HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network

Zhenyu Song, *Princeton University;* Kevin Chen, Nikhil Sarda, Deniz Altınbüken, Eugene Brevdo, Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, and Ramki Gummadi, *Google*

https://www.usenix.org/conference/nsdi23/presentation/song-zhenyu

This paper is included in the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation.

April 17–19, 2023 • Boston, MA, USA

978-1-939133-33-5

Open access to the Proceedings of the 20th USENIX Symposium on Networked Systems Design and Implementation is sponsored by

HALP: Heuristic Aided Learned Preference Eviction Policy for YouTube Content Delivery Network

Zhenyu Song*†, Kevin Chen*, Nikhil Sarda*, Deniz Altınbüken, Eugene Brevdo,
Jimmy Coleman, Xiao Ju, Pawel Jurczyk, Richard Schooler, Ramki Gummadi*

Google

Abstract

Video streaming services are among the largest web applications in production, and a large source of downstream internet traffic. A large-scale video streaming service at Google, YouTube, leverages a Content Delivery Network (CDN) to serve its users. A key consideration in providing a seamless service is cache efficiency. In this work, we demonstrate machine learning techniques to improve the efficiency of YouTube's CDN DRAM cache. While many recently proposed learning-based caching algorithms show promising results, we identify and address three challenges blocking deployment of such techniques in a large-scale production environment: computation overhead for learning, robust byte miss ratio improvement, and measuring impact under production noise. We propose a novel caching algorithm, HALP, which achieves low CPU overhead and robust byte miss ratio improvement by augmenting a heuristic policy with machine learning. We also propose a production measurement method, impact distribution analysis, that can accurately measure the impact distribution of a new caching algorithm deployment in a noisy production environment.

HALP has been running in YouTube CDN production as a DRAM level eviction algorithm since early 2022 and has reliably reduced the byte miss during peak by an average of 9.1% while expending a modest CPU overhead of 1.8%.

1 Introduction

YouTube is one of the largest sources of downstream internet traffic, accounting for 15% of global application traffic in 2021 [27]. It leverages a Content Delivery Network with a presence in more than 200 countries and territories to serve videos to over 2 billion users [30]. Caching in CDNs is done by storing content, such as videos, in proxy servers that are distributed closer to end users instead of delivering content from the origin servers. A CDN uses multiple levels of caches to reduce the cost of content distribution and access latency for end users. A key metric to optimize in CDN caches is byte miss ratio, i.e., the portion of user-requested bytes missed in the CDN cache.

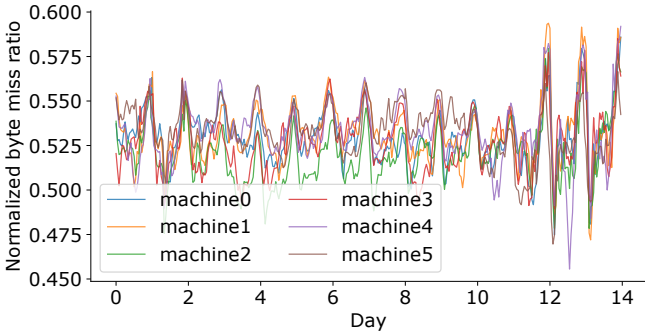


Figure 1: Byte miss ratios over time for six machines on a rack, normalized by dividing with a reference constant to hide the proprietary absolute values. The substantial differences across machines and over time make it hard to measure a new cache algorithm's production impact accurately.

Recently, machine learning techniques have been used to improve cache eviction and admission policies (e.g., [32, 34]). Caching algorithms can benefit from learning patterns from existing workloads, predicting which byte is more likely to be accessed in the future, and using this information to improve caching decisions.

In this paper, we present a new cache eviction algorithm called Heuristic-Augmented Learned Preferences (HALP), and share our experience in deploying HALP at a large scale. From our experience, while reducing the byte miss ratio is important to improve cache efficiency, it is not the sole criterion for deployment. For a solution that uses machine learning to be deployed in a large-scale production environment, there are three main challenges that need to be tackled:

- **Computation overhead for learning.** Learning-based cache algorithms can be more computationally expensive to run compared to heuristic-based algorithms. The model training and prediction cost is high compared to

*Equal technical contributions. The corresponding author is Pawel Jurczyk (pawelj@google.com).

†Zhenyu is affiliated with Princeton University, but this work was done during his internship in Google.

normal cache operation such as LRU eviction. Using LRB [32] as an example, for each eviction it needs to run predictions for 64 objects, which makes deployment cost-prohibitive ($\approx$ 19.2% additional CPU overhead)

- **Robust byte miss ratio improvement.** Learning-based cache algorithms can introduce regressions if their design does not include a regression prevention mechanism. For large-scale systems, bounding regression of byte miss ratio is crucial. YouTube CDN contains a large number of locations, and byte miss ratio regressions in even a few locations could result in degraded user experience. In addition, having a robust algorithm also increases our confidence in the design.
- **Measuring impact under production noise.** It is challenging to accurately measure the impact of a new eviction algorithm in a large-scale deployment. We cannot solely rely on simulations as they are imperfect proxies for production behavior. It is also impractical to replicate user requests and test different algorithms at every location. Therefore, current production practice uses A/B testing. An example is to compare different machines on a rack because machines on a rack share the same hardware/software configurations, and the request mix they receive should be similar. However, in practice, the behaviors of machines are never identical. Figure 1 shows byte miss ratios over time for six machines on a rack. The substantial differences across machines and over time make it hard to measure the production impact of a new algorithm accurately.

To tackle the first two challenges, we develop a novel approach, the HALP policy, to perform eviction decisions with low-overhead and to generalize over the whole production system with limited regressions. It achieves this by augmenting a heuristic policy with machine learning instead of learning a policy end-to-end. The HALP eviction policy uses the heuristic policy to select eviction candidates and the learning policy to pick the final object to evict from those candidates.

To address the third challenge, we developed an impact distribution analysis that evaluates the impact of a new caching algorithm deployment in a noisy production environment.

HALP has been deployed in YouTube's CDN as a DRAM level eviction algorithm since early 2022. It has robustly reduced the byte miss by an average of 9.1%. In addition, these improvements were achieved with a modest 1.8% CPU overhead.

In this paper we make the following three contributions:

- We present Heuristic Augmented Learned Preferences (HALP), a learned cache eviction algorithm with low computation overhead and robust byte miss ratio improvement by augmenting a heuristic policy with a learnable scoring function.
- We propose an impact distribution analysis to measure the impact of a caching algorithm in the presence of production noise.
- We evaluate HALP in YouTube's large-scale production environment and provide a detailed analysis on how it improves the cache efficiency of YouTube CDNs. [1]

The paper is structured as follows: §2 describes the background of the problem. §3 covers the design of HALP. §4 introduces our impact distribution analysis design, and §5 shows the evaluation results.

2 Background

In this section, we give an overview of the YouTube CDN architecture. We then describe heuristic-based caching algorithms and learning-based caching algorithms. Lastly, we describe the key ideas for deploying a learned cache algorithm in a large-scale production environment.

2.1 YouTube CDN Edge Cluster Architecture

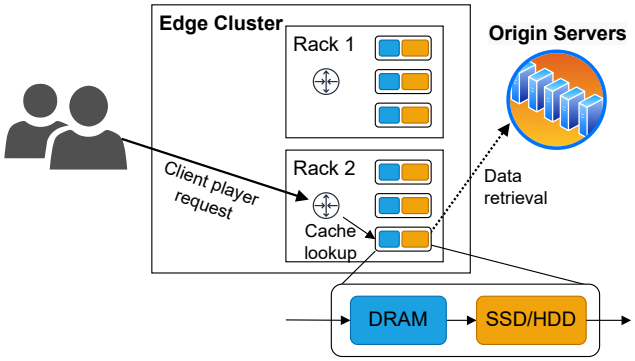


Figure 2: A YouTube CDN edge cluster contains multiple racks of servers. Machines in a rack are of the same type.

YouTube CDN [4, 11] contains edge clusters spreading more than 200 countries and territories globally. As shown in Figure 2, an edge cluster consists of multiple racks. Each rack contains multiple cache servers configured homogeneously using the same type of machines. Servers from different racks may have hardware from different generations. Each cache server is equipped with DRAM, SSDs and HDDs used for caching data chunks. A video is stored in these data chunks on the cache server.

Client player requests are sharded amongst machines in an edge cluster. A request includes a key and a byte range of a data chunk. Because a video is played sequentially, video range requests are issued sequentially as well. On the arrival of a request, the server checks if the requested data chunk is

[1] Two traces from a developed market region and an emerging market region (§5.2) can be shared with interested parties, but a signed data sharing agreement between Google and the outside institutions is required.

in its DRAM. If it is not present (a.k.a. is missed), the data will be fetched from other cache layers such as local SSDs and HDDs, with the remote origin server being the last resort. When the DRAM cache is full and a miss occurs, an eviction algorithm is used to remove data chunks from the cache to insert new data chunks.

As the first caching tier, the DRAM cache serves an important role in reducing traffic for subsequent tiers. It also contributes to the overall storage costs of the YouTube CDN. A better DRAM eviction algorithm with a lower byte miss ratio would require less DRAM to be provisioned while keeping a similar traffic reduction on subsequent tiers. This saves the overall resource cost as long as the computation overhead is modest (which requires additional CPU resources). We focus on the byte miss ratio during the peak hours. This is because during peak hours, large numbers of videos are concurrently accessed, causing the byte miss ratio peaks, which could degrade Quality of Experience (QoE). We therefore focus on reducing the 95th percentile byte miss ratio: **P95 byte miss ratio**. We choose to not directly optimize QoE because it is too noisy as feedback for each cache eviction. §5.3 and §5.4 list other metrics related to cache performance.

The previous eviction algorithm used in production [28] uses heuristic to rank chunks. A score is computed for each chunk by summing its request rate score and end of chunk score, and the chunk with lowest score would be evicted. The request rate score is calculated based on the chunk's past request rate, which captures the temporal locality. The end of chunk score is a binary score indicating whether the previous range request hits the chunk end. Since range requests for a chunk are issued sequentially, after a client requests the last byte of a video chunk, the same chunk is less likely to be fetched again. This score captures the spatial locality.

2.2 Heuristic and Learned Cache Algorithms

Many heuristic cache algorithms maintain a priority queue for objects in the cache and select the lowest priority object to evict when a miss occurs. For example, A Least Recently Used (LRU) policy uses the latest time of access for an object to determine evictions. This ordering is good for workloads where objects that have been accessed recently are more likely to be accessed repeatedly. A First In First Out (FIFO) policy uses the order in which items were inserted into the cache for determining evictions. This ordering performs well for workloads where objects are accessed sequentially. Managing priority queues is efficient, which makes these algorithms efficient as well. However, these algorithms work well in some workloads, but not in others. Caching policies that can self-tune and balance between different features, recency and frequency in Adaptive Replacement Cache (ARC) [25] or object size and frequency in Greedy-Dual-Size-Frequency (GDSF) [12, 13], can cover a wider range of workloads but only adapt to specific features [21], limiting their performance for changing workloads.

Learning-based algorithms like LRB [32] achieve better performance than heuristic algorithms, because they train a model to learn the cache access pattern directly from the trace instead of assuming a static workload behavior. As an example, LRB maintains features for objects that are both currently, and historically present in the cache, and trains a regression model to predict an object's time to next access. When an eviction is required, it randomly samples 64 objects and runs this predictive model on them and evicts the object which is predicted to be accessed furthest in the future.

When optimizing the byte miss ratio for variable-size objects, the eviction methodology is similar to optimizing the miss ratio for uni-size objects. This is because in the variable-size object scenario, we can treat each eviction decision as a group of decisions, which evicts each byte of an object individually.

3 HALP Eviction Policy Design

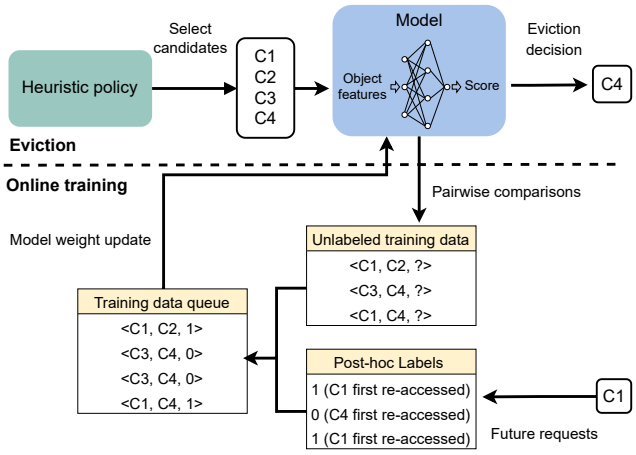


Figure 3: The architecture of HALP. A key component of HALP is a neural network based *score function*, whose inputs correspond to the features for a single eviction candidate, and whose output is a real valued score which tracks the likelihood of a quick re-access to the same object. When an eviction is required, a heuristic policy (e.g., LRU) is used to propose a small set of eviction candidates. Then a neural network-based model ranks the eviction candidates and selects the final eviction decision by pairwise comparisons. The same pairwise comparisons are also used to generate training data for online training.

This section describes the design of HALP, which is illustrated in Figure 3. A key component of HALP is a neural network based *score function*, whose inputs correspond to the features for a single eviction candidate, and whose output is a real valued score which tracks the likelihood of a quick re-access to the same object. When an eviction is required, a

heuristic policy is first used to propose a small set of eviction candidates. Then, a neural network score function is used to re-rank this small set of candidates, to identify the final eviction decision. A key design challenge involves how to learn the scoring function, which involves both generating the training data and adjusting the weights of the neural network. As part of its design, HALP also includes the steps required to efficiently update this score function, starting with randomly initialized weights.

Because of these design choices, HALP can be deployed without the operational overhead of having to separately manage the labeled examples, training procedure and the model versions in separate offline pipelines. Therefore, HALP has minimal extra overhead for operation similar to other heuristic policies, but has the added benefit of being able to take advantage of additional features to make its eviction decisions and continuously adapt to a changing access patterns.

To learn the score function efficiently, we convert the ranking problem into a small set of pairwise preference queries, which is a general and robust framework for learning to rank [9, 29] multiple items from a list. As a result, HALP makes repeated use of pairwise comparisons during decision making to simultaneously generate training data for online training. One challenge in efficiently managing the training data is that the time required to identify the labels is non-deterministic and depends on the future re-accesses to items. HALP snapshots the features generated for pairwise comparisons used at eviction decision time saved as unlabeled training data tuples (see Figure 3). In parallel, HALP continuously observes incoming requests to resolve any pending labels for prior comparisons and generate training data that continuously updates the model.

3.1 Heuristic-based Candidate Selection

A key insight for ensuring a low overhead is that many objects can be easily excluded from eviction consideration without the need to use expensive computations, ML or otherwise. For example, objects near the head of an LRU priority queue are less likely to be good eviction candidates as opposed to objects near the tail. Therefore, we can appropriately bias our learned eviction towards only the tail instead of considering the entire cache, saving overhead on training and inference.

The goal of using a heuristic policy for candidate selection is to reduce the ML computational overhead. It also provides a lower limit on decision quality. This heuristic algorithm can be selected as LRU, LFU, or other heuristic policies. We find in practice LRU policy is sufficient to achieve good performance.

The number of eviction candidates is a hyperparameter. If too many candidates are selected, the ML pipeline overhead will be too high. But too few candidates may lead to not a single good candidate to evict. We find empirically selecting four candidates achieves a good balance between the recall of good candidates and the incurred CPU overhead.

3.2 Ranking-based Learned Eviction

HALP is designed to provide better eviction decisions than the heuristic algorithm in the general case. To achieve this, the pairwise comparisons should pick the eviction decision that is the best for improving cache efficiency. Since the goal of cache eviction is to use the limited size of the cache to receive as many hits as possible, finding the best eviction decision is equivalent to ranking the candidate that will be accessed furthest in the future (or not at all) highest, in effect evicting it before other candidates.

After four eviction candidates are selected from the heuristic, the best candidate is selected based on three pairwise comparisons done in tournament style. The deselected candidates are re-inserted into the heuristic policy. In the case for LRU, those deselected candidates are re-inserted into the head of LRU queue.

To have a theoretical intuition that the combination of a heuristic and a learning policy can increase the robustness of eviction candidate selection, we analyze a simple Gaussian model for the benefits of re-ranking in Appendix B. This analysis underlines the conditions under which such re-ranking might generate more utility than the baseline heuristic.

Online Training. As shown in Figure 3, when a pairwise comparison is done, the same pair of candidates is selected to generate training data. However, at the time of prediction, the required label (i.e., which of these two candidates will be accessed further in the future) is not available. Therefore, an initial feature snapshot is taken at the pairwise comparison during eviction and is buffered in an unlabeled state with a label placeholder until one of the candidates is accessed again, making the label available. Accordingly, HALP maintains a collection of pending comparisons. This collection of pending comparisons continuously observes all incoming requests, and upon the first access to either candidate, a binary label is assigned to construct the training example.

HALP keeps the feature metadata of objects in a "*ghost cache*". For our application of video caching, this metadata is lightweight relative to the sizes of the objects being cached, therefore the information continues to be stored for keys that are evicted from the cache up to a limit. This limit is set to be a multiple of the number of elements in the actual cache to track enough history. Evictions from the ghost cache are performed using LRU when the size exceeds this set limit. When a key is removed from the ghost cache, pending comparisons associated with the key are also deleted.

The training data generated from the above procedure is stored in an in-memory replay buffer. When the replay buffer accumulates 1024 training entries, it creates a mini-batch to update the ML model. The retrain batch size is a hyperparameter of HALP and was chosen empirically. Theoretically, a

pathological workload could have access pattern shifts aligning with the retraining period, causing HALP performance degradation. However, we didn't observe this in our production workloads.

The online training framework and the model itself are written using XLA [2] and carefully crafted C++ code to ensure low overhead. We also leverage uncommon synchronization primitives such as user space per CPU spinlocks and RCUs to ensure maximum performance without sacrificing scalability and thread safety in highly concurrent environments.

ML Model. The model is trained as a binary classification task (which of a pair gets re-accessed first) with cross entropy loss. It is a simple neural network model with one hidden layer. We found that increasing the number of layers did not help improve the model further. With this simple model, we are able to run a pairwise prediction in 720 ns, and each training in several ms. And HALP implementation is based on Google's SmartChoices service [10]. Details about the loss function and the model weight updates are provided in Appendix A.

Feature name	Dimension
Access-based	
Time between accesses	32
Exponential decay counters	10
Number of accesses	1
Average time between accesses	1
Time since last access	1
Video-specific	
End of chunk	1

Table 1: Features used by HALP.

Features. Table 1 shows the features HALP uses. Of these features, time since last access, time between accesses, and exponential decay counters are the same as the features used in [32]. Time since last access and time between accesses capture short-term access patterns while they retain information about at most 32 accesses. Exponential decay counters (EDCs) capture longer term trends. The end of chunk score is identical to the previous production algorithm (§2.1).

4 Impact Distribution Analysis

Comparing cache algorithms may seem like a straightforward hypothesis test (e.g., t-test or z-test) over an A/B testing experiment. A new algorithm with lower byte miss ratio that passes the hypothesis test would generally be considered as an improvement. However, the operating conditions in a large-scale system could be very diverse, and understanding the robustness of an improvement is critical to decision making in practice.

To illustrate the risk of solely relying on mean-shift estimates, consider a scenario where a new algorithm is beneficial for most machines but performs extremely poorly for some small set of machines. In that case, applying the new algorithm everywhere could be sub-optimal. Any algorithms without a theoretical performance lower-bound (relative to an optimal solution) have these risks, but the concern is exacerbated for learning algorithms that are prone to over-fitting.

A naive approach to the diversity problem is to enumerate all configurations and perform separate A/B tests (e.g., one test for each rack where workload and hardware is assumed to be similar). However, this severely limits the number of data points, and the signal to noise ratio for each configuration could be very poor in a production setting. Figure 1 is a typical example of byte miss ratio variation for production machines on the same rack with identical setting.

We propose a novel impact distribution analysis to get a more holistic picture of how a new algorithm is affecting the fleet. Specifically, instead of estimating the average performance change, we try to estimate the *distribution* of performance changes across different conditions.

4.1 Model of Measurements

We model the measured relative improvement as

$$M = I + N \tag{1}$$

where I represents actual impact and N represents the noise. In other words, we model the PDF of M as a convolution between I and N.

The core idea is that we could directly sample M by A/B tests, and sample N by A/A tests (performance difference measured in no-op experiment), and once we have those two distributions, we can get to I by deconvolution.

4.2 Measurement Setup

The environment we want to measure the effect of using HALP comprises of racks from hundreds of locations. In our experiment setup, we randomly split machines in a rack into three different configurations:

- **Experiment Machines:** Experiment machines use the HALP algorithm.
- **No-op Machines:** No-op machines use the baseline caching algorithm. They are used to measure the production environment noise.
- **Control Machines:** Control machines also use the baseline caching algorithm. They are selected as the baseline to compare with the experiment group and no-op group.

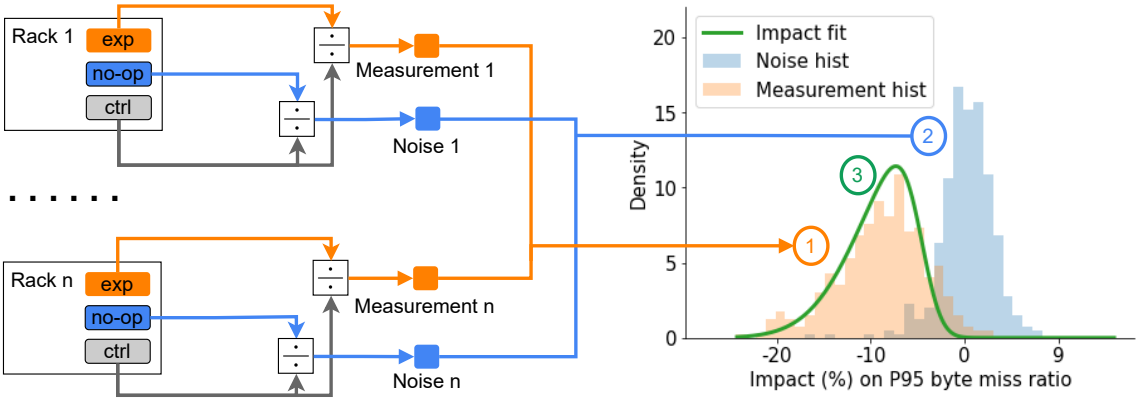


Figure 4: Impact distribution analysis procedure: 1. Estimate measurement distribution. 2. Estimate noise distribution. 3. Fit impact distribution.

Figure 4 shows how the measurements from these three groups are used to calculate the impact distribution. We first collect the relative values (e.g., relative byte miss ratios) of experiment machines over control machines as measurement samples (Measurement M). Then we collect the relative values of no-op machines over control machines as an estimation of the environment noise (Noise N). Finally, we fit an impact distribution from the measurement distribution and noise distribution using each comparison as a data point, as we describe in the next subsection.

4.3 Fitting Impact Distribution

Algorithm 1 Algorithm for fitting impact distribution

Input: measurement samples M, noise samples N, and distribution candidates.

Output: measurement distribution P_M, impact distribution P_I, noise distribution P_N.

1: $P_N = FitByMLE(dist=\text{“t-dist”}, N)$
2: **for** candidate_dist in candidate_distributions **do**
3: // Approximate P_M by discretized grid G.
4: $P_I = FitByMLE(dist=\text{candidate\_dist}, M, P_N)$, with $P_M(m) \approx \sum_{v \in G} P_N(v) P_I(m-v)$
5: **end for**
6: **return** P_I with the highest likelihood

Algorithm 1 describes our algorithm to fit the impact distribution given sample M and N. We first fit the noise distribution P_N using noise samples with maximum likelihood estimation (MLE) (Line 1). We assume the noise has zero mean and follows a t-distribution. We choose a t-distribution because we expect noise to exhibit a symmetric and bell-shaped behavior like the normal distribution but allow fitting to have more degrees of freedom.

Next, we fit the impact distribution (Lines 2-5). This is done in two steps. First, a distribution type for impact needs to be chosen (Line 2). Since this depends on the setting, several well-known distributions could be reasonable candidates. Therefore, we iterate over a list of common distributions (beta, non-central student, and skewed normal) and pick the one that best fits our data. Second, we estimate the measurement distribution by discretizing the distribution into a fine-grained grid G. Then we use the maximum likelihood estimation to fit the chosen distribution candidate to find the measurement distribution that is the best fit (Line 4).

Note that this method is only feasible because we have machine data from more than 200 countries and territories. Without enough samples the fitting will not be able to recover impact accurately from the noise.

5 Evaluation

In this section, our goal is to answer the following questions for HALP and our impact distribution analysis:

- Can HALP improve cache performance without causing regression in production? (§5.3)?
- What is the computation overhead of HALP compared to the previous production algorithm (§5.4)?
- How does HALP compare to other learned and heuristic algorithms (§5.5)?
- What are the effects when changing HALP's hyperparameters (§5.6)?

5.1 Deployment Setup

Deployment HALP was rolled out in production in early 2022. The rollout was done in stages, and the impact of the new algorithm was monitored using the impact distribution

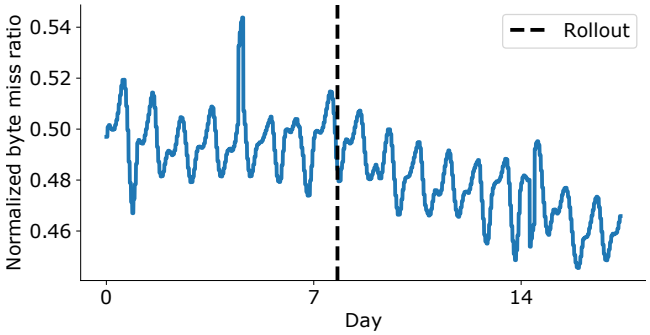


Figure 5: YouTube fleet normalized byte miss ratio for the DRAM level, before and after rollout.

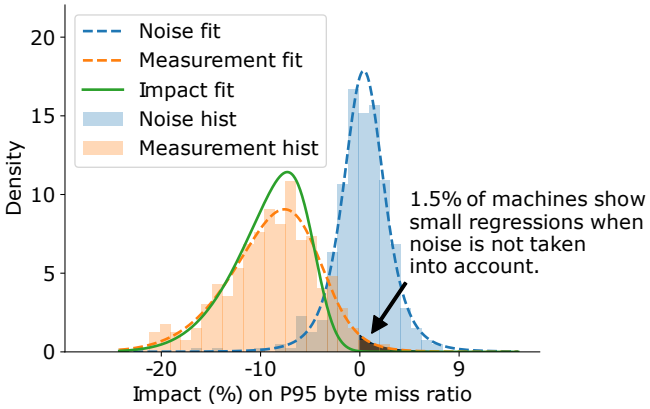


Figure 6: Our impact distribution analysis can "denoise" the experiment measurements from the no-op measurements and reveals HALP's clear average 9.1% byte miss ratio reduction with negligible regression. When noise is not taken into account, the measurements show 1.5% of racks that were impacted negatively, where some machines showed up to 4% byte miss ratio increase.

analysis on the changes in DRAM byte miss ratio. Figure 5 shows the DRAM byte miss ratio changes in the fleet before and after the rollout, which shows a fleet-wide drop.

Release Process HALP has an online design to make sure that the model does not degrade over time. As a result, the online model does not require new releases. While the model does not need to go through production releases, code changes and any improvements in HALP design go into production through releases that happen periodically. HALP is integrated into the existing YouTube release process, which guarantees that during rollout, HALP goes through the release tests and any code changes are released in a safe manner.

Monitoring As part of maintaining stable performance, HALP is integrated into the monitoring setup used for monitoring YouTube deployment. In addition to existing metrics that monitor cache efficiency, two new metrics were added to monitor HALP performance: 1) model accuracy, and 2) the byte miss ratio difference between HALP and a holdout previous production algorithm. For model accuracy, we monitor model loss which is an indicator of how good the model decisions are. For the byte miss ratio difference, we keep 1% of the machines running with previous production algorithm and alert if the byte miss ratio for machines using HALP become worse than the heuristic algorithm. We do not use the impact distribution analysis here as our goal is to detect abnormal behaviors with a low false positive ratio.

5.2 Experimental Methodology

Production experiments. To measure reduction in the byte miss ratio and overhead, we used production experiments and our impact distribution analysis. To use the impact distribution analysis, we randomly selected a small percentage of racks from all locations. For each rack, we selected one experiment machine, one no-op machine, and the rest as control machines (§4). We use one day of data for our measurement after observing HALP training is stable.

Simulation experiments. We use simulation to measure how HALP compares to other cache algorithms and how the changes in hyperparameters effect the performance of HALP. Because simulation does not have production noise and is deterministic, and direct comparisons can be set up reliably, we compare the byte miss ratio from simulations directly instead of using our impact distribution analysis.

For simulation experiments (except two experiments in §5.5), we use traces from a small percentage of randomly selected locations. For each location selected, we choose four traces, each three days long, with each trace coming from different quarters in the calendar year of 2021 (except the retrain interval experiment uses six days long trace). Using a diverse set of traces helps reduce seasonal/weekly noise. For each simulation, the first day of the trace is used as a warm-up, and we measure the P95 byte miss ratio of the next two days.

5.3 HALP Improvements in Production

P95 byte miss ratio. This experiment measures the impact of HALP on the byte miss ratio, disk latency, and joint latency during production. To measure the improvement, we collected byte miss ratios from machines on randomly selected racks and applied our impact distribution analysis. Figure 6 shows the relative change in the byte miss ratio distribution. The regions and lines are the P95 byte miss ratio distribution relative to the control groups.

The blue region is the no-op group relative change distribution. Although the no-op group uses the same configuration as the control group, the noise can cause up to 10% difference in P95 byte miss ratio. The blue dash line shows the fitted t-distribution of the noise. The orange region is the mea-

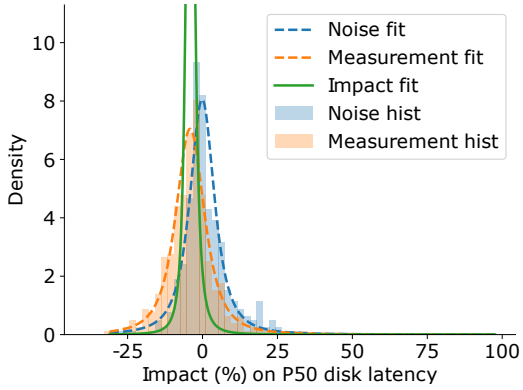


Figure 7: Compared with the previous production algorithm, HALP reduces disk first byte latency by an average of 3.8%.

surement distribution. The orange dash line shows the measurement fitted gamma distribution, which was picked as the best distribution for this specific measurement. When noise is not taken into account, as shown by the orange dash line there is 1.5% of racks that were impacted negatively, where some machines showed up to 4% byte miss ratio increase. However, just looking at the measurement fit, we do not know whether the negative impact is because of the algorithm or the production noise.

Our impact distribution analysis (Section 4) can "denoise" the experiment measurements from the no-op measurements, and the result is the green curve showing a clean byte miss ratio reduction up to 24% with negligible regression, with an average reduction of 9.1%. This shows HALP can not only improve the average byte miss ratio, but also has negligible regression. In addition, the variance of improvements also shows different locations have different access patterns which have different difficulty for learning.

Disk first byte latency. Disk first byte latency is the time between a disk cache receives a request and returns the first byte. It is a good indicator of DRAM cache efficiency because a better DRAM eviction algorithm reduces the number of requests to the disk layer, thus reducing the disk request queue length. Figure 7 shows the disk first byte latency impact of HALP. Compared with the previous production algorithm, the change in latency ranges from a 13% reduction to a 5% increase, with an average reduction of 3.8%.

The tail increase (the part of the impact fit that is above 0) is likely to come from the object miss ratio increase, which is more correlated with disk first byte latency than the byte miss ratio. Different from the byte miss ratio, the object miss ratio is the fraction of user requests missed in the cache. These two metrics may conflict with each other. Since HALP's primary goal is to reduce the byte miss ratio, it may slightly increase the object miss ratio in certain cases.

Join latency. Join latency is the time taken to start video playback after the user hits "play", and one of the most important metrics of streaming. Since join latency is a playback metric that involves both clients and servers instead of servers only, we are unable to use our impact distribution analysis to measure it. We set up an experiment that distributes playbacks from clients to server machines with and without HALP and compares latency on clients. HALP reduces join latency from 1.03% to 1.41%, with an average reduction of 1.22%. This shows that the improvement of the memory cache has a strong impact on the end-to-end user experience.

5.4 HALP Computation Overhead

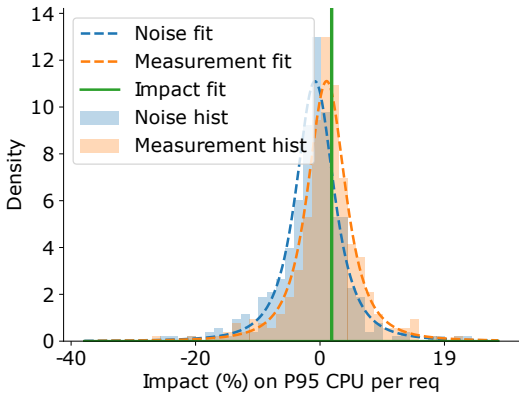


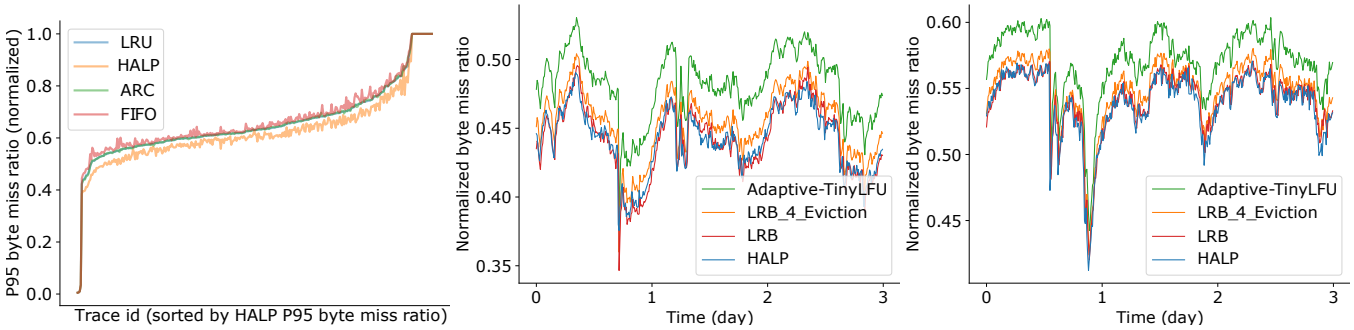
Figure 8: The CPU overhead of HALP is 1.8% per request with low variance. This implies the additional CPU cost is roughly linear to the number of requests.

Learning comes with overheads, so it is important to measure these overheads and underline the trade-offs. In this section, we show the computation overhead associated with using HALP for cache eviction decisions, including prediction and online training.

We have analyzed the extra CPU overhead that is incurred while using HALP using our impact distribution analysis, and Figure 8 shows the impact of P95 CPU normalized per request. The CPU impact is consistently at 1.8% with low variance, meaning the additional CPU cost is roughly linear to the number of requests. This is because both training and prediction costs are roughly linear to the number of requests. For training, the cost is roughly linear to the amount of training data, and on average each prediction generates a single pair of training data. In addition, each eviction requires three pairwise comparisons. Finally, since all locations have similar byte miss ratios, the number of evictions (misses) is roughly linear in the number of requests. To conclude, the computation overhead is small compared to the byte miss ratio improvement.

5.5 HALP vs. Other Cache Algorithms

To further evaluate HALP, we ran simulation experiments to compare it with other cache eviction algorithms.



(a) (b) Developed market (c) Emerging market

Figure 9: (Fig. 9a) P95 byte miss ratios for different cache algorithms over a variety of traces in simulation. HALP achieves a strictly better performance than all other algorithms on 92.6% of traces, and achieves the same performance as the best of the other algorithms on 7% of traces, in effect performing worse than the best algorithm on only 0.4% of traces. (Figs. 9b and 9c) Normalized byte miss ratio over time for HALP and LRB [32] on production traces from a developed market region, and an emerging market region. HALP achieves a similar byte miss ratio, but only needs 4 eviction candidates compared to 64 for LRB. Tuning LRB's eviction candidates from 64 to 4 would increase P95 byte miss by about 2%.

Comparison with classic cache algorithms

We compared HALP with three heuristic cache algorithms: LRU, FIFO, and ARC [25]. Figure 9a shows the normalized P95 byte miss ratios for different traces, sorted by HALP P95 byte miss ratio. HALP achieves a strictly better performance than all other algorithms on 92.6% of traces, and achieves the same performance as the best of the other algorithms on 7% of traces, in effect performing worse than the best algorithm on only 0.4% of traces.

Comparison with advanced cache algorithms

We compared HALP with a state-of-the-art learned cache algorithm LRB [32] and heuristic cache algorithm Adaptive-TinyLFU [15]. We use two YouTube production traces from a developed market region and an emerging market region in 2020 to get robust results. These traces are four days long. Our implementation of LRB and Adaptive-TinyLFU is based on LRB's open-source simulator [1]. For a fair comparison, we extended LRB's features to be identical as HALP. We tuned LRB's major hyperparameter (memory window) using its public implementation. In additional to using LRB's original 64 eviction candidates, we also tested using 4 eviction candidates identical as HALP. Figure 9b and 9c show normalized byte miss ratios over time for Adaptive-TinyLFU, LRB with 4/64 eviction candidates, and LRB. We use the first day of each trace as a warm-up, and exclude it from the figures.

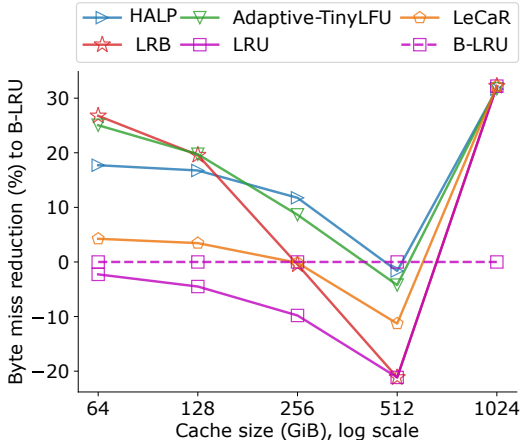


Figure 10: The byte miss reduction of different algorithms compared to B-LRU on a public trace from a Wikipedia CDN node. LRB, Adaptive-TinyLFU, HALP, and B-LRU achieve the best performance individually at cache size 64 GiB, 128 GiB, 256 GiB, 512 GiB. At 1024 GiB, the cache is big enough that all cache algorithms converge.

Adaptive-TinyLFU achieves P95 byte miss ratios of 0.515 and 0.598 on two traces. Compared to it, LRB and HALP achieve smaller ratios (0.479/0.565 for LRB, 0.475/0.564 for HALP). HALP achieves similar P95 byte miss ratios compared to LRB (0.17%/0.83% less byte misses) with less than an order of magnitude computation overhead. For each eviction it only compares 4 eviction candidates instead of 64 for LRB. As a result, HALP computes a prediction for each eviction in 2.1 μs in comparison to 60 μs that is required by LRB [32]. Tuning LRB's eviction candidates from 64 to 4 would increase P95 byte misses by about 2% (to 0.489/0.575). And this increase would be higher with larger cache sizes given there are more number of objects in cache. In addition, LRB's performance is sensitive to the selection of its memory window, which requires extensive tuning.

Comparison on a public general CDN trace

To test HALP's performance on a general CDN workload, we evaluated HALP on a trace from a Wikipedia CDN node [32]. We mimic LRB evaluation settings in cache sizes, the warmup length, and the byte miss reduction metric (Figure 9(a) in LRB), but we converted the object sizes into uni-size. We

select the uni-size to be 32 KiB to match the average request size of the original trace. We compare HALP with the best-performing cache algorithms in LRB evaluations, i.e., LRB, Adaptive-TinyLFU, LeCaR, B-LRU, and LRU. We ran HALP by our simulator, and baseline algorithms by LRB public simulator. For LRB, we use the hyperparameter values described in the paper and its website. Compared to LRB, HALP does not use the additional categorical feature in the trace.

Figure 10 shows the byte miss reduction of different algorithms compared to an industry standard algorithm B-LRU (LRU-eviction policy using a Bloom filter as admission control [24]). None of the algorithms achieves the best performance across all cache sizes. LRB, Adaptive-TinyLFU, HALP, and B-LRU achieve the best performance individually at cache size 64 GiB, 128 GiB, 256 GiB, 512 GiB. At 1024 GiB, the cache is big enough that all cache algorithms converge. At such cache size B-LRU suffers from its admission control. Our observation for this trace is the frequency of objects remains stable over time, making past frequency a reliable indicator of future frequency and allowing the frequency-based heuristic algorithms such as Adaptive-TinyLFU to perform well. In contrast, the workload on YouTube exhibits strong spatial locality, which means that past frequency is less indicative of future frequency, resulting in a lower performance of the frequency-based heuristic algorithms. Note the differences between these results and Figure 9(a) in LRB are likely due to the uni-size object transformation.

5.6 Hyperparameter Selection

We validate the effect of different hyperparameters. This includes different numbers of eviction candidates, different neural network architectures, and different retrain intervals.

Neural network architecture

HALP uses a simple neural network with one hidden layer. Here we vary the number of hidden neurons in the hidden layers and measure the byte miss ratio.

Fig. 11a shows the relationship between the geometric mean of P95 normalized byte miss ratio of all traces as the number of neurons in the hidden layers increase logarithmic from 1 to 256. We see a marginal benefit by increasing the number of neurons up to 8. Beyond this point, more hidden representations do not help. To keep a safe margin, we select the number of hidden neurons in our deployment to be 20.

Number of eviction candidates

HALP uses this parameter in training and prediction. After candidates are selected by the heuristic policy, it iteratively does pairwise ranking to select the final chunk to evict, and later uses these comparisons to generate training data. We vary the number of eviction candidates in the simulation, and measure the byte miss ratio. Note that this changes training and prediction distributions in lock-step.

Fig. 11b shows the relationship between the geometric mean of P95 normalized byte miss ratio of all traces and the number of candidates selected by the heuristic algorithm varying from 2 to 16. As the number of eviction candidates increases from 2 to 4, the byte miss ratio reduces from 60.4% to 59.3%. Further increasing the number of eviction candidates has a marginal effect. Large numbers of eviction candidates have a marginal benefit of the byte miss ratio, but too large a number may harm the byte miss ratio if the other training hyperparameters are not adjusted accordingly.

The number of pairwise comparisons per eviction increases from 3 to 7 when the number of eviction candidates increases from 4 to 8. This increase in CPU does not justify the less than 1% relative byte miss ratio reduction, as a result HALP uses four eviction candidates and does three pairwise comparisons.

Retrain interval

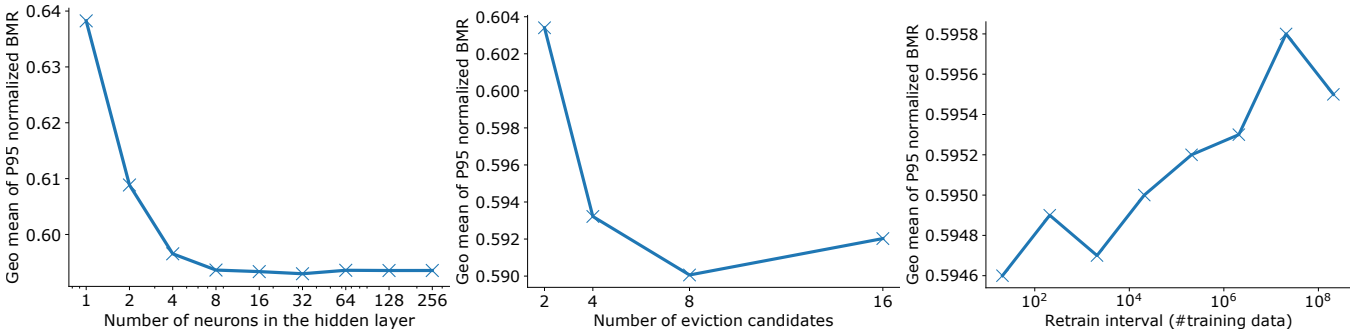
HALP trains online as new requests are processed. We conduct simulation experiments to test different retrain intervals. In order to only test the difference in updating the model online, we increase the trace length to be 6 days from 3 days. We use the first 3 days to train the model in the same retrain interval, and validate that the training loss has been stable. After that, we vary the retrain intervals in the next 3 days, and only measure the byte miss ratio during the latter 3 days.

Fig. 11c shows the relationship between the geometric mean of P95 normalized byte miss ratio of all traces and the retrain intervals. As the retrain intervals increases from processing every 1 new training data input to every 10^8 new training data input, the byte miss ratio slightly increases by less than 0.2%. Our hypothesis is that the traffic pattern change is slow in most traces. But to increase the algorithm robustness, we keep the retrain interval to be every 1024 training data input. This is acceptable in production given the CPU increase is only 1.8% and enables the model to adjust to unpredictable quick workload changes.

6 Related Work

Heuristic-based cache algorithm. Many heuristic-based cache algorithms have been proposed in the past six decades, and from them the most impactful ones include LRU, FIFO, CLOCK [3], SLRU [19], 2Q [18], ARC [25], and TinyLFU [15,16]. Many heuristic algorithms have low computation overhead and provable competitive ratios. But because they are not adaptive enough, they work well in some traces but not in others.

Learned cache eviction. Recently, many learning-based cache algorithms have been proposed to make cache eviction decisions. Table 2 summarizes the state-of-the-art learned cache eviction and admission algorithms. We list four properties of learning-based cache algorithms: target application, whether they are used to make admission or eviction decisions, if they employ online learning, and which underlying machine learning algorithm they employ.



(a) (b) (c)

Figure 11: Geometric mean of P95 normalized byte miss ratio of all traces (a) as the number of neurons in the hidden layers increases logarithmically from 1 to 256. We see a marginal benefit by increasing the number of neurons up to 8. (b) as the number of candidates selected by the heuristic algorithm varies from 2 to 16. As the number of eviction candidates increases from 2 to 4, the byte miss ratio reduces from 60.4% to 59.3%. Further increasing the number of eviction candidates has a marginal effect. (c) as the retrain interval increases by how much training data is processed. As the retrain interval increases from processing every 1 new training data input to every 10^8 new training data input, the byte miss ratio slightly increases by less than 0.2%.

CACHEUS [26] proposes two new heuristic algorithms: SR-LRU, a scan-resistant version of LRU, and CR-LFU, a churn-resistant version of LFU. Then it proposes a regret minimization algorithm to switch between these two experts. As a limitation, the overall algorithm cannot adapt to a new workload if neither of the two experts can adapt to it. In addition, the metadata overhead scales linearly with the number of experts because each one needs to maintain its priority queue. [36] learns next request distribution from tags collected by a distributed tracing framework. It combines a lookup table, a K-Nearest Neighbor approach, and a Transformer model to achieve low overhead and high accuracy. But it has a high learning overhead. LRB [32] uses a regression model to approximate Relaxed Belady, a relaxed oracle algorithm. It uses random sampling to generate eviction candidates and training data. Because of a large number (64) of candidates are needed, the eviction has a high computation overhead. In addition, generating training data with enough critical objects is costly due to the uniform sampling process. And LRB's performance is sensitive to the selection of the memory window (its major hyperparameter).

Parrot [22] and LFO [8] use imitation learning to mimic the oracle algorithm. The objective is to achieve an end-to-end design, but they suffer from a distribution shift. This is because they train their models in an offline fashion, and in practice learning-based cache algorithms have a substantial gap from an oracle, and the objects in the cache as a result differ from a cache that would use an oracle algorithm.

AViC [5] is designed for a video streaming CDN, leveraging the constant speed sequential access patterns, and predicts the time to the next access for the following chunks. However, it has a high implementation overhead because of the complex synchronization between video sessions. Glider [31] targets an eviction policy for CPU caches, and uses an LSTM model for offline analysis. It uses a fast SVM model for an online policy heavily leveraging the program counter (PC) address feature, which is unavailable in the CDN domain.

LHD [6] estimates the hit density of an object between admission and eviction using Bayesian approaches. But it cannot scale with increasing number of features since it does not have a general model for prediction. Predictive Marker [23] is a theoretical work using learning to augment a cache using the Marker algorithm. This idea inspires the design of HALP.

Another line of works [14,20] use reinforcement learning to directly optimize an eviction policy with the target objective. But because cache feedback (hit) can take tens of millions of steps, reinforcement learning approaches suffer from such long feedback and currently have lower performance than supervised learning approaches in practice.

Learned cache admission. In addition to learned eviction policies, many recent research proposed to use learning in cache admission. Cache admission is helpful when a cache has a bottleneck in write constraints (e.g. SSD write amplification and endurance), or a large portion of objects are never reaccessed. The prominent papers include Flashield [17], CacheLib [7], and CacheSack [35]. Because their decision space is more limited than that of eviction algorithms, they have worse performance. HALP's eviction policy can be used jointly with a learned admission policy.

Statistical hypothesis test. Many statistical hypothesis tests have been proposed. But they often focus on using small data, and not on measuring the distribution change. For example, a standard t-test [33] measures the change of mean value.

Algorithm	Year	Target application	Learned admission or eviction	Online learning?	Algorithm
HALP (ours)	2022	CDN	Eviction	Yes	1-hidden-layer MLP, Heuristic + pairwise preference ranking from re-accesses.
CacheSack [35]	2022	Flash cache	Admission	Yes	Greedy optimization
CACHEUS [26]	2021	Storage	Eviction	Yes	Heuristic algorithms w. regret minimization
Learning on distributed traces [36]	2021	Storage	Eviction	No	Lookup Table, K-Nearest Neighbors, Transformers
LRB [32]	2020	CDN	Eviction	Yes	Decision trees
Parrot [22]	2020	CPU	Eviction	No	Transformers
CacheLib [7]	2020	Multipurpose	Admission	No	*Private*
AViC [5]	2019	CDN	Both	No	Decision trees
Glider [31]	2019	CPU	Eviction	Yes	SVM
Flashield [17]	2019	Flash cache	Admission	No	SVM
LHD [6]	2018	KV store	Eviction	Yes	Probability model
LFO [8]	2018	CDN	Eviction	No	Decision trees
Predictive Marker [23]	2018	/	Eviction	/	Learning + Marker algorithm
Harvesting randomness [20]	2017	KV store	Eviction	Yes	Reinforcement learning

Table 2: A summary of state-of-the-art learned cache eviction and admission algorithms

7 Future Work

For future work, we aim to expand HALP to the SSD and HDD caching tiers of the YouTube CDN. We also seek to jointly optimize eviction and admission policies. Another line of future work we plan to explore is to redesign the features and model architecture leveraging existing hardware accelerators. Right now, HALP uses only CPUs and the pairwise comparisons that use the model to pick candidates are subject to the CPU overhead limits acceptable in production. With accelerators like GPUs or TPUs we will be able to explore a larger design space of features and model architectures. One challenge here is how to design an asynchronous batched eviction algorithm to achieve a high utilization of accelerators while preventing it on path of cache operations that require a low latency.

8 Conclusion

This work describes the design, implementation, and evaluation of HALP, a learned caching algorithm that has been deployed to a large-scale production CDN. We also describe an impact distribution analysis method that allows us to measure the impact of deploying a new cache algorithm in a production setting with significant measurement noise. The key insight of HALP is to augment a preexisting heuristic caching policy with machine learning, using the heuristic policy to pick candidates for eviction and the ML model to decide which candidate to evict. The key insight of our impact distribution analysis is modeling machine level measurement noise by comparing machines with HALP deployed against no-op machines.

These design decisions enable HALP's robust byte miss reduction by an average of 9.1%. In addition, these improvements were achieved with a modest CPU overhead of 1.8%.

Acknowledgements

We are grateful to our anonymous reviewers, our shepherd Francis Yan, Ken Barr, Nils Krahnstoever, Jeff Dean, Martin Maas whose extensive comments substantially improved this work. We also thank Yundi Qian and Richard McDougall who contributed to the early stages of the project.

References

[1] LRB open-source simulator. https://github.com/sunnyszy/lrb.

[2] XLA. https://www.tensorflow.org/xla.

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

[4] Vijay Kumar Adhikari, Sourabh Jain, and Zhi-Li Zhang. Youtube traffic dynamics and its interplay with a tier-1 isp: An isp perspective. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, IMC '10, page 431–443, New York, NY, USA, 2010. Association for Computing Machinery.

[5] 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.

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

[7] Benjamin Berg, Daniel S Berger, Sara McAllister, Isaac Grosof, Sathya Gunasekar, Jimmy Lu, Michael Uhlar, Jim Carrig, Nathan Beckmann, Mor Harchol-Balter, et al. The CacheLib caching engine: Design and experiences at scale. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 753–768, 2020.

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

[9] Christopher J. C. Burges, Tal Shaked, Erin Renshaw, Ari Lazier, Matt Deeds, Nicole Hamilton, and Greg Hullender. Learning to rank using gradient descent. *Proceedings of the 22nd international conference on Machine learning*, 2005.

[10] Victor Carbune, Thierry Coppey, Alexander Daryin, Thomas Deselaers, Nikhil Sarda, and Jay Yagnik. Smartchoices: hybridizing programming and machine learning. *arXiv preprint arXiv:1810.00619*, 2018.

[11] Meeyoung Cha, Haewoon Kwak, Pablo Rodriguez, Yong-Yeol Ahn, and Sue Moon. I tube, you tube, everybody tubes: Analyzing the world's largest user generated content video system. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, IMC '07, page 1–14, New York, NY, USA, 2007. Association for Computing Machinery.

[12] Ludmila Cherkasova. Improving WWW proxies performance with greedy-dual-size-frequency caching policy. Technical report, Hewlett-Packard Laboratories, 1998.

[13] Ludmila Cherkasova and Gianfranco Ciardo. Role of aging, frequency, and size in web cache replacement policies. In Bob Hertzberger, Alfons Hoekstra, and Roy Williams, editors, *High-Performance Computing and Networking*, pages 114–123, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

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

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

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

[17] 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.

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

[19] Ramakrishna Karedla, J Spencer Love, and Bradley G Wherry. Caching strategies to improve disk system performance. *IEEE Computer*, 27(3):38–46, 1994.

[20] 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.

[21] Jie Li, Jinlong Wu, György Dán, Åke Arvidsson, and Maria Kihl. Performance analysis of local caching replacement policies for internet video streaming services. In *2014 22nd International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, pages 341–348, 2014.

[22] Evan Liu, Milad Hashemi, Kevin Swersky, Parthasarathy Ranganathan, and Junwhan Ahn. An imitation learning approach for cache replacement. In *International Conference on Machine Learning*, pages 6237–6247. PMLR, 2020.

[23] Thodoris Lykouris and Sergei Vassilvtiskii. Competitive caching with machine learned advice. In *International Conference on Machine Learning*, pages 3296–3305. PMLR, 2018.

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

[25] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-Tuning, low overhead replacement cache. In *2nd USENIX Conference on File and Storage Technologies (FAST'03)*, San Francisco, CA, March 2003.

[26] Liana V Rodriguez, Farzana Yusuf, Steven Lyons, Eysler Paz, Raju Rangaswami, Jason Liu, Ming Zhao, and Giri Narasimhan. Learning cache replacement with CACHEUS. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 341–354, 2021.

[27] Sandvine. The global internet phenomena report january 2022, January 2022. Available at https://www.sandvine.com/hubfs/Sandvine_Redesign_2019/Downloads/2022/Phenomena%20Reports/GIPR%202022/Sandvine%20GIPR%20January%202022.pdf?utm_referrer=https%3A%2F%2Fwww.sandvine.com%2Fphenomena, accessed 06/17/22.

[28] Richard Schooler and Pawel Jurczyk. Streaming media cache for media streaming service, August 27 2019. US Patent 10,397,359.

[29] Nihar Shah and Martin Wainwright. Simple, robust and optimal ranking from pairwise comparisons. *Journal of Machine Learning Research*, 18:1–38, 2018.

[30] Shailesh Shukla. Introducing media cdn—the modern extensible platform for delivering immersive experiences, April 2022. Available at https://cloud.google.com/blog/products/networking/introducing-media-cdn, accessed 09/07/22.

[31] Zhan Shi, Xiangru Huang, Akanksha Jain, and Calvin Lin. Applying deep learning to the cache replacement problem. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 413–425, 2019.

[32] Zhenyu Song, Daniel S Berger, Kai Li, and Wyatt Lloyd. Learning relaxed belady for content distribution network caching. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 529–544, 2020.

[33] Student. The probable error of a mean. *Biometrika*, pages 1–25, 1908.

[34] 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.

[35] Tzu-Wei Yang, Seth Pollen, Mustafa Uysal, Arif Merchant, and Homer Wolfmeister. CacheSack: Admission optimization for google datacenter flash caches. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 1021–1036, 2022.

[36] Giulio Zhou and Martin Maas. Learning on distributed traces for data center storage systems. *Proceedings of Machine Learning and Systems*, 3, 2021.

A Details about the loss function and model weight updates

To explain why we use a cross entropy loss: let w denote the neural network weight parameters and $s_w(f(k,t))$ denote the score output of the neural network for features corresponding to cache key k at time t. Assume that the feedback generation process for the pairwise comparison orders the cache key k_1 ahead of k_2 while querying for the first access to either after time t (i.e. k_1 arrives before k_2 for the first access to either of them after time t). In this case, the cross entropy loss penalizes the loss according to how much the predicted score for k_2 exceeds that of k_1. Specifically, with $\Delta = s_w(f(k_2,t)) - s_w(f(k_1,t))$, as the difference in scores, the neural network weight parameters w are adjusted based on the gradient of the loss function $log(1+e^{\Delta})$. When $\Delta \ll 0$, the loss is close to 0, but when $\Delta \gg 0$, the loss is linear in Δ and varies smoothly around 0.

B Analysis of a simple model for reranking

Let (U,H,L) be a triple of jointly distributed random variables. Let

$$(U_1,H_1,L_1),\ldots,(U_n,H_n,L_n)$$

be id samples $\sim (U,H,L)$. The value H_i corresponds to the score for item i predicted by some (heuristic) ranking policy and similarly, L_i corresponds to the score predicted by, (say a learned) ranking policy L. U_i denotes the true utility from object i, but this is a latent variable. The problem is to choose an index i such that U_i is as large as possible. Define the expected utility of a policy $X \in \{U,H,L\}$ as follows:

$$\mathcal{U}(X) = \mathbb{E}[U_{\arg\max_i(X_i)}]$$

We'd like to pick $i^* \triangleq \arg\max_i(U_i)$, which achieves the optimal utility $\mathcal{U}(U)$, but we only observe (H,L) with U being a latent variable. Given two ranking policies H and L, define an aggregate selection policy on them, $\pi(H,L)$ as a map [2], $\pi : \mathbb{R}^{2n} \mapsto [n]$, and the resulting expected utility as $\mathcal{U}(\pi) \triangleq \mathbb{E}[U_{\pi(H,L)}]$. Next, we describe and analyze a simple aggregation strategy that we've discovered to be useful in learned caching. Let $\lambda_X(i)$ be defined as the item with ranked

[2] $[n]$ denotes the set $\{1,\ldots,n\}$

order[3] i when using the ranking policy X for $X \in \{U, H, L\}$. For each $k \in [n]$, define $\pi_k(H,L)$ as the item chosen when re-ranking the top k items in the heuristic H according to L.

$$\pi_k(H,L) \triangleq \lambda_H(j), \texttt{ where } j = \arg\max_{i \in [k]} L_{\lambda_H(i)}$$

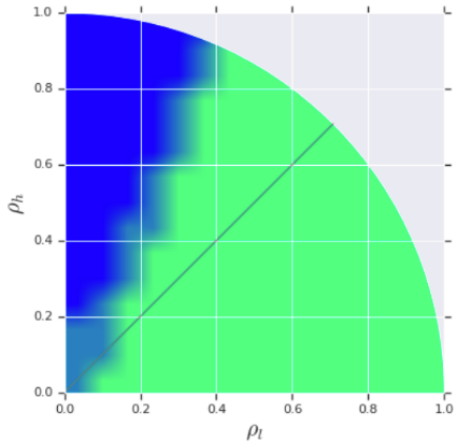


Figure 12: The benefit of rank aggregation evaluated over various configurations for the correlation coefficients based on a sample of $N = 20$ items. The diagonal line indicates points at which $\rho_H = \rho_L$. The green region indicates configurations where $\mathcal{U}(\pi_2(H,L)) > \mathcal{U}(H)$ based on a 95% confidence interval generated from bootstrap estimates of the sample mean. the blue region indicates areas where $\mathcal{U}(\pi_2(H,L)) > \mathcal{U}(H)$ with at least a 95% CI. The grey areas are where the confidence interval overlaps with 0. Interestingly, even when $\rho_H > \rho_L$, it could be advantageous to switch to the lightweight reranking of just the top two items despite having a poor (e.g., learned) policy L. Our experiments indicate that as we increase n, it is better to uniformly switch from H to $\pi_2(H,L)$ for all configurations.

The notation implies $\pi_1(H,L) = \lambda_H(1)$ and $\pi_n(H,L) = \lambda_L(1)$. In other words, $\mathcal{U}(\pi_1(H,L)) = \mathcal{U}(H)$ and $\mathcal{U}(\pi_n(H,L)) = \mathcal{U}(L)$. To understand precisely when the proposed aggregation might help, we now make some assumptions to help with mathematical tractability, analysis and visualization. Let $\rho_H \geq 0, \rho_L \geq 0$ be such that $\rho_H^2 + \rho_L^2 \leq 1$, and consider the jointly Gaussian distribution,

$$(U,H,L) \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} 1 & \rho_H & \rho_L \\ \rho_H & 1 & 0 \\ \rho_L & 0 & 1 \end{bmatrix}\right)$$

It is clear that $\mathcal{U}(H)$ and $\mathcal{U}(L)$ are monotone in ρ_H, ρ_L respectively under the above assumptions. To map the above model to a motivating practical scenario, think of H as an efficient heuristic strategy (e.g. LRU). L could be imagined to be a learned policy that is (1) expensive to evaluate (2) not always safe, i.e. we can end up with $\rho_L < \rho_H$. The proposed mechanism addresses both of these issues simultaneously. For (1) we only need to invoke L on at most e.g. $k = 2$ items, and for (2) the below claim argues that we get an improved outcome, $\mathcal{U}(\pi_2(H,L))$ compared to the baseline strategy $\mathcal{U}(H)$ (which is also naturally greater than $\mathcal{U}(L)$, when $\rho_H > \rho_L$). Based on numerical analysis, we observe, and hypothesize more generally, that as $n \to \infty$, the re-ranking strategy improves over the pure heuristic policy, i.e. $\mathcal{U}(\pi_2(H,L)) > \mathcal{U}(H)$. The above hypothesis applies to arbitrary positive values of ρ_H and ρ_L. It says that we can improve on H even with a worse alternate policy L. This helps give some evidence for why such a strategy appears to be "safe" in terms of improvement over the baseline. In Figure 12, we plot the situation numerically for $N = 20$, for all possible problem configurations of ρ_H, ρ_L.

[3] order 1 is the largest item.