-request of objects evicted by Belady's MIN algorithm. We show that the byte miss ratio of the relaxed Belady algorithm using the Belady boundary as its threshold is close to that of Belady's MIN algorithm. Approximating relaxed Belady gives our system a much larger set of choices to aim for, which has two major consequences for our design. It allows our system to run predictions on a small sampled candidate set—e.g., 64—which dramatically reduces computational cost. And, it allows our system to quickly adapt to changes in the workload by gathering training data that includes the critical examples of objects that relaxed Belady would have selected for replacement.

Even with the insight of relaxed Belady, the design space of potential ML architectures (features, model, loss function, etc.) is enormous. Exploring this space using end-to-end metrics like byte miss ratio is prohibitively time-consuming because they require full simulations that take several hours for a single configuration. To enable our exploration of the ML design space we introduce an eviction decision quality metric, the *good decision ratio*, that evaluates if the next request of an evicted object is beyond the Belady boundary. The good decision ratio allows us to run a simulation only once to gather

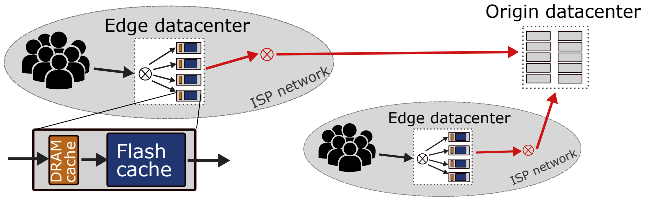


Figure 1: CDNs place servers in user proximity around the world (e.g., in a user's ISP). Incoming requests are sharded among several caching servers, which use combined DRAM and flash caches. Cache misses traverse expensive wide-area networks to retrieve data from the origin datacenter.

training data, prediction data, and learn the Belady boundary. Then we can use it to quickly evaluate the quality of decisions made by an ML architecture without running a full simulation. We use the good decision ratio to explore the design space for components of our ML architecture including its features, model, prediction target, and loss function among others.

These concepts enable us to design the Learning Relaxed Belady (LRB) cache, the first CDN cache that approximates Belady's algorithm. Using this approach in a practical system, however, requires us to address several systems challenges including controlling the computational overhead for ML training and prediction, limiting the memory overhead for training and prediction, how to gather online training data, and how to select candidates for evictions.

We have implemented an LRB simulator and an LRB prototype within Apache Traffic Server [1,2,10,43].[1] Our evaluation results using 6 production CDN traces show that LRB consistently performs better than state-of-the-art methods and reduces WAN traffic by 4–25% compared to a typically deployed CDN cache—i.e., using LRU replacement that is protected with a Bloom filter [22]. Our evaluation of the prototype shows that LRB runs at a similar speed to a heuristic-based design and its computational overhead and memory overhead are small.

2 Background and Motivation

In this section, we quantify the opportunities and challenges that arise in CDN caching. We also discuss the constraints on the design space of LRB.

2.1 The Challenge of Reducing WAN Traffic

A CDN user request is directed to a nearby CDN server (e.g., using DNS or anycast [33]), which caches popular web and media objects. Each cache miss has to be fetched across the wide-area network (WAN) (Figure 1). As WAN bandwidth needs to be leased (e.g., from tier 1 ISPs), CDNs seek to reduce WAN traffic, which corresponds to minimizing their cache's byte miss ratio.

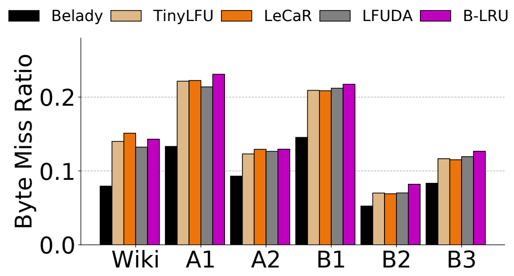


Figure 2: Simulated byte miss ratios for Belady and the top-performaning policies from Section 6 for six CDN production traces and a 1 TB flash cache size. There is a large gap of 25–40% between Belady and all other policies.

CDN caching working sets are massive compared to available cache space. To reduce bandwidth, a CDN deploys DRAM and flash caches in each edge server.[2] A commercial CDN serves content on behalf of thousands of content providers, which are often large web services themselves. This leads to a massive working set, which is typically sharded across the (tens of) servers in an edge cluster [25,77]. Even after sharding traffic, an individual cache serves traffic with a distinct number of bytes much larger than its capacity. Consequently, there exists significant competition for cache space.

There are significant opportunities to improve byte miss ratios. CDN traffic has a variety of unique properties compared to other caching workloads. For example, a striking property observed in prior work [63] is that around 75% of all objects do not receive a second request within two days (so-called "one-hit-wonders"). If these objects are allowed to enter the cache, the vast majority of cache space is wasted. Therefore, major CDNs deploy B-LRU, an LRU-eviction policy using a Bloom filter [63,67,74] to prevent one-hit-wonder objects from being admitted to the cache.

We quantify the opportunity to improve byte miss ratios over B-LRU. To this end, we use 6 production traces from 3 different CDNs, which are sharded at the granularity of a single SSD (around 1–2 TB). Figure 2 compares the byte miss ratios of B-LRU and the other top-performing policies from Section 6 to Belady's MIN [16], and we find that there remains a gap of 25–40%.

Exploiting miss ratio opportunities is challenging in practice. The key challenge is that workloads change rapidly over time and differ significantly between servers and geographic locations. These changes occur due to load balancing different types of content, e.g., web traffic is more popular in the morning and video more popular at night. Due to these rapid changes and breadth of different access patterns, prior

[1] The source code of our simulator and prototype alongside with documentation and the Wikipedia trace used in our evaluation is available at `https://github.com/sunnyszy/lrb` .

[2] CDNs typically prefer flash storage over HDDs as cache reads lead to random reads with very high IOPS requirements. While some CDNs have hybrid deployments, many CDNs rely entirely on flash [46,67,74,78].

	PT-1	PD-2	ET-1	ET-2	CET-1	CET-2
Mean CPU	5%	3%	6%	16%	7%	18%
Peak CPU	19%	12%	10%	24%	13%	30%

Table 1: Mean and peak CPU load in Wikipedia's production deployment CDN across 6 datacenters in three different timezones, for March 2019. Peak CPU load is below 30%.

caching policies (Section 7) only achieve marginal improvements over B-LRU (Section 6). In fact, many recent caching policies lead only to gains on some traces with certain cache size configurations.

In conclusion, there is significant demand to realize caching policies that can automatically adapt to a geographic location's workload and to changes in request patterns over time. Machine learning is well suited to achieving this goal. But, leveraging machine learning for CDN caching requires us to overcome many challenges we describe in Section 4.

2.2 Opportunity and Requirements

Deployment opportunity. We find that today's CDN caches typically have spare processing capacity. Table 1 shows the CPU load in six production deployments of Wikipedia's CDN for March 2019. We see that, on average, there is 90% spare processing capacity. Even under peak load, there is still 70% CPU capacity available, as disk and network bandwidth are frequent bottlenecks. Thus, the opportunity in current deployment is that we can use this excess processing capacity as part of more sophisticated caching algorithms.

Deployment requirements. There are three key constraints when deploying on existing CDN caching servers.

- **Moderate memory overhead** of a few GBs but not 10s of GBs because large quantities are not available [19].
- **Not require TPUs or GPUs** because these are not currently deployed in CDN servers [41,63].
- **Handle tens of thousands of requests per second** because this is the request rate seen at CDN caches [19].

3 Approximating Belady's MIN Algorithm

In order to approximate Belady's MIN algorithm in the design of an ML-based cache, this section introduces the relaxed Belady algorithm, Belady boundary, and good decision ratio.

3.1 Relaxed Belady Algorithm

It is difficult for an ML predictor to directly approximate Belady's MIN algorithm for two reasons. First, the cost of running predictions for all objects in the cache can be prohibitively high. Second, in order to find the object with the farthest next request, an ML predictor needs to predict the time to next request of all objects in the cache accurately.

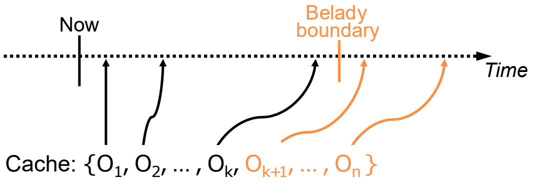


Figure 3: The relaxed Belady algorithm partitions objects into two sets based on their next requests. The next requests of $O_{k+1},\ldots,O_n$ are beyond the threshold (Belady boundary).

To overcome these two challenges, we define the *relaxed Belady algorithm*, a variant of Belady's MIN, that randomly evicts an object whose next request is beyond a specific threshold, as shown in Figure 3. The algorithm partitions the objects of a cache into two sets each time it needs to make an eviction decision: objects whose next requests are within the threshold (set 1) and objects whose next requests are beyond the threshold (set 2). If set 2 is not empty, the algorithm randomly evicts an object from this set. If set 2 is empty, the algorithm reverts to the classical Belady among object in set 1.

The relaxed Belady algorithm has two important properties. First, if the threshold is far, its byte miss ratio will be close to that of Belady's MIN. Second, since the algorithm can evict any of the objects in set 2, not necessarily the object with farthest distance, there will be many eviction choices in the cache each time it needs to evict an object.

These properties greatly reduce the requirements for an ML predictor. Instead of predicting next requests for all objects in the cache, the ML predictor can run predictions on a small sample of candidates in the cache for an eviction as long as the sample includes objects from set 2. This allows a dramatic reduction of computational cost. In addition, it reduces the required precision of the ML predictor. It only needs to find an object whose next request is beyond the threshold instead of the farthest. This greatly reduces the memory overhead when gathering training data: instead of having to track objects indefinitely, it is sufficient to track objects until they are re-requested or cross the threshold.

Shorter thresholds thus make the task of the ML predictor more tractable. While longer thresholds move the byte miss ratio of relaxed Belady closer to Belady's MIN. It is thus important to find the proper threshold.

3.2 Belady Boundary

To systematically choose a threshold for the relaxed Belady algorithm, we introduce the *Belady boundary* as the minimum of time to next request for all evicted objects by Belady's MIN algorithm. It is intuitively meaningful—Belady's MIN kept all objects with a next request less than this boundary in the cache—but requires future information to compute. In practice, we assume that the Belady boundary is approximately stationary. This allows us to approximate the Belady boundary by computing the minimum of the next requests of

Cache size (GB)	64	128	256	512	1024
Boundary (x cache size)	3	2	2	2	2
Objects beyond boundary	19%	19%	16%	12%	10%
Increase in byte miss ratio over Belady	10%	13%	12%	11%	9%

Table 2: Comparison of the relaxed Belady algorithm to Belady MIN for the Wikipedia trace with different cache sizes.

all evicted objects by the Belady MIN algorithm during the machine learning warmup period.

Table 2 shows how the Belady boundary affects the byte miss ratio of relaxed Belady on the Wikipedia trace. We find that relaxed Belady applies random eviction to a set of objects, e.g., 19% for a 64 GB cache. While this Belady boundary enables efficient implementations, it comes at the cost of 9-13% more misses. Compared to the 25%–40% gap between state-of-the-art online algorithms and Belady's MIN (Figure 2), this increase seems acceptable.

Next, we show how to use the Belady boundary to establish a decision quality metric, which is used to make design decisions throughout Section 4.

3.3 Good Decision Ratio

To design an algorithm that makes better decisions we need to determine the quality of individual decisions. End-to-end metrics like byte miss ratio, however, reflect the aggregated quality of many decisions. When an algorithm has a byte miss ratio higher than Belady we know it made some bad decisions, but we cannot determine which of its decisions were bad. The individual misses that comprise byte miss ratio are similarly unhelpful because they are also the result of many earlier decisions.

Our eviction decision metric is defined with respect to the relaxed Belady algorithm with the Belady boundary as its threshold. Evicting an object is a *good decision* if the next request of the object is beyond the Belady boundary. It is a *bad decision* if its next request is within the Belady boundary.

We find that an algorithm's *good decision ratio*—i.e., # good decisions / # total eviction decisions—correlates strongly with the byte miss ratio it ultimately achieves (Section 6.4). This metric plays a key part in the design of LRB.

4 Design of Learning Relaxed Belady Cache

This section presents the design details of LRB, which uses ML to imitate the relaxed Belady algorithm. Accomplishing this requires simultaneously addressing two previously unsolved problems:

- How to design an ML approach that decreases byte miss ratio relative to state-of-the-art heuristics?
- How to build a practical system with that approach?

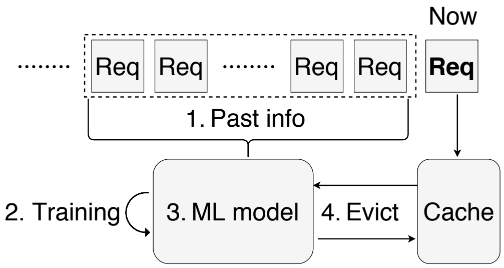


Figure 4: General architecture that uses ML for caching with 4 key design issues: 1) what past information to use for ML, 2) how to generate online training datasets, 3) what ML model to use, and 4) how to select eviction candidates.

Each problem introduces several challenges in isolation, simultaneously addressing them additionally requires us to balance their often-competing goals.

Figure 4 presents a general architecture that uses ML to make eviction decisions. We identify four key design issues:

(1) Past information. The first design issue is determining how much and what past information to use. More data improve training quality, but we need to limit the information in order to build a practical system, as higher memory overhead leads to less memory that can be used to cache objects.

(2) Training data. The second design issue is how to use past information for training. As workloads vary over time the model must be periodically retrained on new data. Thus, a practical system must be able to dynamically generate training datasets with proper labels.

(3) ML architecture. The third design issue is selecting a machine learning architecture that makes good decisions. This includes selecting features, the prediction target, and the loss function as well as how to translate predictions into eviction decisions. In addition, these decisions need to be compatible with available hardware resources to build a practical system.

(4) Eviction candidate selection. The final design issue is how to select eviction candidates. Although an approximation of the relaxed Belady algorithm can evict any object whose next request is beyond the Belady boundary and there are many eviction choices, we still need a method to select a small set of candidates that includes such objects with a high probability.

The rest of this section describes the design decisions of LRB for each of these design issues. Our general approach is to guide our design using the good decision ratio metric defined in Section 3.

4.1 Past information

LRB keeps information about an object only when its most recent request is within the *sliding memory window*. The information within the sliding memory window is used for training and prediction.

Setting the size of the sliding memory window is important to the performance of LRB. If the sliding memory window is too short, LRB will not have enough training data to produce a good model and may not have the information it needs during prediction to make an accurate prediction. If the sliding memory window is too long, it may take too much space away from cached data.

LRB uses each trace's validation prefix to set the sliding memory window hyperparameter. For small cache sizes, the validation prefix is long enough to derive the "optimal" sliding memory window, which achieves the highest good decision ratio. For large cache sizes, we use a least-squares regression line fit to the relationship between small cache sizes and their optimal sliding memory window parameters.

We remark that, at the beginning of a trace (or during initial deployment), LRB requires training data to build an initial model. Thus, our implementation uses LRU as a fallback until sufficient training data is available.

4.2 Training Data

Acquiring training data for our problem is challenging because the features and corresponding label exist at widely disparate times. Consider an object that is requested once and then not requested again until 5 hours later. The features of the object exist and vary at all times in the 5-hour range. The label—i.e., what we want to predict—does not exist until the end of that range, when we know the "future" by waiting until it is the present. To address the time disparity, LRB decouples generation of unlabeled training data from labeling that data.

Unlabeled training data generation. To generate unlabeled training data LRB periodically takes a random sample of objects in the sliding memory window and then records the current features of those objects. We choose to randomly sample over objects instead of over requests to avoid biasing the training data towards popular objects with many requests. Such popular objects should be represented in the training data so the model learns not to evict them. Training data for less popular objects, however, is more important because they include the objects beyond the Belady boundary—the good eviction decisions we want our model to learn. We choose to randomly sample over all objects in the sliding memory window instead of only those currently in the cache as cached objects are similarly biased.

Labeling training data. LRB uses two methods for assigning labels to training data. The first is to wait until an object is requested again. When this happens, we know the "future" and use that to determine the label. Some objects, however, will not be requested until far—e.g., 5 days—in the future.

Waiting until such objects are requested again would introduce many problems. First, it would require excessive memory overhead as we maintain features for many objects that were requested a potentially very long time ago. Second, it would make some of the training data excessively old, e.g., 5 days old. Finally, it would never include training data for objects that are never requested again—which are a significant portion of the good decisions we want our method to learn.

LRB's second method for labeling training data avoids these problems by leveraging an insight related to the Belady boundary: all objects that are not requested again for at least a boundary amount of time are equivalent good eviction decisions. Thus, once the time since last request for an object exceeds the Belady boundary we can safely label it. LRB uses this second labeling method when an object falls out of the sliding memory window.

4.3 ML Architecture

This subsection describes the components of LRB's ML architecture. For each component, we describe our choice, describe its rationale, and then use the good decision ratio to explore the design space. This exploration is complicated by the fact that each component influences the others—feature set A may be better for model X while feature set B may be better for model Y. Even with our good decision ratio doing a full exploration of all possible combinations is still prohibitive. Instead, we fix each of the other components on what we ultimately select in LRB's design and then vary the specific component.

4.3.1 Features: Deltas, EDCs, and Static

LRB uses three different types of features: deltas, exponentially decayed counters, and static features. The rationale for these features is to provide a superset of the information used by previous heuristics. Because our model learns the weight for different features adding more features should not harm its accuracy. Thus, the primary tradeoff for features is weighing how much they help the model against the overhead they incur. To minimize the overhead of the features we strive to make storing and updating them efficient (Section 5).

Deltas. Deltas indicate the amount of time between consecutive requests to an object. Delta$_1$ indicates the amount of time since an object was last requested. Delta$_2$ indicates the time in between an object's previous two requests and so on, i.e., delta$_n$ is the amount of time between an object's n^{th} and $(n-1)^{th}$ previous requests. If an object has been requested only n times, then above delta$_n$ are ∞. We include deltas as features because they subsume the information used by many successful heuristics. For instance, LRU uses delta$_1$, LRU-K uses the sum of delta$_1$ to delta$_k$, and S4LRU uses a combination of delta$_1$ to delta$_4$.

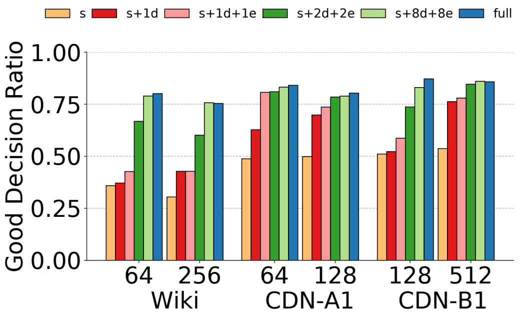


Figure 5: Good decision ratios for accumulating features for LRB on three traces at two cache sizes. LRB uses static features, 32 deltas, and 10 EDCs (full).

Exponentially decayed counters (EDCs). EDCs track an approximate per-object request count over longer time periods, where keeping exact counts would require excessive memory overhead. Each EDC_i is initialized to 0. Whenever there is a request to the object, we first update Delta_1 and then $C_i = 1 + C_i \times 2^{-\text{Delta}_1/2^{9+i}}$. Unlike Delta_1, C_i will not be updated until another request arrives for the same object. The EDCs with larger values of i cover larger periods, e.g., an object that was highly popular 1 million requests ago but is now not popular would have a low EDC_1 but still have a high EDC_{10}.

The exact EDC equation is motivated by prior observation in block storage caching [57] and video popularity prediction [79], where EDCs accurately approximate the decay rate of object popularities. Tracking long-term popularity is used by many other algorithms such as LFU variants [11,75], Hyperbolic [21], and LRFU [57] (which uses a single EDC). LRB uses multiple EDCs with different decay constants to capture the request rate to the object over multiple time horizons.

Static features. Static features include additional unchanging information about an object. They include the size of the object and its type—e.g., video on demand segment, live video segment, image. We include static features because they are available, require little memory overhead, and intuitively correlate with different access patterns. For instance, live video segments might be useful in a cache for only a short period, whereas video on demand segments might be useful in the cache for much longer.

Feature effectiveness, number of deltas, number of EDCs. To determine if this set of features is effective for our ML architecture we evaluate them using the good decision ratio in the validation prefix of our traces at many cache sizes. We also use the good decision ratio to determine how many deltas we store and how many EDCs we store. Figure 5 shows the results for an accumulation of static features, Delta_1, EDC_1, Delta_2 and EDC_2, Deltas 3–8 and EDCs 3—8, and then Deltas 9–32 and EDCs 9–10 (full). As expected, the addition of more features improves the good decision ratio but has diminishing returns. Results on the other three traces and at other cache sizes are not shown but are similar.

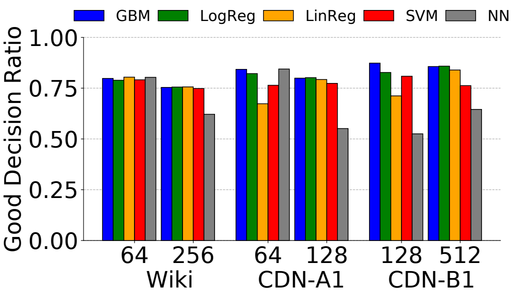


Figure 6: Good decision ratios for models. LRB uses GBM as it robustly achieves high good decision ratios on all traces/cache sizes.

The number of deltas and EDCs is each a tradeoff between giving the ML architecture more information to use for decisions and storage overhead. We settle on 32 deltas and 10 EDCs because both appear to be past the point of diminishing returns and thus we are not disadvantaging our ML architecture by stopping there. We go as high as these numbers because of the efficiency of our feature storage and our ability to control memory overhead with the sliding memory window.

4.3.2 Model: Gradient Boosting Machines

LRB uses Gradient Boosting Machines [39] (GBM) for its model, which outperform all other models we explored and are highly efficient on CPUs. We were inspired to explore GBM based on their success in many other domains with tabular data [23,30,42,61,62,72,80]. We also explored linear regression, logistic regression, support-vector machines, and a shallow neural network with 2 layers and 16 hidden nodes. Figure 6 shows the good decision ratios of the different models on three traces at two cache sizes. Results on the other traces and other cache sizes are similar and not shown. GBM robustly achieve a high good decision ratio. Additionally, GBM do not require feature normalization and can handle missing values efficiently, which are common as objects have a varying number of Deltas. In addition, GBM are highly efficient to train and use for prediction. On typical CDN server hardware, we can train our model in 300 ms. And, we can run prediction on 64 eviction candidates in 30 μs.

4.3.3 Prediction Target: log (time-to-next-request)

LRB uses regression to predict the time-to-next-request for objects that are requested within the sliding memory window. Regression enables LRB to rank these objects (as relaxed Belady) and also serves as a confidence measure for distance to the Belady boundary. Specifically, LRB predicts the log(time-to-next-request) for an object as we primarily care about which side of the boundary the next request is on. If the Belady boundary is 1M (measured by the number of requests), then the difference between predicting 100K and 2M

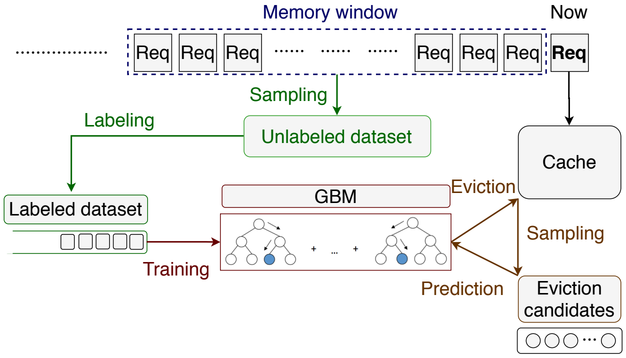


Figure 7: Detailed architecture overview of LRB.

is larger than the difference between 2M and 3M. For objects that are not requested within the sliding memory window, LRB assigns a label as 2× the window size.

We also explored predicting the time from the last request to the next request, binary classification relative to a boundary, and unmodified time-to-next-request. We chose log(time-to-next-request) because it achieved a higher good decision ratio than all of these alternatives.

4.3.4 Loss Function: L2

LRB uses L2 loss (mean square error). We also tried all eight loss functions in the LightGBM library [54]. We chose L2 loss because it achieved the highest good decision ratio.

4.3.5 Training dataset Size: 128K

LRB trains a new model once it accumulates a dataset of 128K labeled training samples. We also explored training dataset sizes that were much smaller than 128K and up to 512K. We found that good decision ratio increases with dataset size but has diminishing returns. We choose 128K because larger dataset sizes further increase training time and overhead without a noticeable improvement in good decision ratio.

4.4 Eviction Candidate Selection.

LRB randomly samples cached objects to gain eviction candidates and runs a batch prediction for all candidates. LRB evicts the candidate whose predicted next request distance is the longest.

We determine the choice for our random sample size—64 samples—using the good decision ratio. We find that 64 samples are past the point of diminishing returns, and thus choosing it does not disadvantage our ML architecture. It is also still low enough for low overhead—prediction on 64 samples takes LRB only 30 μs.

4.5 Putting All Together

Putting our design decisions together with the general architecture (Figure 4) from the beginning of the section, we have the complete design of LRB as shown in Figure 7.

LRB learns from the requested objects in a sliding memory window whose length approximates the Belady boundary. The features (deltas, EDCs, static) of these objects are stored in a compact data structure that is updated as new requests arrive, moving the sliding memory window forward.

A sampling process continuously samples data from this data structure, generating an unlabeled dataset. A separate labeling process continuously labels that data. When the labeled dataset is full (128K examples), LRB starts training a GBM model, and empties the labeled dataset. After that, whenever the labeled dataset is full again, LRB repeats the process and replaces the old model with the new one. If a current request is a cache miss and needs to evict an object, LRB randomly draws $k = 64$ eviction candidates and runs GBM predictions on them. LRB evict the candidate with the farthest predicted next access time.

5 Implementation

We have implemented a LRB prototype within Apache Traffic Server (ATS). We have also implemented a LRB simulator in order to compare with a wide range of other algorithms. The two implementations share the same code and data structures as a C++ library with about 1400 lines of code.

5.1 Prototype

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

To implement LRB, we have replaced the lookup data structures for ATS's disk cache with the architecture described in Section 4.5. We have also changed ATS to make eviction decisions asynchronously by scheduling cache admissions in a lock-free queue. A known challenge when implementing SSD-based caches is write amplification due to random writes [36, 58]. In order to implement LRB, a production implementation would rely on a flash abstraction layer such as RIPQ [78] or Pannier [58]. Unfortunately, we have no access to such flash abstraction layers (e.g., RIPQ is proprietary to Facebook). Our implementation thus emulates the workings of a flash abstraction layer, reading at random offsets and writing sequentially to the SSD.

As the memory cache is typically small, which has a negligible impact on the byte miss ratio [19], we leave this part of

Object class: # past requests	1	2	3	4	5	6- 12	13- 31	$\geq$32
Obj fraction (%)	36	11	5	3	2	1	<1	<1
Overhead (bytes)	25	94	98	102	106	$\leq$134	$\leq$210	214

Table 3: LRB's memory overhead depends on an object's number of past requests, which we call the object's class.

ATS unchanged. In total, the changes to ATS, excluding the LRB library, amount to fewer than 100 lines of code.

5.2 Simulator

We have implemented an LRB simulator based on the AdaptSize simulator [19]. Besides LRB, our simulator implements 14 state-of-the-art caching algorithms.[3] These include classic and learning-based algorithms. The classic algorithms include LRUK [68], LFUDA [11, 75], S4LRU [46], LRU, FIFO, Hyperbolic [21], GDSF [11] and GDWheel [60]. The learning-based algorithms include an adaptive version of TinyLFU [34] (which subsumes static TinyLFU [35]), LeCaR [81], UCB [31] (reinforcement learning), LFO [17] (supervised learning), LHD [15] and AdaptSize [19]. These 14 algorithms and the simulation environment take about 11K lines of C++. For TinyLFU, LFO, LHD, and AdaptSize, we verified parameter configurations match the authors'.

5.3 Optimizations

We have implemented two main optimizations to reduce overhead and improve performance.

Computationally efficient feature updating. To minimize the overhead of maintaining features we seek to update them rarely and for necessary updates to be efficient. LRB accomplishes this by favoring *time-invariant* features when possible, i.e., features that need only be calculated once. Static features by definition fall in this category. We choose to make deltas relative to the time in between consecutive requests instead of the time between n requests ago and the most recent request because this makes all deltas other than delta_1 time-invariant. A new request to an object shifts delta_n to delta_{n+1}. The overhead to compute delta_1 is low and we make EDC updates efficient using a lookup table with precomputed decay rates.[4]

Overall, these optimizations result in a constant and small feature update overhead per request. LRB updates an object's feature only at the times when it is requested and when it is sampled as an eviction candidate.

We compress our features to minimize their memory overhead. Our compression is based on the predominance of "one-hit wonders" (Section 2). Consequently, we treat objects for which we registered only a single request separately. Since such an object was requested only once in the sliding memory window, there is no recency information and EDC values are all 0. Instead of keeping this redundant information, we store only the key, object size, object type, the last request time, and a pointer to a struct. This reduces the memory overhead for single-request objects to 25 B on the Wikipedia trace. When objects receive more requests, we populate the struct with only as many deltas as they have past requests. This further reduces the memory overhead for all objects with fewer requests than the maximum number of deltas, 32. The struct also contains the EDCs, which are compressed to a single float [57]. Table 3 shows the distribution and overhead of objects with different number of requests.

6 Evaluation

This section evaluates our LRB prototype. We additionally use simulations to compare LRB to a wide range of state-of-the-art caching algorithms. We aim at answering the following questions:

- What is the WAN traffic reduction using our LRB prototype compared to the ATS production system (Sec 6.2)?
- What is the overhead of our LRB prototype compared to CDN production systems (Sec 6.3)?
- How does the byte miss ratio of LRB compare to state-of-the-art research systems on a wide range of CDN traces and cache sizes (Sec 6.4)?
- What is the gap between Belady and LRB (Sec 6.5)?
- Can LRB be improved using a longer validation prefix to select the sliding memory window (Sec 6.6)?

6.1 Experimental Methodology

This subsection describes the traces, competing cache algorithms, warm-up, and the testbed in our experiments.

Traces Our evaluation uses CDN traces from three CDNs, two of which chose to remain anonymous.

- **Wikipedia**: A trace collected on a west-coast node, serving photos and other media content for Wikipedia pages.
- **CDN-A**: Two traces (A1 and A2) collected from two nodes on different continents serving a mixture of web traffic for many different content providers.
- **CDN-B**: Three traces (B1–B3) collected from nodes in the same metropolitan area, each serving a different mixture of web and video traffic for many different content providers.

Table 4 summarizes key properties of the six traces.

State-of-the-art algorithms. In the prototype experiments, we compare our LRB implementation (Section 5) to unmodified Apache Traffic Server (ATS, version 8.0.3), which approximates a FIFO eviction policy. Our simulations compare

[3] For Adaptive-TinyLFU, we integrate the original author's implementation (`https://github.com/ben-manes/caffeine`) into our simulator.

[4] The size of the table is $W/2^{10} \times 4\,\text{B}$ where W is the sliding memory window size—e.g., if W is 2^{28} (256 million), the table is $2^{18} \times 4\,\text{B} = 1\,\text{MB}$.

		Wikipedia	CDN-A1	CDN-A2	CDN-B1	CDN-B2	CDN-B3
Duration (Days)		14	8	5	9	9	9
Total Requests (Millions)		2,800	453	410	1,832	2,345	1,986
Unique Obj Requested (Millions)		37	89	118	130	132	116
Total Bytes Requested (TB)		90	156	151	638	525	575
Unique Bytes Requested (TB)		6	65	17	117	66	104
Warmup Requests (Millions)		2,400	200	200	1,000	1,000	1,000
Request Obj Size	Mean (KB)	33	644	155	460	244	351
	Max (MB)	1,218	1,483	1,648	1,024	1,024	1,024

Table 4: Summary of the six production traces that are used throughout our evaluation spanning three different CDNs.

LRB with 14 state-of-the-art caching algorithms (Section 5.2). In addition, our simulations include Belady's MIN [16] and relaxed Belady (Section 3) as benchmarks.

Testbed. Our prototype testbed consists of three Google cloud VMs acting as a client, CDN caching server, and backend/origin server, respectively. The VMs are n1-standard-64 VMs with 64 VCPUs, 240 GB of DRAM. To maximize SSD throughput, we use eight local 375 GB NVMe flash drives and combine them into one logical drive using software raid.

Clients are emulated using our C++ implementation (≈ 200 LOC), which uses 1024 client threads. The backend/origin server uses our own concurrent C++ implementation (≈ 150 LOC). This emulation method can saturate the network bandwidth between client and caching server. The clients replay requests in a closed loop to stress the system being tested and accelerate evaluation.

In Section 6.3, clients send requests in an open loop, using the original trace timestamps to closely emulate production workloads for latency measurements. Both unmodified ATS and LRB use a 1 TB flash cache size. Our simulations are based on the request order following the request timestamps and a range of different cache sizes.

To emulate network RTTs [19], we introduce around 10 ms and 100 ms of delays to the link between client and proxy and the link between origin and proxy respectively.

Metadata overhead. Different algorithms may have different metadata overheads. For fairness, all algorithms except Adaptive-TinyLFU [5] in the experiments have deducted their metadata overheads from corresponding cache sizes in all experiments. For example, if an experiment is for a 64 GB cache, an algorithm with 2 GB of metadata overhead will use 62 GB to store its data.

Validation and warmup trace sections. The first 20% of every trace is used as a "validation" section where LRB tunes its hyperparameters (Section 4). Each experiment uses a warmup during which no metrics are recorded. The warmup section is always longer than the validation section and is defined by the time by which every algorithm has achieved a stable byte miss ratio. Table 4 lists each trace's warmup section.

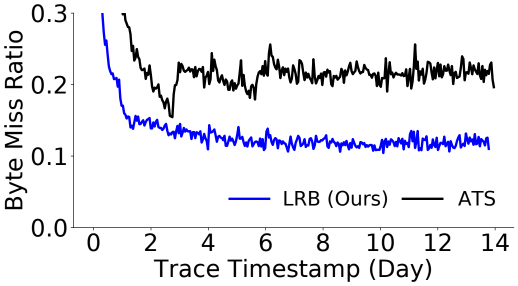


Figure 8: The byte miss ratios of LRB and unmodified ATS for the Wikipedia trace using a 1 TB cache size.

6.2 WAN Traffic Reduction of LRB Prototype

Figure 8 compares the byte miss ratios of LRB and unmodified ATS for a 1 TB cache using the Wikipedia trace. LRB achieves a better overall byte miss ratio than ATS. We observe that LRB reaches a lower miss ratio than ATS within the first day, and LRB continues to improve as it obtains more training data. Throughout the experiment, LRB's miss ratio is more stable than ATS's. We wait for both to stabilize and measure the WAN bandwidth consumption on days 12-14. LRB reduces the average bandwidth by 44% over ATS. LRB also reduces the 95th percentile bandwidth (the basis of some CDN traffic contracts [6]) by 43% over ATS.

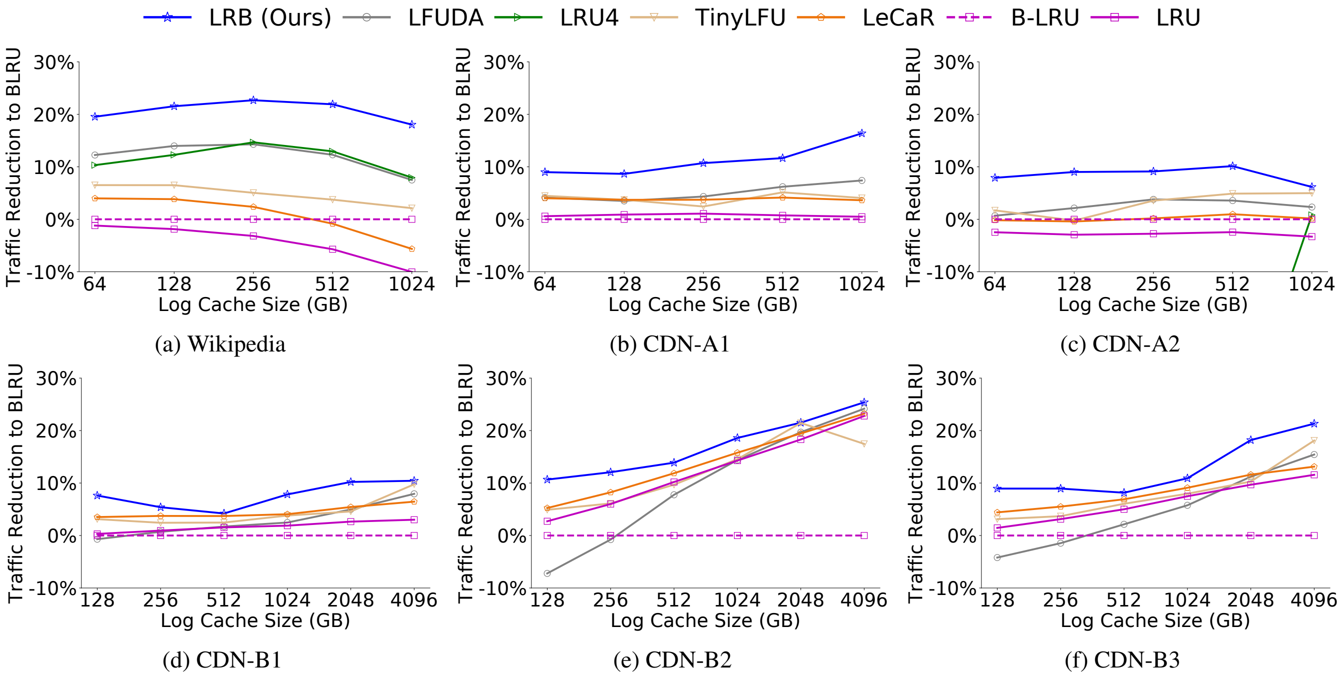
6.3 Implementation Overhead

Table 5 compares the overhead of LRB against unmodified ATS. We measure throughput, CPU, and memory utilization at peak throughput ("max" experiment). Note that Table 5 quantifies overheads for the Wikipedia trace, which is pes-

Metric	**Experiment**	**ATS**	**LRB**
Throughput	max	11.66 Gbps	11.72 Gbps
Peak CPU	max	9%	16%
Peak Mem	max	39 GB	36 GB
P90 Latency	normal	110 ms	72 ms
P99 Latency	normal	295 ms	295 ms
Obj Misses	normal	5.7%	2.6%

Table 5: Resource usage for ATS and LRB in throughput-bound (max) and production-speed (normal) experiments.

[5] Our experiments used an existing implementation of Adaptive-TinyLFU simulator which does not provide overhead information.



(a) Wikipedia

(b) CDN-A1

(c) CDN-A2

(d) CDN-B1

(e) CDN-B2

(f) CDN-B3

Figure 9: WAN traffic reduction over B-LRU at varying caches for Belady, LRB, and the six best state-of-the-art algorithms. LRB typically provides 4–25% WAN traffic savings over B-LRU.

simistic as all object sizes in all other traces are at least 5× larger, which results in significantly less overhead.

LRB has no measurable throughput overhead but its peak CPU utilization increases to 16% from 9% for ATS and 12% for B-LRU. We remark that even the busiest production cluster at Wikipedia ("CET-2" in Table 1 in Section 2) has sufficient CPU headroom.

We measure the number of object misses (which weights requests equally) when replaying the Wikipedia trace using its original timestamps ("normal" experiment in Table 5). LRB has less than half as many object misses as ATS. This miss reduction allows LRB to improve the 90th percentile latency (P90) by 35% compared to ATS. At the 99th percentile (P99), LRB achieves a similar latency to ATS because the origin server latency dominates.

Cache size	Wiki	A1	A2	B1	B2	B3
64 GB	3.0%	1.0%	1.7%	-	-	-
128 GB	2.1%	0.6%	1.4%	0.9%	0.8%	1.1%
256 GB	1.4%	0.5%	1.2%	0.8%	0.5%	0.6%
512 GB	1.0%	0.4%	1.0%	0.6%	0.4%	0.5%
1 TB	0.6%	0.3%	0.7%	0.5%	0.4%	0.4%
2 TB	-	-	-	0.4%	0.3%	0.3%
4 TB	-	-	-	0.3%	0.3%	0.3%

Table 6: Fraction of space allocated to metadata for LRB.

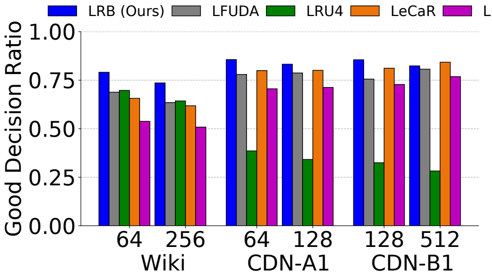


Figure 10: Good decision ratios for different algorithms. Good decision ratio correlates strongly with byte miss ratio.

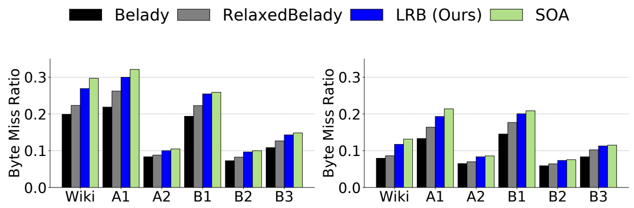
As LRB's memory overhead depends on the cache size and average object size in a trace, Table 6 measures the peak memory overhead for all traces and all evaluated cache sizes. Across all configurations, LRB always uses less than 3% of the cache size to store LRB metadata. While this overhead reduces the effective cache size, the small loss in byte miss ratio is more than offset by LRB's significant miss ratio improvements.

We believe these experimental observations show that LRB is a practical design for today's CDNs and that it can be deployed on existing CDN hardware.

6.4 LRB vs. State-of-the-art Algorithms

We compare the byte miss ratio of LRB to 14 state-of-the-art caching algorithms using simulations with a wide range of cache sizes using the six different traces.

Of the 14 algorithms, nine algorithms (FIFO, Hyperbolic, GDSF, GDWheel, UCB, LFO, AdaptSize, S4LRU, and LHD)



(a) 256 GB

(b) 1 TB

Figure 11: Comparison of byte miss ratios for Belady, relaxed Belady, LRB, and the best state-of-the-art (SOA) policy.

achieve a low byte miss ratio on at least one CDN traces. To improve readability, we show only the five best-performing algorithms in the following figures (TinyLFU, LeCaR, LRUK, LFUDA, and LRU). Figure 9 shows the wide-area network traffic reductions of each of these algorithms relative to B-LRU, with different cache sizes using the six traces.

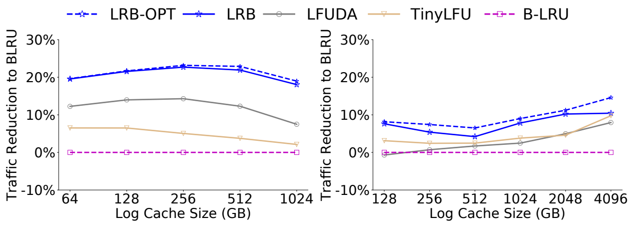
LRB robustly outperforms the best state-of-the-art algorithms. It achieves the lowest byte miss ratio for all 33 CDN trace and cache size combinations. Overall, LRB reduces WAN traffic by 4–25% on average.

Note that LRB's WAN traffic reduction does not generally decrease with larger cache sizes. For example, on CDN-B2 the traffic reduction steadily increases between 128 GB and 4 TB. Our interpretation is that these traces span a wide range of different CDN request patterns. On average (across all cache sizes), LRB reduces WAN traffic by over 13% compared to B-LRU. Additionally, LRB is robust across all traces whereas no prior caching algorithm consistently improves the performance across traces and cache sizes. For example, LRU4 is the best on Wikipedia, but among the worst on other traces. LFUDA does well on Wikipedia, CDN-A1, but poorly on CDB-B3. TinyLFU does well on CDN-B3 and CDN-A2, but not on Wikipedia. These results indicate that heuristic-based algorithms work well with certain patterns and poorly with others.

To further understand where LRB's improvement comes from, Figure 10 shows the good decision ratio of LRB, the three best-performing state-of-the-art algorithms, and LRU. TinyLFU is not included as its implementation does not allow us to evaluate individual decisions. LRB achieves 74–86% good decision ratios, which are the highest for all but one of six combinations. This implies that LRB learns a better workload representation and is able to leverage its model to make good eviction decisions. Overall, we find that the good decision ratio of an algorithm strongly correlates with its byte miss ratio. One exception is CDN-B1 512 GB, where LeCaR has a higher good decision ratio than LRB, but with a worse byte miss ratio.

6.5 Comparison to Belady

We have seen that LRB provides significant improvements over state-of-the-art algorithms. We now compare LRB with the offline Belady MIN (oracle) and relaxed Belady algorithms. Figure 11 compares their byte miss ratios on three cache sizes. We further compare to the state-of-the-art policy, i.e., best performing policy, on each trace and cache size.



(a) Wikipedia

(b) CDN-B1

Figure 12: Traffic reductions of LRB whose sliding memory window parameter are trained by a small portion of trace vs. LRB-OPT whose parameters are trained by the full trace.

We find that LRB indeed reduces the gap between state-of-the-art algorithms and Belady, e.g., by about a quarter on most traces. While a significant gap still remains to Belady, LRB imitates relaxed Belady, which is thus a better reference point for LRB (Section 2). Relaxed Belady represents an ideal LRB with 100% prediction accuracy. The figure shows that LRB is closer to relaxed Belady, e.g., one third to half the distance on most traces. The remaining gap between LRB and relaxed Belady is due to our model's prediction error. This suggests that improvements in the prediction model are a promising direction for future work.

6.6 Sliding Memory Window Size Selection

LRB tunes its memory window on a 20% validation prefix. However, as described in Section 4, the validation prefix is short compared to the optimal sliding memory window choice. To evaluate how much better LRB can do without this restriction, we experimentally determine the best sliding memory window for the full trace. We call the resulting LRB variant "LRB-OPT". LRB-OPT further improves the performance of LRB. Figure 12 shows the traffic reduction over B-LRU for the Wikipedia and CDN-B3 trace. For large cache sizes, LRB-OPT achieves an additional 1–4% traffic reduction compared to LRB. All in all, the ready availability of large amounts of training data in production will be highly beneficial for LRB.

7 Related Work

Although cache design has been studied since 1965 [84], we are only aware of three prior works that have studied how to leverage Belady's MIN [16]. Shepherd Cache [71] used a limited window to the future to emulate Belady's algorithm and defers replacement decisions until reuses occur. Hawkeye and Harmony [47, 48] applied Belady's algorithm to the history of memory accesses to help make better eviction or prefetch decisions. These previous efforts are for hardware caches (in the processor) and do not apply to software cache designs, especially with variable-sized objects.

The rest of the extensive literature can be divided into two major lines of work: caching algorithms and methods to adapt these algorithms to the workload.

Heuristic caching algorithms. We classify these algorithms by the features they use in the eviction decisions (Section 2). The two most widely used features are recency and frequency. Recency is typically measured as variants [3,44,52,68,70] of Delta_1 (as defined in Section 4). Some works also consider static features such as object sizes [4,5,12,24,37,76]. Application ids are commonly considered in shared caches [27,28]. Some algorithms rely entirely on frequency-based features (similar to individual EDCs) [7,11,38,53,63,75].

Another common approach is to combine a recency feature, a frequency feature, and an object's size. This is achieved with either a fixed weighting function between these features [5, 15,18,21,26,35,40,46,56,59,69,73,86,88], or by using a single-parameter adaptation method to dynamically set the weights [13,19,34,49–51,65,81].

Two recent proposals consider a larger range of features [17, 55] and are evaluated using simulations. Unfortunately, they are either outperformed [55] or barely match [17] the miss ratio of recency/frequency-heuristics such as GDSF [26].

LRB uses a superset of all these features and introduces a low-overhead implementation in the ATS production system.

Workload adaptation methods. There are three different methods to adapt caching systems and their parameters to changing workloads.

Only a few papers propose to use machine learning as an adaptation method [8, 17, 20, 31, 32, 55, 87]. There are two branches. The first branch uses reinforcement learning [31,32,55,87], where the state-of-the-art is represented by UCB in our evaluation. The second branch uses supervised learning [17,20], where the state-of-the-art is represented by LFO in our evaluation. All of these proposals are evaluated using simulations. Unfortunately, both UCB and LFO perform much worse than simpler heuristics such as GDSF [26] on CDN traces (Section 6). In fact, recent work concludes that caching "is not amenable to training good policies" [55]. LRB overcomes this challenge using a different feature set (Section 4) and shows the feasibility of implementing machine learning in CDN production systems (Section 5).

The most common prior tuning approaches rely on shadow cache and hill climbing algorithms [13,19,28,34,50,52,56,65, 81,82,88]. Shadow caches are simulations that evaluate the miss ratio of a few parameter choices in parallel. Hill climbing repeatedly reconfigures the system to use the best parameter seen in these simulations, and restarts the simulations in the neighborhood of the new parameter.

Several caching systems rely on mathematical prediction models [9,45,64,83,85]. All heuristics-based algorithms and workload adaptation methods share the same fundamental limitation: they work well with certain workloads and poorly with others.

8 Conclusions

In this paper, we have presented the design, implementation, and evaluation of LRB and have shown that it is possible to design a practical ML-based CDN cache that approximates Belady's MIN algorithm and outperforms state-of-the-art approaches over 6 production CDN traces.

The key advantage of using ML to approximate Belady's MIN algorithm over access-pattern specific or heuristics-based approaches is that it can intelligently make cache eviction decisions based on any access pattern.

More importantly, we have introduced the relaxed Belady algorithm, Belady boundary, and good decision ratio as an eviction quality metric, which has enabled us to take a fundamentally new approach to caching that approximates Belady's MIN algorithm. We expect that these concepts can benefit others in the future.

We have shown that LRB's implementation is practical and deployable by replacing the caching component of a production CDN caching system with LRB. Our experiments show that its throughput and latency are on par with the native implementation. LRB requires neither GPUs nor accelerators and, in fact, its additional CPU and memory overheads are moderate and within the constraints of today's CDN server hardware. This deployable design is enabled by key design decisions including our feature selection, sliding memory window, training data selection, and choice of a lightweight machine learning model.

We have also shown that there is a sizable gap between LRB and the relaxed Belady offline algorithm. A promising direction of future work is to further improve the prediction accuracy of ML in LRB.

While we show tuning LRB's hyperparameters can be done on a small validation trace, we plan to further simplify deployment by automating the tuning of the sliding memory window parameter. These improvements will be available from LRB's repository at `https://github.com/sunnyszy/lrb`.

Acknowledgements

This research is partly funded by a Comcast Innovation grant, a Princeton University fellowship, and a Facebook research grant. We are grateful to our anonymous reviewers, our shepherd Rebecca Isaacs, Amit Levy, and David Liu whose extensive comments substantially improved this work. We also thank Fei Gao, Qizhe Cai, Rong Huang, Jeffrey Helt, and Jennifer Lam who contributed to the project at different stages.

References

[1] Companies using apache traffic server. `https://trafficserver.apache.org/users.html`. Accessed: 2019-04-22.

[2] Netlify. `https://www.netlify.com/open-source/`. Accessed: 2019-04-22.

[3] *A paging experiment with the multics system*. MIT Press, 1969.

[4] Marc Abrams, C. R. Standridge, Ghaleb Abdulla, S. Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. Technical report, Virginia Polytechnic Institute & State University Blacksburgh, VA, 1995.

[5] Marc Abrams, Charles R Standridge, Ghaleb Abdulla, Edward A Fox, and Stephen Williams. Removal policies in network caches for World-Wide Web documents. In *ACM SIGCOMM*, pages 293–305, 1996.

[6] Micah Adler, Ramesh K Sitaraman, and Harish Venkataramani. Algorithms for optimizing the bandwidth cost of content delivery. *Computer Networks*, 55(18):4007–4020, 2011.

[7] Charu Aggarwal, Joel L Wolf, and Philip S Yu. Caching on the world wide web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):94–107, 1999.

[8] Zahaib Akhtar, Yaguang Li, Ramesh Govindan, Emir Halepovic, Shuai Hao, Yan Liu, and Subhabrata Sen. Avic: a cache for adaptive bitrate video. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies*, pages 305–317, 2019.

[9] George Almasi, Calin Cascaval, and David A Padua. Calculating stack distances efficiently. In *MSP/ISMM*, pages 37–43, 2002.

[10] Apache. Traffic Server, 2019. Available at `https://trafficserver.apache.org/`, accessed 09/18/19.

[11] Martin Arlitt, Ludmila Cherkasova, John Dilley, Rich Friedrich, and Tai Jin. Evaluating content management techniques for web proxy caches. *Performance Evaluation Review*, 27(4):3–11, 2000.

[12] Hyokyung Bahn, Kern Koh, Sam H Noh, and SM Lyul. Efficient replacement of nonuniform objects in web caches. *IEEE Computer*, 35(6):65–73, 2002.

[13] Sorav Bansal and Dharmendra S Modha. CAR: Clock with adaptive replacement. In *USENIX FAST*, volume 4, pages 187–200, 2004.

[14] Novella Bartolini, Emiliano Casalicchio, and Salvatore Tucci. A walk through content delivery networks. In *IEEE MASCOTS*, pages 1–25, 2003.

[15] Nathan Beckmann, Haoxian Chen, and Asaf Cidon. LHD: Improving cache hit rate by maximizing hit density. In *USENIX NSDI*, pages 389–403, 2018.

[16] Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems journal*, 5(2):78–101, 1966.

[17] Daniel S Berger. Towards lightweight and robust machine learning for cdn caching. In *ACM HotNets*, pages 134–140, 2018.

[18] Daniel S Berger, Sebastian Henningsen, Florin Ciucu, and Jens B Schmitt. Maximizing cache hit ratios by variance reduction. *ACM SIGMETRICS Performance Evaluation Review*, 43(2):57–59, 2015.

[19] Daniel S. Berger, Ramesh Sitaraman, and Mor Harchol-Balter. Adaptsize: Orchestrating the hot object memory cache in a content delivery network. In *USENIX NSDI*, pages 483–498, 2017.

[20] Adit Bhardwaj and Vaishnav Janardhan. PeCC: Prediction-error Correcting Cache. In *Workshop on ML for Systems at NeurIPS*, December 2018.

[21] Aaron Blankstein, Siddhartha Sen, and Michael J Freedman. Hyperbolic caching: Flexible caching for web applications. In *USENIX ATC*, pages 499–511, 2017.

[22] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[23] Christopher J.C. Burges. From RankNet to LambdaRank to LambdaMART: An Overview. Technical report, Microsoft Research Technical Report MSR-TR-2010-82, 2010.

[24] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *USENIX symposium on Internet technologies and systems*, volume 12, pages 193–206, 1997.

[25] Fangfei Chen, Ramesh K Sitaraman, and Marcelo Torres. End-user mapping: Next generation request routing for content delivery. *ACM SIGCOMM*, 45(4):167–181, 2015.

[26] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In *High-Performance Computing and Networking*, pages 114–123, 2001.

[27] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Dynacache: dynamic cloud caching. In *USENIX HotCloud*, 2015.

[28] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. Cliffhanger: Scaling performance cliffs in web memory caches. In *USENIX NSDI*, pages 379–392, 2016.

[29] CISCO. Cisco visual networking index: Forecast and trends 2022, February 2019. Available at `https://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white-paper-c11-741490.pdf`, accessed 09/18/19.

[30] Eli Cortez, Anand Bonde, Alexandre Muzio, Mark Russinovich, Marcus Fontoura, and Ricardo Bianchini. Resource central: Understanding and predicting workloads for improved resource management in large cloud platforms. In *ACM SOSP*, pages 153–167, 2017.

[31] Renato Costa and Jose Pazos. Mlcache: A multi-armed bandit policy for an operating system page cache. Technical report, University of British Columbia, 2017.

[32] Jeff Dean. Is google using reinforcement learning to improve caching? Personal communication on 2018-09-27, September 2018.

[33] John Dilley, Bruce M. Maggs, Jay Parikh, Harald Prokop, Ramesh K. Sitaraman, and William E. Weihl. Globally distributed content delivery. *IEEE Internet Computing*, 6(5):50–58, 2002.

[34] Gil Einziger, Ohad Eytan, Roy Friedman, and Ben Manes. Adaptive software cache management. In *ACM Middleware*, pages 94–106, 2018.

[35] Gil Einziger and Roy Friedman. Tinylfu: A highly efficient cache admission policy. In *IEEE Euromicro PDP*, pages 146–153, 2014.

[36] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. Flashield: a hybrid key-value cache that controls flash write amplification. In *USENIX NSDI*, pages 65–78, 2019.

[37] Bin Fan, David G Andersen, and Michael Kaminsky. MemC3: Compact and concurrent memcache with dumber caching and smarter hashing. In *USENIX NSDI*, pages 371–384, 2013.

[38] Bin Fan, Hyeontaek Lim, David G Andersen, and Michael Kaminsky. Small cache, big effect: Provable load balancing for randomly partitioned cluster services. In *ACM SoCC*, page 23, 2011.

[39] Jerome H Friedman. Greedy function approximation: a gradient boosting machine. *Annals of statistics*, pages 1189–1232, 2001.

[40] Nicolas Gast and Benny Van Houdt. Transient and steady-state regime of a family of list-based cache replacement algorithms. In *ACM SIGMETRICS*, pages 123–136, 2015.

[41] Syed Hasan, Sergey Gorinsky, Constantine Dovrolis, and Ramesh K Sitaraman. Trade-offs in optimizing the cache deployments of cdns. In *IEEE INFOCOM*, pages 460–468, 2014.

[42] Xinran He, Junfeng Pan, Ou Jin, Tianbing Xu, Bo Liu, Tao Xu, Yanxin Shi, Antoine Atallah, Ralf Herbrich, Stuart Bowers, et al. Practical lessons from predicting clicks on ads at facebook. In *Proceedings of the Eighth International Workshop on Data Mining for Online Advertising*, pages 1–9, 2014.

[43] Leif Hedstrom. Deploying apache traffic server, 2011. Oscon.

[44] 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 *USENIX ATC*, pages 57–69, 2015.

[45] Xiameng Hu, Xiaolin Wang, Lan Zhou, Yingwei Luo, Zhenlin Wang, Chen Ding, and Chencheng Ye. Fast miss ratio curve modeling for storage cache. *ACM Transactions on Storage (TOS)*, 14(2):12, 2018.

[46] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C Li. An analysis of Facebook photo caching. In *ACM SOSP*, pages 167–181, 2013.

[47] Akanksha Jain and Calvin Lin. Back to the future: leveraging belady's algorithm for improved cache replacement. In *ACM/IEEE ISCA*, pages 78–89, 2016.

[48] Akanksha Jain and Calvin Lin. Rethinking belady's algorithm to accommodate prefetching. In *45th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2018, Los Angeles, CA, USA, June 1-6, 2018*, pages 110–123, 2018.

[49] Song Jiang, Feng Chen, and Xiaodong Zhang. CLOCK-Pro: An effective improvement of the clock replacement. In *USENIX ATC*, pages 323–336, 2005.

[50] Song Jiang and Xiaodong Zhang. LIRS: an efficient low inter-reference recency set replacement policy to improve buffer cache performance. *ACM SIGMETRICS*, 30(1):31–42, 2002.

[51] Shudong Jin and Azer Bestavros. GreedyDual* web caching algorithm: exploiting the two sources of temporal locality in web request streams. *Computer Communications*, 24:174–183, 2001.

[52] Theodore Johnson and Dennis Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *VLDB*, pages 439–450, 1994.

[53] George Karakostas and Dimitrios N Serpanos. Exploitation of different types of locality for web caches. In *IEEE ISCC*, pages 207–212, 2002.

[54] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In *Advances in Neural Information Processing Systems*, pages 3146–3154, 2017.

[55] Mathias Lecuyer, Joshua Lockerman, Lamont Nelson, Siddhartha Sen, Amit Sharma, and Aleksandrs Slivkins. Harvesting randomness to optimize distributed systems. In *ACM HotNets*, pages 178–184, 2017.

[56] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. On the existence of a spectrum of policies that subsumes the least recently used (LRU) and least frequently used (LFU) policies. In *ACM SIGMETRICS*, volume 27, pages 134–143, 1999.

[57] Donghee Lee, Jongmoo Choi, Jong-Hun Kim, Sam H Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. LRFU: A spectrum of policies that subsumes the least recently used and least frequently used policies. *IEEE transactions on Computers*, (12):1352–1361, 2001.

[58] Cheng Li, Philip Shilane, Fred Douglis, and Grant Wallace. Pannier: Design and analysis of a container-based flash cache for compound objects. *ACM Transactions on Storage (TOS)*, 13(3):24, 2017.

[59] Cong Li. Dlirs: Improving low inter-reference recency set cache replacement policy with dynamics. In *ACM MSST*, pages 59–64, 2018.

[60] Conglong Li and Alan L Cox. Gd-wheel: a cost-aware replacement policy for key-value stores. In *EUROSYS*, pages 1–15, 2015.

[61] Ping Li. Robust logitboost and adaptive base class (abc) logitboost. *arXiv preprint arXiv:1203.3491*, 2012.

[62] Xiaoliang Ling, Weiwei Deng, Chen Gu, Hucheng Zhou, Cui Li, and Feng Sun. Model ensemble for click prediction in bing search ads. In *Proceedings of the 26th International Conference on World Wide Web Companion*, pages 689–698, 2017.

[63] Bruce M Maggs and Ramesh K Sitaraman. Algorithmic nuggets in content delivery. *ACM SIGCOMM CCR*, 45:52–66, 2015.

[64] Richard L. Mattson, Jan Gecsei, Donald R. Slutz, and Irving L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems journal*, 9(2):78–117, 1970.

[65] Nimrod Megiddo and Dharmendra S Modha. ARC: A self-tuning, low overhead replacement cache. In *USENIX FAST*, volume 3, pages 115–130, 2003.

[66] Kianoosh Mokhtarian and Hans-Arno Jacobsen. Caching in video cdns: Building strong lines of defense. In *ACM EuroSys*, page 13, 2014.

[67] Devon H. O'Dell. Personal communication at Fastly.

[68] Elizabeth J O'Neil, Patrick E O'Neil, and Gerhard Weikum. The LRU-K page replacement algorithm for database disk buffering. *ACM SIGMOD*, 22(2):297–306, 1993.

[69] Sejin Park and Chanik Park. Frd: A filtering based buffer cache algorithm that considers both frequency and reuse distance. In *ACM MSST*, pages 59–64, 2017.

[70] Seon-yeong Park, Dawoon Jung, Jeong-uk Kang, Jinsoo Kim, and Joonwon Lee. CFLRU: a replacement algorithm for flash memory. In *ACM/IEEE CASES*, pages 234–241, 2006.

[71] Kaushik Rajan and Govindarajan Ramaswamy. Emulating optimal replacement with a shepherd cache. In *IEEE/ACM MICRO*, pages 445–454, 2007.

[72] Matthew Richardson, Ewa Dominowska, and Robert Ragno. Predicting clicks: estimating the click-through rate for new ads. In *Proceedings of the 16th international conference on World Wide Web*, pages 521–530, 2007.

[73] Luigi Rizzo and Lorenzo Vicisano. Replacement policies for a proxy cache. *IEEE/ACM TON*, 8:158–170, 2000.

[74] Emanuele Rocca. Running Wikipedia.org, June 2016. Available at `https://www.mediawiki.org/wiki/File:WMF_Traffic_Varnishcon_2016.pdf`, accessed 09/18/19.

[75] Ketan Shah, Anirban Mitra, and Dhruv Matani. An O(1) algorithm for implementing the LFU cache eviction scheme. Technical report, Stony Brook University, 2010.

[76] David Starobinski and David Tse. Probabilistic methods for web caching. *Performance evaluation*, 46:125–137, 2001.

[77] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K Sitaraman. Footprint descriptors: Theory and practice of cache provisioning in a global cdn. In *ACM CoNEXT*, pages 55–67, 2017.

[78] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. RIPQ: advanced photo caching on flash for facebook. In *USENIX FAST*, pages 373–386, 2015.

[79] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. Popularity prediction of facebook videos for higher quality streaming. In *USENIX ATC*, pages 111–123, 2017.

[80] Ilya Trofimov, Anna Kornetova, and Valery Topinskiy. Using boosted trees for click-through rate prediction for sponsored search. In *Proceedings of the Sixth International Workshop on Data Mining for Online Advertising and Internet Economy*, pages 1–6, 2012.

[81] Giuseppe Vietri, Liana V Rodriguez, Wendy A Martinez, Steven Lyons, Jason Liu, Raju Rangaswami, Ming Zhao, and Giri Narasimhan. Driving cache replacement with ML-based LeCaR. In *USENIX HotStorage*, 2018.

[82] Carl Waldspurger, Trausti Saemundsson, Irfan Ahmad, and Nohhyun Park. Cache modeling and optimization using miniature simulations. In *USENIX ATC*, pages 487–498, 2017.

[83] Carl A Waldspurger, Nohhyun Park, Alexander Garthwaite, and Irfan Ahmad. Efficient MRC construction with SHARDS. In *USENIX FAST*, pages 95–110, 2015.

[84] Maurice V Wilkes. Slave memories and dynamic storage allocation. *IEEE Transactions Electronic Computers*, 14(2):270–271, 1965.

[85] Jake Wires, Stephen Ingram, Zachary Drudi, Nicholas JA Harvey, and Andrew Warfield. Characterizing storage workloads with counter stacks. In *USENIX OSDI*, pages 335–349, 2014.

[86] Roland P Wooster and Marc Abrams. Proxy caching that estimates page load delays. *Computer Networks and ISDN Systems*, 29(8):977–986, 1997.

[87] Chen Zhong, M Cenk Gursoy, and Senem Velipasalar. A deep reinforcement learning-based framework for content caching. In *IEEE CISS*, pages 1–6, 2018.

[88] Yuanyuan Zhou, James Philbin, and Kai Li. The multi-queue replacement algorithm for second level buffer caches. In *USENIX ATC*, pages 91–104, 2001.