SS-LRU: A Smart Segmented LRU Caching

Chunhua Li[†], Man Wu[†], Yuhan Liu[†], Ke Zhou[†✉], Ji Zhang[†], Yunqing Sun[‡]
[†]WLNO, Huazhong University of Science and Technology, China, [‡]Tencent Technology (Shenzhen) Co., Ltd., China
{li.chunhua, yuhanliu, zhke, jizhang}@hust.edu.cn; 1216575767@qq.com; silviasun@tencent.com

ABSTRACT

Many caching policies use machine learning to predict data reuse, but they ignore the impact of incorrect prediction on cache performance, especially for large-size objects. In this paper, we propose a smart segmented LRU (SS-LRU) replacement policy, which adopts a size-aware classifier designed for cache scenarios and considers the cache cost caused by misprediction. Besides, SS-LRU enhances the migration rules of segmented LRU (SLRU) and implements a smart caching with unequal priorities and segment sizes based on prediction and multiple access patterns. We conducted Extensive experiments under the real-world workloads to demonstrate the superiority of our approach over state-of-the-art caching policies.

CCS CONCEPTS

• **Computer systems organization → Cloud computing**.

KEYWORDS

Cache Replacement, Machine Learning, Smart, Cost-sensitive

ACM Reference Format:
Chunhua Li, ManWu, Yuhan Liu, Ke Zhou, Ji Zhang, Yunqing Sun. 2022. SS-LRU: A Smart Segmented LRU Caching. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22), July 10-14, 2022, San Francisco, CA, USA* ,6 pages. https://doi.org/10.1145/3489517.3530469

1 INTRODUCTION

Caching is one of the most fundamental and effective ways to improve system performance. It is widely used in storage systems, file systems, databases, web servers, and other applications such as data compression and photo sharing [1–3; 9]. With caching, the hot data items are swapped into the high-speed but low-capacity storage layer (i.e, performance layer) from backend storage device via cache replacement algorithms. Therefore, an effective caching strategy is very vital to optimize system performance.

Cache hit rate is one of the most important metrics for evaluating cache algorithm, each algorithm strives to make most accesses hit the performance layer, whether basic LRU and LFU, or more advanced SLRU [4] and ARC [5]. To ensure that the reused items can be kept in the cache, many studies adopt machine learning (ML) algorithms to predict future access probabilities for the current visiting item. If the item is predicted to be revisited in the near future, it will be swapped into the cache or be promoted to the top of the cache according to cache replacement policies. Otherwise, it will be likely to be demoted or discarded from the cache. For example, some researchers adopt decision tree and SVM algorithm to predict whether the visiting web page will be browsed again for web prefetching [6–8]. Vietri G et al.[10] achieve a dynamical policy selection between LRU and LFU based on ML technology. Keramidas G et al. [11] improve the cache performance on CPU by predicting when data will be reused. Wang et al. [1] use the logistic regression (LR) model to predict the future access probabilities of the cached items for picture archiving system. Zhou et al. [2] design a prefetcher based on the observed immediacy feature, but they focus on low-frequency photos' access performance. Wang et al. [3] choose random forest (RF) classifier to make a decision for photo caching admission and propose a "One-time-Access-Exclusion" policy, but the improvement in hit rate is not ideal.

Although the above ML-based cache policies can improve the cache performance to some extent, they have two main limitations. (1) The classification algorithms they adopted are based on the overall distribution of training sample features, without considering the influence of individual features (such as file size) on prediction metric. But for cache scenarios, the sample size is a cache-relevant feature, misclassifying a large-size object will lead to greater cache overhead. (2) For widely used segmented LRU (SLRU) policies, such as S3LRU, they use the same rules to migrate objects in different segments and even allocate the same size to each segment. Thereby, once two cache hits, the object will be upgraded to the highest level segment. As a result, low-frequency access objects may reside a longer time before being swapped out of the cache, while high-frequency objects may be evicted quickly due to their long access intervals. It is obviously unreasonable, especially for objects that happen to be accessed only twice (seen in Section 2.1).

To address the above limitations, we design and implement a smart segmented LRU caching policy called SS-LRU, which utilizes our proposed size-aware classifier *SAdaCost* to make accurate prediction. For objects to be accessed, SAdaCost will impose a higher penalty to ensure accurate prediction if the prediction is wrong. We further design a size-adjustable SLRU cache policy *AS3LRU* which enhances the migration rules and implement a smart SLRU (SS-LRU) based on prediction and multiple access patterns. Our main contributions can be summarized as follows:

(1) We propose a size-aware classification algorithm for caching applications, SAdaCost. To the best of our knowledge, we are the first to consider the impact of cache-relevant features on classification performance.

(2) For SAdaCost, we introduce two cost parameters for two cases of prediction errors, and establish the relationship between them and sample size to achieve accurate prediction for large-size objects. Compared to AdaBoost classifier, SAdaCost improves prediction accuracy by 20.17% and gets 2.86% net gains in terms of hit rate.

DAC'22, July 10-14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530469

(3) We design a smart segmented LRU cache replacement policy, SS-LRU, which can adaptively tune migration rules and segment size. Based on SAdaCost's prediction, SS-LRU improves hit rate by 7.77% over S3LRU.

(4) We implement a smart S3LRU caching simulator and evaluate the prediction performance and cache performance using real-world workloads. Experiments show that our approach outperforms state-of-the-art algorithms.

2 MOTIVATIONS

We first analyze the one-week real-world workloads collected from Tencent[1] QQPhoto Server and obtain some access patterns, then further investigate the impact of misprediction to cache item on cache performance via trace replay. More details about workload features will be described in Section 4.1.

2.1 Some Observations

Observation 1: Table 1 gives the hit contribution of photos with different access frequency. We can see that high-frequency (>10) photos are only a small fraction of all photos, but they get more than 70% of hits. While low-frequency (<=2) photos make up 84.97% of total photos and 37.68% of total requests, but they only get 7.41% of hits. So, it is of great significance to assign higher priority for high-frequency photos and prolong their lifetime in the cache.

Table 1: Access frequency and hit rate contribution (HRC)

frequency	request ratio	photo ratio	HRC
f = 1	25.5800%	68.7091%	0
f = 2	12.1047%	16.2570%	7.4131%
f = (2,5]	13.1884%	9.7673%	11.5242%
f = (5,10]	8.0157%	2.9160%	10.2647%
f = (10,100]	19.8042%	2.2220%	32.9761%
f = (100,1000]	11.2258%	0.1171%	19.8909%
f = (1000,+]	10.0812%	0.0115%	17.9309%

Note: *HRC* is used to reveal the contribution to cache hit of different access frequency objects. It is calculated from the number of hits on the objects within a given frequency range and the total number of hits within a certain time window.

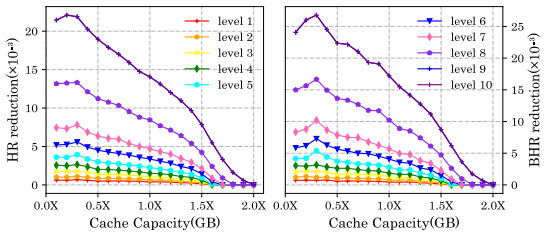
Observation 2: We analyze the hit distance (*HD*) distribution of photos using LRU policy under 1-day's workload traces. The distribution of *HD* reflects how long the requested object resides in cache. We find that 35.49% of hits with 2-frequency have 1-hit distance. That is, about 35.49% of requests will never visit the same photo again after sequential access, which is an interesting finding. It hints us not to promote those objects after sequential access.

Observation 3: In order to explore the impact of misclassified objects with different sizes on cache performance, we combine AdaBoost with S3LRU to conduct experiments. We divide all photos into 10 levels according to their sizes. From Figure 1 we can see that the drop in hit rate (HR) and byte hit rate (HBR) becomes more and more significant as the image size increases. This means that mispredicting large-size objects will incur higher cache overhead than that of small-size objects.

2.2 Motivations

(1) **Higher prediction accuracy for large-size objects**. Based on observation 3, we need a classifier with high prediction accuracy, especially for large-size objects, to reduce cache pollution and access latency caused by wrong prediction, which requires us

[1]Tencent Inc. is the largest social network service company in China, whose QQPhoto is a platform to store and share personal photos for billions of users.



(from level 1 to level 10, image size increases in order. X is 10% of the total photos size)

Figure 1: The relationship between the size of misclassified photos and the reduction of hit ratio and byte hit rate. The bigger the image size, the more significant the decline.

to consider the impact of individual feature (such as file size) on classification results. Although AdaBoost classifier has been proven to have sound prediction result [13], it is modeled based on the whole distribution of training sample features. Besides, it assigns the equal penalty weight for all misclassified objects. In fact, there are two cases for prediction errors. One is mispredicting an object that will not actually be accessed again, i.e, false positive (FP). The other is the opposite, i.e, false negative (FN). These two cases have different effects on cache performance and need to be distinguished. Accordingly, we need a size-aware classifier that can consider both sample size and error type during iterative learning.

(2) **Higher hit rate & low access latency.** Based on observation 1 and 2, we need an uneven promotion rules and segment sizes for segmented LRU, ensuring that objects with a certain frequency range can reside in the cache longer. In addition, according to Huang's research, an 8.5% improvement in hit rate can reduce 20.8% downstream traffics to the backend storage [14]. Therefore, to minimize fetching objects from low-performance storage device, which results in system traffic and user longer waiting, we still strive to improve hit rate.

3 SS-LRU DESIGN

3.1 Overall Architecture

Figure 2 depicts the architecture of our caching system which consists of five parts: Client, Cache Server, SAdaCost Classifier, Backend Storage and Offline Analysis & Training.

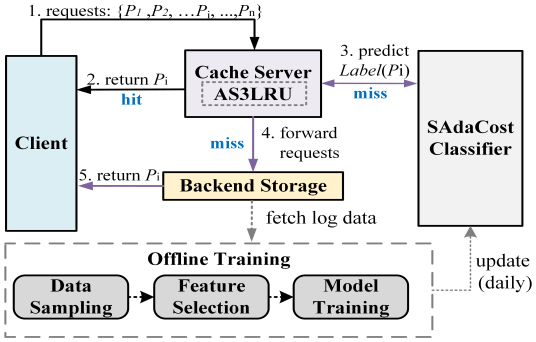


Figure 2: SS-LRU architecture. The combination of SAdaCost and AS3LRU enables a smart SLRU.

SAdaCost Classifier and *Cache Server* are the core components, their combination enables a smart segment LRU caching. *SAdaCost Classifier* serves for *Cache Server* and maintains a size-aware

classification model through offline training. *Cache Server* adopts enhanced SLRU (AS3LRU) to manage the entire cache space, which divides cache space into three segments with unequal size and priority. Each segment is managed by LRU algorithm but observing the promotion rules which will be discussed in Section 3.3.

Offline Training performs data sampling, feature selection and model training, the first two will be discussed in Section 4.1 and the *model training* will be described in detail in Section 3.2. The main task of *Offline Training* is to update the classifier model at a regular interval. According to our analysis on the access pattern of real-world workloads, the daily update model has good stability that is also in good agreement with Sun's conclusion [12], so we choose to update SAdaCost classifier on a daily basis to ensure that the classifier conforms to user access patterns. Since it only takes a few minutes to train model at a time, while the predictor runs at about 0.4us, that is, the impact of the predictor on system performance can be ignored.

As shown in Figure 2, when a request sequence P_1, P_2, …, P_i, …, P_n is reached, *Cache Server* first finds them in the cache, if hits, the photo P_i is directly returned to *Client*. Otherwise, the request is put forwarded to *SAdaCost Classifier* and *Backend Storage* simultaneously. Next, *SAdaCost* predicts whether P_i will be revisited in the near future, according to the prediction result, P_i is then placed to the corresponding segment in the cache. Meanwhile, P_i is retrieved from *Backend Storage* and returned to *Client*.

3.2 SAdaCost Classifier

In this section, we will describe the SAdaCost algorithm, analyze how the two cost-sensitive parameters play a role in the algorithm, and further analyze the design principles and their relationship.

Algorithm Description. SAdaCost is based on AdaBoost algorithm [13], but considers individual feature during training to make it suitable for cache scenarios. Taking a sequence of training objects S as the input, where S=(x_1, y_1), (x_2, y_2),…,(x_n, y_n), x_i and y_i is the feature and label of the i-th instance, respectively. If the i-th instance will be revisited later, y_i=+1; otherwise, y_i=-1.

First, set the weight of each object to $1/n$ (n is the number of objects in the training dataset). Then, train T weak classifiers iteratively. For the t-th iteration, SAdaCost first uses the set of distribution D_t to learn and obtain a weak classifier $f_t(x)$, then calculates the error estimation of $f_t(x)$ on this set with Equation (1):

$$\varepsilon_t = \sum_{i=1}^{n} \frac{D_t|f_t(x_i) - y_i)|}{\sum_{i=1}^{n} D_t(i)} \tag{1}$$

where D_t (i) is the weight of the i-th object in the t-th iteration. After that, the weight of classifier $f_t(x)$ in final classifier can be achieved with Equation (2):

$$\alpha_t = \frac{1}{2} ln \frac{1-\varepsilon_t}{\varepsilon_t} \tag{2}$$

Next, update the weight distribution of training dataset with Equation (3):

$$D_{t+1}(i) = \frac{D_t(i)}{Z_t} \times \begin{cases} exp(-\alpha_t), & if\ y_i = f_t(x_i) \\ exp(\alpha_t)\beta(i), & if\ y_i \neq f_t(x_i) \end{cases} \tag{3}$$

where Z_t is a normalization factor such that D_{t+1} will be a distribution, β_i is the cost-sensitive parameter which is used to adjust the weight for wrong prediction.

At last, the stronger classifier $F(x)$ can be expressed as follows:

$$F(x) = sign(\sum_{t=1}^{T} \alpha_t f_t(x)) \tag{4}$$

Algorithm Characteristic. We distinguish two cases of misprediction when updating the weight. In particular, when tuning the parameter β_i, we consider the size of the requested objects which will not be revisited but are predicted reuse. As shown in Table 2, we set β_i=w_1 when the object incorrectly predicted will actually be revisited, and β_i=w_2 when the object incorrectly predicted will not actually be revisited. For correct prediction, β_i=0. We will further discuss these two cost-sensitive parameters.

Table 2: The cost parameter matrix for SAdaCost

Predict	Actual	
	re-accessed	not re-accessed
re-accessed	$\beta_i = 0$	$\beta_i = w_2$
not re-accessed	$\beta_i = w_1$	$\beta_i = 0$

(1) $\boldsymbol{w_1 > 1}$ **and** $\boldsymbol{w_2 < w_1}$. Accurate prediction can greatly improve cache performance. So, we first try to make accurate prediction for objects that will be revisited. If fails, the weight in Equation (3) should be increased, i.e, w_1 should be greater than 1. Second, we try to improve the prediction accuracy of large objects that will never be accessed again. If incorrectly predicted, they will be populated to the cache and result in cache pollution. The bigger the object, the more costly it is. w_2 is thereby introduced as a cost parameter related to the size of object. To avoid over-classification and causing the large objects to be used cannot be cached, we agree that w_2 should be less than w_1. Meanwhile, w_2 should not approach zero, otherwise the penalty for wrong prediction will disappear.

(2) **Relationship between** $\boldsymbol{w_1}$, $\boldsymbol{w_2}$ **and object size**. In order to minimize the search time for the optimal solution, we simply consider the linear relationship between them, i.e, $w_2 = w_1 \times a$, where $0 < a < 1$. We first control the range of a as (1/2,1), (2/3,1), (3/4,1), (4/5,1), (1/3,1) and (1/4,1), then design six combination relationships based on the size of predicted object *size*(i) and the maximum size of all requested objects *maxsize*. We conduct extensive experiments via trace replay and find that when w_1 and w_2 satisfy Equation 5, both classification quality and cache performance are optimal.

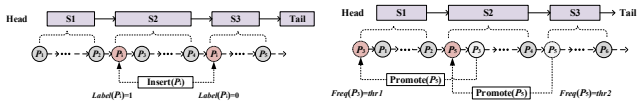
$$w_2 = w_1 \times (1 + size(i)/maxsize)/2 \tag{5}$$

3.3 SS-LRU Policy

The goal of SS-LRU is to keep high-frequency objects in the cache for as long as possible, and to evict low-frequency objects that are no longer accessed as soon as possible. For that, SS-LRU combines multiple features such as recency, frequency, hit distance and prediction result to design replacement policy and promotion rule. Algorithm 1 describes the SS-LRU policy, it includes four basic operations and segment size balancing.

(1) Basic Operations. SS-LRU maintains three LRU segments (S3LRU) with different sizes and priorities by performing four basic operations: insertion, promotion, demotion and eviction. Assume that *S1*, *S2* and *S3* correspond to the highest, middle and lowest level segment, respectively. *S1* stores the most frequently used (MFU) objects, and *S3* stores the least frequently used (LFU) objects. Since the operation of demotion and eviction are the same as S3LRU, we will only describe the operation of insertion and promotion.

Insertion Operation. As shown in Figure 3a, when an object P_i is requested, SS-LRU first updates its frequency (line2). If P_i hits, SS-LRU updates its hit distance and then determines whether to promotion (line3-16). Otherwise, SS-LRU calls *PredictOperation*(P_i) to predict its reuse and then performs an insertion based on prediction result (line22-28). $Label(P_i) = 1$ means that P_i will be used in the near future, so it is placed at the top of *S2*. Otherwise, it is put at the top of *S3*. However, S3LRU puts all new objects at the top of *S3*, increasing the probability that the objects to be accessed will be evicted from the cache quickly.



(a) insertion (b) promotion

Figure 3: Insertion and promotion operation

Promotion Operation. Unlike SLUR's promotion rule "*upgrade on hits*", which makes an object lift to the highest segment after two hits, SS-LRU sets flexible promotion rules for different segments based on multiple access patterns. An object can be promoted to a higher segment only when its access frequency reaches a certain threshold and its hit distance constraint is satisfied. As shown in Figure 3b, when the access frequency of P_5 in *S3* reaches *thr2* and its hit distance meet the lifting rule, P_5 will be lifted to the head of *S2*; otherwise, it is moved to the head of *S3* (line5-9). Similarly, P_3 in *S2* can be promoted to the *S1* when it meets the promotion rules between *S1* and *S2* (line10-14). For objects in *S1*, since they are already in the highest segment, SS-LRU follows the rules of LRU and moves them to the top of the current segment.

(2) Cache space balancing. SS-LRU sets segment size according to statistical analysis and trace replay. When an insert or promotion operation is performed, the segment or cache space may be insufficient. In this case, SS-LRU will perform demotion and eviction operations according to its replacement policy (line30-37). First, evicting objects at the tail of *S3* until it has enough space (line31-33). Second, demoting objects at the tail of *S1* and *S2* to ensure that *CSize1* and *CSize2* keep the proper segment space (line34-37).

Algorithm1 SS-LRU cache replacement policy

Inputs: request sequence: $P = P_1, P_2, \ldots, P_n$; maximum cache size: *MaxSize*; current size of *S1*, *S2* and *S3*: *CSize1*, *CSize2*, *CSize3*; allocation ratio: sr_1, sr_2; promotion thresholds: *thr1*, *thr2*

```
1: for P_i in P:
2:   Update Frequency(P_i)
3:   if P_i in cache:
4:     Update HitDistance(P_i);
5:     if P_i in S3:
6:       if Frequency(P)_i>thr2 and HitDistance(P_i) meets constraint:
7:         Move P_i to the head of S2; CSize2 += Size(P_i);
8:       else:
9:         Move P_i to the head of S3;
10:    elif P_i in S2:
11:      if Frequency(P)_i>thr1:
12:        Move P_i to the head of S1; CSize1 += Size(P_i);
13:      else:
14:        Move P_i to the head of S2;
15:    elif P_i in S1:
16:      Move P_i to the head of S1;
17:  else:
18:    PredictOperation(P_i)
19:  CacheBalance();
20: end for
21:
22: function PredictOperation(P_i)
23:   Use SAdaCost to predict the label of P_i;
24:   if Label(P_i) == 1:
25:     Put P_i into the head of S2; CSize2 += Size(P_i);
26:   else:
27:     Put P_i into the head of S3; CSize3 += Size(P_i);
28:   return None
29:
30: function CacheBalance()
31:   while (CSize1+CSize2+CSize3+Size(P_i))>MaxSize:
32:     Remove the object P_last3 at the end of S3;
33:     CSize3 -= Size(P_last3);
34:   while CSize1>MaxSize*sr_1:
35:     Move object P_last1 at the end of S1 to head ofS2;
36:   while CSize2>MaxSize*sr_2:
37:     Move object P_last2 at the end of S2 to head of S3;
38:   return None
```

4 PERFORMANCE EVALUATION

4.1 Experiment Setup

Dataset. We use QQPhoto access traces for evaluation, which has been collected from Tencent Foshan mobile site and spans from Aug.26 to Aug.30 in 2019. The total number of requests and photos is 5.63 billions and 176 millions, respectively. Each request entry contains the following information: request timestamp, photo ID, image format (jpg, webp, etc.), specification (100×100, 200×200, 512×512, etc), handling time, photo size, terminal type (PC or mobile), time interval from uploading to visiting, and some auxiliary information irrelevant to our research.

To efficiently experiment with such large set of traces without missing their original access patterns, we sample one-thousandth of raw traces as our experimental data set using reservoir sampling method [12]. To verify the reliability of our sampling method, we evaluate the sampled traces with LRU policy used by QQPhoto. The deviation of photo hit rate and byte hit rate between raw and sampled traces is +0.44% and +0.41% respectively. Accordingly, we believe that our sampled traces can faithfully represent the original traces. We conduct experiments on sampled traces and set cache space to be 10% of the total photos size, which corresponds real cache configuration of QQPhoto server.

Evaluation Metrics. We use hit rate (HR) and byte hit rate (BHR) to evaluate cache performance, and choose the commonly used metrics such as accuracy, precision, recall, F1 and AUC to report the classification result. F1 is the harmonic mean of precision and recall, AUC (Area Under the receiver operating characteristic Curve) reflects the ability of classifier to distinguish positive and negative instance. A higher AUC means the classifier has better performance. There are four kinds of prediction results: true positives (TP), false positives (FP), false negatives (FN) and true negatives (TN). TP and TN mean that the predicted results agree with the actual results. Accordingly, the higher the accuracy is, the better the predictor is. To gain sound cache performance, we hope the classification accuracy to be as high as possible.

4.2 Model Parameters

Classifier parameter w_1 and w_2. We first set w_2 =1, then increase the value of w_1 from 1 with a step of 0.2, and conduct experiments on our SAdaCost classifier combined with S3LRU policy. We find that SAdaCost tends to predict more photos to be revisited as w_1 increases, when w_1 reaches a certain value, both HR and BHR reach the maximum value, at this time, the prediction accuracy and AUC also achieve ideal results. Once w_1 exceeds this value, HR and BHR

decline gradually, and so does AUC. In our experiments, w_1=2.8 or 2.6 and w_2 is calculated according to Equation 5.

Cache segment size and promotion threshold. Since the classifier model is updated daily, the values of *thr1* and *thr2* in algorithm1 are determined according to the statistical analysis on the previous day's access records. For instance, *thr1*=5 and *thr2*=2 based on Day1's trace. Moreover, for those objects with 2-frequency and 1-hit distance, we don't move them to the higher segment. As a result, hit rate is improved by 0.58%-2.05%. In addition, the space allocation ratio of S_1, S_2 and S_3 is 1:7:2 based on trace replay experiment, at which both HR and BHR are maximum.

4.3 Performance of SAdaCost Classifier

Prediction Performance. We choose the following seven classification algorithms for comparative experiments: C4.5, Naive Bayes (NB), K-Nearest Neighbor (KNN), Logistic Regression (LR), Random Forest (RF), Bootstrap aggregating (Bagging) and AdaBoost. As can be seen from Table 3 that SAdaCost has obvious advantages in all terms. Compared to AdaBoost, SAdaCost improves accuracy by 20.17% and AUC by 10.93%. NB and LR have a high recall rate but low accuracy. Although AdaBoost and RF have high precision, their recall rate is very low, namely, many photos that will be visited again are predicted not to be visited, resulting in longer access latency. In all, the outstanding prediction results of SAdaCost are expected to bring a significant improvement in cache performance.

Table 3: The performance of different classifiers

Classifier	Accuracy	Precision	Recall	F1	AUC
C4.5	0.4708	0.7933	0.2122	0.3348	0.5595
Bagging	0.4212	0.8391	0.0965	0.173	0.5326
NB	0.652	0.6453	0.9893	0.7811	0.5363
KNN	0.6564	0.7990	0.6047	0.6884	0.6741
LR	0.6391	0.6478	0.9315	0.7642	0.5388
RF	0.4462	0.8692	0.1386	0.2390	0.5517
AdaBoost	0.6174	**0.9935**	0.3931	0.5633	0.6944
SAdaCost	**0.8191**	0.8502	0.8640	**0.8571**	**0.8037**

Note: The details of these algorithms in the table can be found in [15], and all classifiers are implemented by sklearn with default parameters.

Contribution to Caching. In order to investigate the gains of SAdaCost on cache performance, we combine AS3LRU with various classifiers to conduct trace replay experiments. Figure 4 shows the hit rate and byte hit rate of different combination. We can find that SAdaCost-AS3LRU has the highest HR and BHR, while LR-AS3LRU and NB-AS3LRU perform the worst which also coincides with their lowest accuracy in Table 3. When the cache size is at 0.1X, SAdaCost-AS3LRU increases HR by 7.45% and BHR by 8.12% compared with NB-AS3LRU. Comparing to AdaBoost-AS3LRU which has better performance, SAdaCost-AS3LRU also improves HR by 2.86% and BHR by 3.61% at 0.2X, that is, our SAdaCost classifier can get at least 2.86% gains in terms of hit rate.

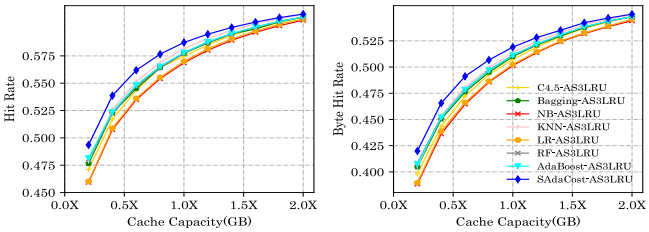


Figure 4: Cache performance of different classifiers combined with AS3LRU

4.4 Cache Performance

Hit Rate (HR). Figures 5 shows how the hit rate varies with the cache size for different cache policies. We can see that our scheme achieves a higher HR than other cache policies. Compared with FIFO and LRU, our method improves HR by 8.64%-29.66% and 4.12%-19.10%, respectively. Even compared with advanced algorithms, SS-LRU also shows great advantages. It raises HR by 2.98%-7.43% over ARC and 2.87%-7.77% over S3LRU. We further calculate the net gains of our AS3LRU compared to S3LRU and find that AS3LRU can gain 2.12% on average. This proves that cache performance is improved after optimizing the segment space allocation and enhancing the promotion rules.

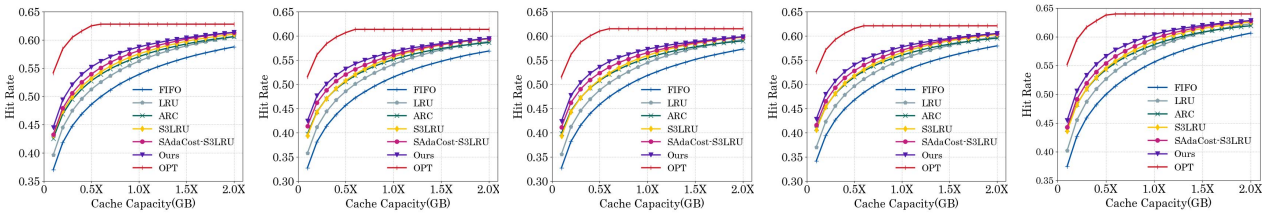
Byte Hit Rate (BHR). Figures 6 shows how the byte hit rate varies with the cache size. As can be seen from the figure, our scheme achieves a higher BHR than other cache policies. Compared with FIFO and LRU, we improve BHR by 9.72%-33.02% and 4.97%-21.68%. Compared with advanced algorithms such as S3LRU and ARC, SS-LRU also increases BHR by 2.09%-9.42% and 3.28%-8.17% respectively. We further analysis the contribution of AS3LRU to performance and get an average gain of 2.38% compared to S3LRU. This further illustrates that our smart segmented cache policy can effectively manage the cache space.

Further, for horizontal comparison with related studies, we compare the hit rate growth of our scheme and that of Wang's method [3]. For the same data set, when the cache size is set to 0.1X, our scheme improves HR and BHR by 7.77% and 9.42% over S3LRU respectively, while Wang's approach only improves by 3.8% and 4.1% respectively. The main reason is that Wang's method adopts RF classifier and filters the objects that are no longer accessed based on prediction results. However, RF does not consider the cost caused by inaccurate prediction. Our scheme not only considers the cost of misprediction, but also optimizes the space allocation and promotion rules according to the access characteristics.

Access Latency. Referring to Wang's method of calculating access latency [3] which includes the time of making a prediction, we systematically test the access latency for different algorithms (shown in Figure 7). It can be seen that although our approach is based on an ensemble learning classifier, it still reduces access latency to some extent. Compared to FIFO and LRU, the access latency is dropped by 9.37%-14.27% and 4.48%-9.89% respectively. Compared with ARC and S3LRU, the access latency is also declined by 1.47%-4.73% and 1.09%-4.85% respectively. This further proves that our scheme can effectively decreases the downstream traffic to the backend storage system.

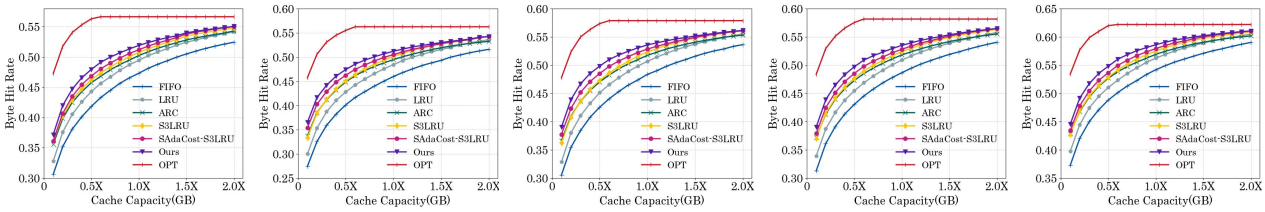
5 CONCLUSION

We propose a smart segment LRU cache replacement policy that manages cache space based on our proposed size-aware cost-sensitive classifier. Our classifier SAdaCost considers the impact of incorrect prediction on the cache performance and increases the penalty for wrong classification by introducing two cost-sensitive parameters. The larger the size of the predicted object, the higher the penalty. Our classifier can get at least 3.61% gains in terms of BHR compared to AdaBoost algorithm. Besides, we enhance SLRU's promotion rule based on access characteristics of real workloads, design a size-adjustable SLRU cache replacement policy, which can earn 2.12%



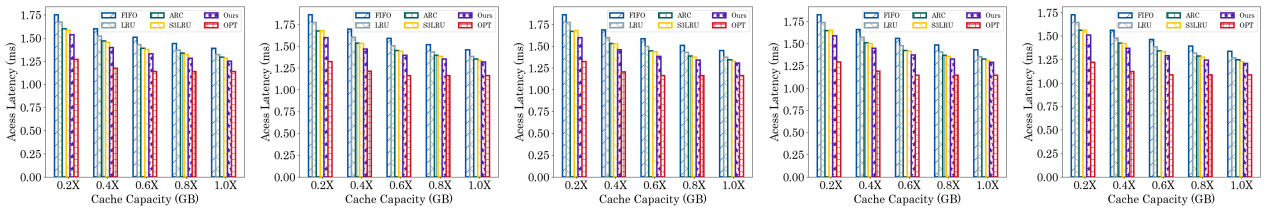
(a) Day1 (b) Day2 (c) Day3 (d) Day4 (e) Day5

Figure 5: Hit rate of different cache policies under different traces of different days. OPT in the figure is an optimal offline algorithm [16] which evicts the object whose future access is the furthest. OPT can help refer to the theoretical limitation that cache policy can reach.



(a) Day1 (b) Day2 (c) Day3 (d) Day4 (e) Day5

Figure 6: Byte hit rate of different cache policies under different traces of different days



(a) Day1 (b) Day2 (c) Day3 (d) Day4 (e) Day5

Figure 7: Access latency of different cache replacement algorithms

in terms of HR compared to S3LRU. In the end, our approach can greatly improve cache performance and decrease access latency over the state-of-the-art cache policies. Our approach can be used for file caching, object caching, web caching and other applications where the requested object size is various.

6 ACKNOWLEDGMENTS

This work is supported in part by the Innovation Group Project of National Natural Science Foundation of China (61821003) and the National Natural Science Foundation of China (62172180).

REFERENCES

[1] Wang Y, Yang Y, Han C, et al. LR-LRU: A PACS-Oriented Intelligent Cache Replacement Policy. IEEE Access, 7:58073–58084, 2019.
[2] Ke Zhou and Si Sun et al. Improving Cache Performance for Large-Scale Photo Stores via Heuristic Prefetching Scheme. IEEE Transactions on Parallel and Distributed Systems, 30(9):2033-2045, 2019.
[3] Hua Wang and Jiawei Zhang et al. Cache What You Need to Cache: Reducing Write Traffic in Cloud Cache via "One-Time-Access-Exclusion" Policy. ACM Transactions on Storage, 16(3):1-24, 2020.
[4] Ramakrishna Karedla and J. Spencer Love et al. Caching Strategies to Improve Disk System Performance. Computer, 27(3):38-46, 1994.
[5] Nimrod Megiddo and Dharmendra S. Modha. ARC: A Self-tuning, Low Overhead Replacement Cache. In Proceedings of the 2Nd USENIX Conference on File and Storage Technologies (FAST'03). USENIX Association, 115–130, 2003.
[6] Chao Wang. Web cache intelligent replacement strategy combined with GDSF and SVM network re-accessed probability prediction. Journal of Ambient Intelligence & Humanized Computing, 11(2):581-587, 2020.
[7] Kuttuva Rajendran Baskaran and Kalaiarasan Chellan. Improved Performance by Combining Web Pre-Fetching Using Clustering with Web Caching Based on SVM Learning Method. International Journal of Computers Communications & Control, 11(2):67, 2016.
[8] Phet Aimtongkham and Chakchai So-In et al. A novel web caching scheme using hybrid least frequently used and support vector machine. In 13th International Joint Conference on Computer Science and Software Engineering. IEEE, 2016.
[9] Ke Zhou and Yu Zhang et al. LEA: A Lazy Eviction Algorithm for SSD Cache in Cloud Block Storage. In 36th IEEE International Conference on Computer Design, pages 569-572. IEEE Computer Society, 2018.
[10] Vietri G, Rodriguez L V, Martinez W A, et al (2018). Driving cache replacement with ML-based LeCaR. HotStorage'18: the 10th USENIX Conference on Hot Topics in Storage and File Systems.
[11] Keramidas G., Petoumenos P., Kaxiras S. (2007). Cache replacement based on reuse-distance prediction. International Conference on Computer Design. IEEE, 245–250.
[12] Ke Zhou and Si Sun et al. Demystifying Cache Policies for Photo Stores at Scale: A Tencent Case Study. In Proceedings of the 32nd International Conference on Supercomputing, pages 284-295. ACM, 2018.
[13] P. Cao, S. Irani. Cost-aware www proxy caching algorithms. In: Proceedings of the USENIX Symposium on Internet Technologies and Systems, USENIX, 1997(12):193–206
[14] Qi Huang and Ken Birman et al. An analysis of Facebook photo caching. In ACM SIGOPS 24th Symposium on Operating Systems Principles, pages 167-181. ACM, 2013.
[15] Ethem Alpaydin. 2014. Introduction to Machine Learning. MIT Press, Cambridge, MA.
[16] Belady and A. L. A study of replacement algorithms for a virtual-storagecomputer.Imb Syst J, 5(2):78–101, 1966.