Cache What You Need to Cache: Reducing Write Traffic in Cloud Cache via "One-Time-Access-Exclusion" Policy

HUA WANG and JIAWEI ZHANG, Huazhong University of Science and Technology
PING HUANG, Temple University & Huazhong University of Science and Technology
XINBO YI, Huazhong University of Science and Technology
BIN CHENG, Shenzhen Tencent Computer System Co., Ltd.
KE ZHOU, Huazhong University of Science and Technology

The SSD has been playing a significantly important role in caching systems due to its high performance-to-cost ratio. Since the cache space is typically much smaller than that of the backend storage by one order of magnitude or even more, write density (defined as writes per unit time and space) of the SSD cache is therefore much more intensive than that of HDD storage, which brings about tremendous challenges to the SSD's lifetime. Meanwhile, under social network workloads, quite a lot writes to the SSD cache are unnecessary. For example, our study on Tencent's photo caching shows that about 61% of total photos are accessed only once, whereas they are still swapped in and out of the cache. Therefore, if we can predict these kinds of photos proactively and prevent them from entering the cache, we can eliminate unnecessary SSD cache writes and improve cache space utilization.

To cope with the challenge, we put forward a "one-time-access criteria" that is applied to the cache space and further propose a "one-time-access-exclusion" policy. Based on these two techniques, we design a prediction-based classifier to facilitate the policy. Unlike the state-of-the-art history-based predictions, our prediction is non-history oriented, which is challenging to achieve good prediction accuracy. To address this issue, we integrate a decision tree into the classifier, extract social-related information as classifying features, and apply cost-sensitive learning to improve classification precision. Due to these techniques, we attain a prediction accuracy greater than 80%. Experimental results show that the one-time-access-exclusion approach results in outstanding cache performance in most aspects. Take LRU, for instance: applying our approach improves the hit rate by 4.4%, decreases the cache writes by 56.8%, and cuts the average access latency by 5.5%.

CCS Concepts: • **Computer systems organization → Cloud computing**;

Additional Key Words and Phrases: SSD, photo caching, machine learning, social network

This article is the extended version of a paper that appeared in *Proceedings of the 47th International Conference on Parallel Processing (ICPP'18)*.

This work was supported in part by the Innovation Group Project of the National Natural Science Foundation of China (no. 61821003).

Authors' addresses: H. Wang, J. Zhang, X. Yi, and K. Zhou (corresponding author), Wuhan National Laboratory for Optoelectronics and School of Computer Science and Technology, Huazhong University of Science and Technology; emails: {hwang, jiaweizhang, orglanss, zhke}@hust.edu.cn; P. Huang, Department of Computer and Information Sciences, Temple University and Huazhong University of Science and Technology; email: templestorager@temple.edu; B. Cheng, Shenzhen Tencent Computer System Co., Ltd.; email: bencheng@tencent.com.

1553-3077/2020/07-ART18 $15.00
https://doi.org/10.1145/3397766

ACM Reference format:
Hua Wang, Jiawei Zhang, Ping Huang, Xinbo Yi, Bin Cheng, and Ke Zhou. 2020. Cache What You Need to Cache: Reducing Write Traffic in Cloud Cache via "One-Time-Access-Exclusion" Policy. *ACM Trans. Storage* 16, 3, Article 18 (July 2020), 24 pages.
https://doi.org/10.1145/3397766

1 INTRODUCTION

Traditional HDDs have long been the main media for information storage. Entering the big data era where high storage performance is highly expected for various applications, storage systems are facing performance challenges. To accelerate the performance of services, flash-memory-based SSDs have been widely used as a cache layer on top of HDD-based storage systems. Currently, research on the SSD buffer cache mainly targets two goals: lifetime extension and performance improvement.

Lifetime of the SSD. A key factor influencing the SSD's lifetime is the amount of writes experienced by an SSD, which we call *write traffic*. Unfortunately, in contrast to serving as a backend storage, the write traffic to a caching SSD is much more significant than the backend storage systems, as the basic cache principle is to absorb popular content. Take a typical caching structure, for instance: one SSD (1 TB), serving as a caching layer, for a backend server array that is composed of 10 HDDs (10*2 TB), the ratio of write density (defined as the number of writes per unit time and space) on the caching SSD to the underlying storage system is approximately 20:1, assuming that accesses to the storage space are uniformly distributed. Serving as a caching media instead of backend storage renders an SSD subject to extensive writes. Therefore, how to judiciously manage the amount of write becomes a crucial problem for caching SSDs. Otherwise, a caching SSD would wear out quickly due to intensive writes.

At present, most of the research on the lifetime extension of an SSD has focused on the following aspects: optimizing garbage collection [5, 46] and the wear leveling mechanism [7, 13, 21], improving ECC robustness [16, 26], employing a write cache to reduce SSD writes [30, 44], and building the FTL based on content locality [6, 42]. Although those approaches are effective in extending SSD lifetime, our proposed approach in this work is complementary to them and reduces unnecessary writes to caching SSDs by enforcing an admission policy, which could in the first place potentially avoid unnecessary cache writes.

Performance of the SSD. Improvement of SSD performance is mainly achieved through two ways: hardware and algorithm. Ideally, people want an adequately large SSD space to accommodate all data. In reality, the SSD cache space can only be much smaller. However, replacement algorithms can greatly affect caching performance, from the basic LRU, FIFO, and LFU to the more advanced S3LRU [22], LIRS [19], and ARC [27], each one excelling some locality, such as temporal/spatial/content locality, therefore being appropriate for certain scenarios, and there is no "one-size-fits-all" algorithm.

Tencent Inc. is the largest social network service provider in China, whose online communication software, WeiXin and QQ, serve billions of clients, where the amount of data is huge and, more importantly, increases steadily and fast. Take QQ, for example: until August 2017, there were 2 trillion photos worth 300 PBs, with a daily increase of 300 million photos, and 50 billion daily views. To cope with such a scale of intensive accesses, cache servers are configured with an SSD. During the past 2 years, we've been working with the company's engineering team on optimizing Tencent's photo buffer cache. Originally, we tried to expand cache space and apply different cache algorithms, which unfortunately led to minimal improvements because the current cache space is already very large. We further analyzed the traces of the photo cache and found that a large number of photos are accessed only once, but they are still swapped into the cache like other photos, which causes cache inefficiencies. This observation inspired us in that if we can avoid those photos entering the SSD cache, we can greatly reduce SSD writes. In doing so, cache performance will benefit from the increase of cache utilization.

To materialize this idea, we found that the biggest challenge was in the prediction of future visits. Most existing prediction methods are based on historical information, but the accesses we need to predict in this case are one-time-access, which has no historical information for reference. As a result, the accuracy of prediction using existing methods is hard to guarantee. However, service types and users are diverse and varying, which leads to dynamic and regularly unpredictable access patterns. Therefore, it is not realistic for the prediction to be implemented in a certain way. After various attempt efforts, we realize that machine learning provides a good solution in our scenario, as it not only excels at massive data analysis but also adapts to dynamic workloads and automatically adjusts feature selection and prediction, which makes it appropriate for non-history-oriented prediction. We experimented with seven mainstream machine learning methods, and we chose the decision tree in our classifier.

The contributions of our research include the following:

(1) We propose an adaptive "one-time-access-exclusion" policy, and based on that, we implement an admission policy for a caching system. To the best of our knowledge, we are the first to propose the criteria of "one-time-access".
(2) We suggest to use machine learning methods to predict future photo accesses. To address the challenge of unavailability of historical information, we rely on feature extraction and training model optimization to increase the prediction accuracy to reach greater than 80%.
(3) We implement a caching classifier married with the one-time-access-exclusion policy, and experimental evaluations demonstrate that it leads to impressive improvements in all of the following aspects: hit rate, SSD writes, and access latency.

The rest of this article is structured as follows. Section 2 presents the overview of Tencent photo caching and the motivation for our research. Section 3 describes the application of machine learning to caching. Section 4 describes the design and implementation of intelligent caching. Section 5 presents our experimental results. Section 6 discusses the related works, and we conclude in Section 7.

2 BACKGROUND AND MOTIVATIONS

2.1 Tencent Photo Caching

QQ is a highly popular instant messenger software developed by Tencent Inc., and the QQ album is the application hosting photos for QQ clients. At the end of 2017, QQ album stored more than 2 trillion pictures for about 850 million active users, and it experienced 50 billion views per day.

Figure 1 illustrates the architecture of QQPhoto, which is the backend photo storage system for the QQ album. The most distinguished feature of QQPhoto is that the upload and download channels are separate from each other. It should be noted that the upload and download here are relative to the client, which means that upload and download correspond to the client's write and read operation, respectively. There are two reasons for adopting this approach. One reason is due to an app-specific feature: one single uploaded photo will receive many or even a huge number of visits. The other reason is related to the displaying requirement: to adapt to different types of client terminals, several versions in different resolution categories, such as a, b, c, m, l, and o (more detail will be discussed in Section 3.4.1), would be generated for an original photo after uploading. As an overall access feature of QQPhoto, download traffic is about 1,000 times more intensive than that

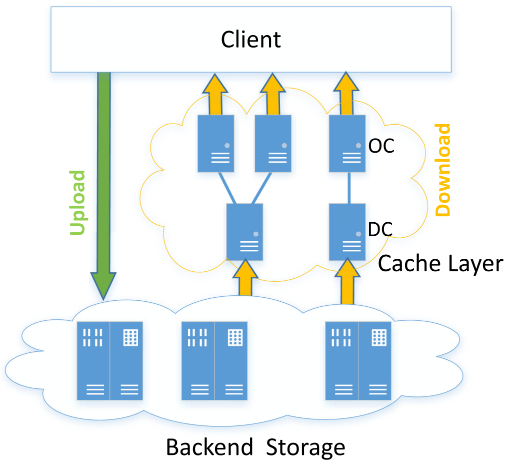


Fig. 1. Architecture of Tencent photo caching.

of the upload traffic. To shield the dramatic traffic difference and prevent read/write disturbance, the upload and download channels are set to be independent of each other.

Since the download traffic is much heavier than the upload traffic, and the download speed will be more influential to the clients' experience on the cloud service, it is crucial to configure the buffer cache in the download channel. There are typically two layers of caching: the outside cache layer (OC) and the datacenter cache server (DC). OC sits close to the client and is responsible for providing prompt response to clients' requests, whereas DC is located at the data center, aiming to relieve the traffic burden to backend storage. Both layers are equipped with SSDs, forming a distributed buffer cache to handle the extraordinarily large number of photo requests generated by numerous clients.

2.2 Observations and Main Idea

Photo caching efficiency is crucial to the performance of the QQ album platform. To optimize photo caching, it is necessary to understand the appropriateness of the current caching architecture to the QQ album application, so as to discover the aspects that could be improved.

We collect and investigate a 9-day-long access log of the QQ album's download channel, totaling 1,481,617,402 photos and 5,856,501,598 access records, to obtain insights into the characteristics of the QQ photo workload. We make our observations via analyzing traces and simulating trace experiments.

Observation 1: Many photo files are accessed only once. Our analysis on the trace has shown that 910,568,705 photos, accounting for 61.5% of the total, were accessed only once. Since there are 25.5% of compulsory misses, which indicates cache miss will occur upon the first requests to photo files, even assuming the cache space is infinitely large, the cache hit rate will be capped at 74.5%, which is called a *ceiling glass* phenomenon. And among the whole accesses, those to the one-time-access photos occupy 15.5%. In traditional caching methods, all one-time-access photos are swapped into the cache when accessed, which not only pollutes cache space but also generates a large number of unnecessary SSD writes, accelerating SSD wear.

Besides the preceding analysis on the trace, we also conducted an investigation on the cache performance running on the trace. Generally, both the cache replacement algorithm and cache size are two key factors affecting cache performance. We want to experimentally demonstrate

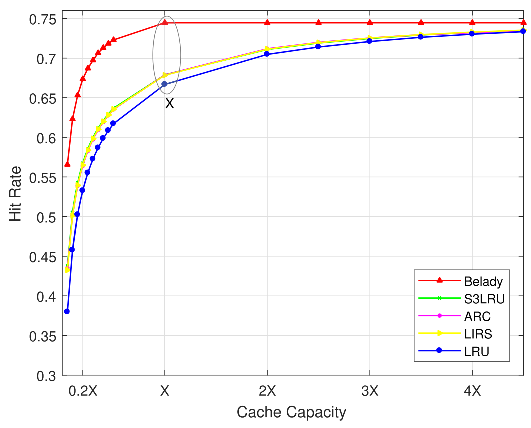


Fig. 2. Hit rate under different cache capacities.

how these two factors affect cache performance of the specific QQPhoto trace. We conduct experiments on the collected trace using four common and popular cache algorithms: LRU, S3LRU, ARC, and LIRS. In addition, for the comparison with the ideal case, we also include the results of the Belady algorithm [35], which generates the optimal upper bound cache performance. The experimental results are shown in Figure 2, where the x and y axes represent cache capacity and hit rate, respectively. We can gain two other observations from the figure.

Observation 2: Existing cache algorithms perform nearly equally and are limited in further improving cache performance. Except for Belady sitting far away from the other curves, there are negligible differences among the other algorithms on the hit rate. Specifically, S3LRU, ARC, and LIRS almost have the same hit rate and outperform LRU by only about 1%. Moreover, the gaps between Belady and other algorithms are about 9% at X cache size and gradually decrease to 4% when cache space increases to 4X. This observation implies that optimizing the cache replacement algorithm only has limited efficiency in improving cache performance.

Observation 3: Increasing cache size does not benefit cache performance. As can be observed from Figure 2, for all cache algorithms, as the cache space increases, the cache hit rates see diminished benefit returns. The inflection point occurs at cache size X, which is particularly obvious for Belady algorithm. After the inflection point, the curves appear to be horizontally stable. Overall, after the cache size reaches a certain size, further increasing cache size does not improve the cache hit ratio but only increases the cost. The stagnation of the hit rate for infinite cache space is attributed to the ceiling glass phenomenon mentioned earlier.

In summary, our preceding observations demonstrate that in this specific application scenario, we should resolve to look for novel approaches other than centering around the conventional wisdom of cache designing (e.g., choice of algorithm and cache capacity increase) to improve cache performance.

2.3 Main Idea and Challenges

When used as the cache, SSDs are subject to intensive writes that will accelerate the SSD's wearing speed and shorten its lifespan. Fortunately, the QQ album application contains a large number of one-time written photo files (Observation 1), which does not help cache performance if cached and thus could be eliminated from the cache. Meanwhile, optimizations such as using a more advanced cache algorithm (Observation 2) and increasing cache capacity size (Observation 3) do not generate further improvement toward the Belady's optimal bounds due to the ceiling glass phenomenon, which leads us to improve cache utilization from different perspectives. Based on Observation 1, we propose a one-time-access-exclusion policy, which excludes photos that would not be accessed in a foreseeable future to enter into the cache. Doing so brings dual benefits, such as improve cache utilization and reduce SSD cache writes.

The key point to the one-time-access-exclusion policy is to determine whether a photo access is one-time-access or not when the photo is accessed, which is challenging to implement. First, making a prediction here is non-history oriented, completely different from traditional history-based prediction, which is a relatively well-researched topic and has many effective solutions. Second, the amount of concurrent data access is huge, and the workload changes dynamically as time goes on, demanding prediction to be adaptive.

Applying machine learning in computer architecture is an emerging area of research. Machine learning excels in massive data analysis, and more importantly, it is capable of automatically adjusting feature selection and updating the training model according to the dynamic workload. Therefore, machine learning is highly suitable for non-history-oriented prediction.

3 APPLICATION OF MACHINE LEARNING TO THE CACHE

To prevent one-time-access files from entering the cache, it is necessary to predict if the currently accessed file is a one-time-access file or not. The prediction target here is one-time-access file, which has never been accessed before, and therefore no historical information can be leveraged. Non-history-oriented prediction is more difficult than history-based prediction, which has been studied a lot [9, 31, 45]. Furthermore, there is another problem that increases the difficulty of non-history-oriented prediction in the cloud storage service environment where service types and users are diverse and varying, which results in dynamic workloads. Therefore, it is not realistic for the prediction to be handled in a certain fixed way. Machine learning methods not only accomplish efficient analysis for massive data but also adapt to dynamic workloads and automatically adjust feature selection. There is a recent research that adopts machine learning in optimizing dynamic tradeoffs in NVMs [10].

3.1 Data Sampler

The function of the data sampler is to obtain the training data and tag them—that is, to observe some photos randomly for a period of time and decide whether they are one-time-access files or not. The sampler selects photos that need to be observed with a certain probability $p_{sampler}$. If the reuse distance of the photos is less than or equal to M, whose computation method is explained in Section 3.2, it is marked as a non-one-time-access file. Otherwise, the photo file is marked as a one-time-access file. If there are multiple visits to the same file, only the features of the first visit of the file are considered as a training sample. In other words, the sampler only processes the first visit of a missed photo file. Instead of processing each visit, processing is done on every missed file.

The reason the sampler only considers the first visits is twofold. First, in the cache architecture, the classifier is mainly used to distinguish one-time-access files, and the photo with multiple visits within a short term does not need to be judged by the classifier once the photo has been cached. Second, during the processing of decision tree classification, which is used in our approach, each leaf node corresponds to a subset of training samples, and if the subset contains more photos of the same type, it is more likely to be identified as the corresponding photo types. If we treat each visit instead of the missed file as a training sample, it is more likely that there will be a larger proportion of non-one-time-access files, which will affect the accuracy of the classification results.

The maximum number of photos observed by the sampler is

$$C_{sampler} = M \times (1 - h) \times p_{sampler}, \tag{1}$$

where h represents cache hit rate, $M \times (1 - h)$ denotes the number of new files in nearly M visits, and then is multiplied by the sampling rate to obtain the size of the sampler space. The sampler uses a FIFO strategy for cache eviction.

3.2 Criteria for One-Time-Access

As mentioned before, the key to the one-time-access-exclusion policy is to determine one-time-access photos, for which we should establish the criteria of one-time-access first.

In the nearly 9 days' worth of collected QQPhoto logs, about 61.5% of the photos are accessed once and account for 15.5% of the total access records. If we can remove those once-accessed photo files, we can reduce 15.5% of SSD cache writes. However, using this straightforward identification method, there will also be considerable photos that are not actually accessed again throughout their whole cache life if their reaccess distances are large enough. In other words, this method will bring about false-negative judgment. Therefore, we need more sophisticated and accurate criteria.

Since cache space has a limited capacity and can only hold a maximum number of photos, more realistic and accurate criteria should account in the number of cached photos and the cache-residing period of the photos. If the reaccess distance of a photo is larger than its cache-residing period, which equals to the time interval from the moment of entering the cache to the moment of being evicted from the cache, then the photo is identified as a one-time-access photo.

Suppose that the lower bound of the reaccess distance to be judged into one-time-access is M, which means that if the reaccess distance of a photo file is larger than M, then the photo would be identified as a one-time-access photo. We next explain how to determine the value of M.

The key point of the new criteria is to determine the reaccess distance (denoted as M) of a photo file, which is defined as the number of successive accesses between the time it is brought into the cache and the time it is accessed again. The problem comes down to determine the value of M based on which the file can be classified.

Assume that the cache hit rate is h, then there are $M(1 - h)$ misses out of M accesses, for a full cache in a normal working state, which means that $M(1 - h)$ times of replacements will occur. Take the LRU algorithm, for example: suppose that cache size is C and the average size of one photo is S, then a one-time-access photo will be evicted from the cache after C/S times of replacement, resulting in the following equation:

$$M(1 - h) = C/S. \tag{2}$$

Therefore, $M = C/[S(1 - h)]$.

We aim to filter out as many one-time-access files as possible, which means that not all $M(1 - h)$ missed files will enter the cache. Suppose that the percentage of one-time-access files is p, which is the ratio of the number of one-time-access requests to the number of total requests, then M times of access will cause $M(1 - h)(1 - p)$ times of replacement. Therefore, the preceding equation can be refined as follows:

$$M(1 - h)(1 - p) = C/S. \tag{3}$$

Performing a trivial transformation, we have $M = C/[S(1 - h)(1 - p)]$.

From the preceding equation, we know that with a constant h, M and p are positively correlated, which implies that M increases with p. However, from the perspective of the photo caching system, a larger M means more rigorous judgment criteria for one-time-access, and files whose reaccess distance is larger than M become fewer, which indicates that p will be smaller. Therefore, the trend can be summarized as $p \uparrow \rightarrow M \uparrow \rightarrow p \downarrow$. Likewise, there also exists the trend of $p \downarrow \rightarrow M \downarrow \rightarrow p \uparrow$.

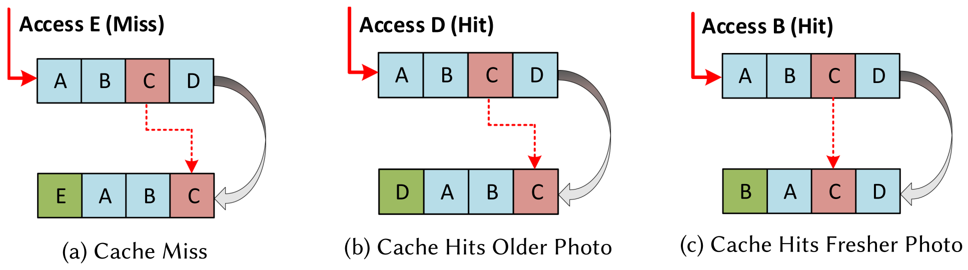


Fig. 3. Three cases of a cache access. Cases of (a) and (b) would change the position of a photo (file C) in the cache, whereas case (c) causes no change to the cache position.

We can calculate M and p based on the trend analyzed earlier. If p is set to be 0 initially, a corresponding M can be computed. Using this M, p can be obtained through testing the real-world traces, and after several iterative processes, p converges to a constant. Empirically, we set the iterations to be 3. Similarly, we can also estimate the value of h.

To be more rigorous, in the LRU algorithm, not only the replacement caused by a cache miss will demote $Photo_i$ denoted as C in Figure 3(a) but also hitting a file older than $Photo_i$ will cause demotion of $Photo_i$, as shown in Figure 3(b). Only the third case, as shown in Figure 3(c), will not change its position in the cache. Now assuming that the probability of hitting photos older than $Photo_i$ is r, $M \times h \times r$ demotions will occur. As a result, the equation of M could be refined as follows:

$$M(1-h)(1-p) + M \cdot h \cdot r = C/S. \tag{4}$$

Therefore,

$$M = \frac{C}{S[(1-h)(1-p) + h \cdot r]}. \tag{5}$$

For simplicity, we use Equation (3) in our experiments, and we explore the values of M, p, and h for different cache sizes in Section 5.1.

3.3 Classifying Algorithm

3.3.1 Selection of the Classifying Algorithm. There are many machine learning algorithms that can be employed for prediction. Our goal is to find a good tradeoff between high precision and low complexity. To achieve this goal, we conducted extensive comparisons among existing mainstream classifying algorithms. We sampled from the log dataset, retaining 100 records in every minute out of a total of 144,000 records, and finally collected approximately 70,000 sampled trace records. Then we split the sampled dataset into a training dataset and a testing dataset through cross validation. Table 1 gives the performance of the tested classifiers, and Figure 4 shows their ROC curves. For the sake of narrative clarity, we list machine learning metrics in Table 2 and Table 3. As can be seen from Table 1, decision tree and ensemble learning methods (e.g., AdaBoost, Random Forest) all achieve good classifying accuracy. Intuitively, ensemble learning, which skillfully integrates multiple decision trees, is a better choice for the classification algorithm because of better accuracy. However, we found that when using an AdaBoost/Random Forest ensemble learning classifier, even if the number of base learners (decision tree) is increased to 50, it only improves the accuracy by 1% while bringing about 50 times the computational cost. In terms of classifier selection, we follow two guidelines: one is that the classifier should have a higher AUC, which means a lower false positive and a higher true positive. The other is that the classifier should be simple enough and its complexity low. According to the criteria, we choose the decision tree [3] as the final classifier,

Table 1. Performance Comparison of Different Classifiers

Algorithm	Precision	Recall	Accuracy	AUC
AdaBoost	0.842	0.876	0.854	0.854
Random Forest	0.847	0.850	0.846	0.846
Decision Tree	0.846	0.841	0.842	0.842
BP NN	0.754	0.879	0.794	0.793
KNN	0.760	0.797	0.770	0.769
Naive Bayes	0.563	0.991	0.606	0.602
Logic Regression	0.603	0.505	0.582	0.583

Note: The details of these algorithms in this table can be found in the work of Alpaydin [1].

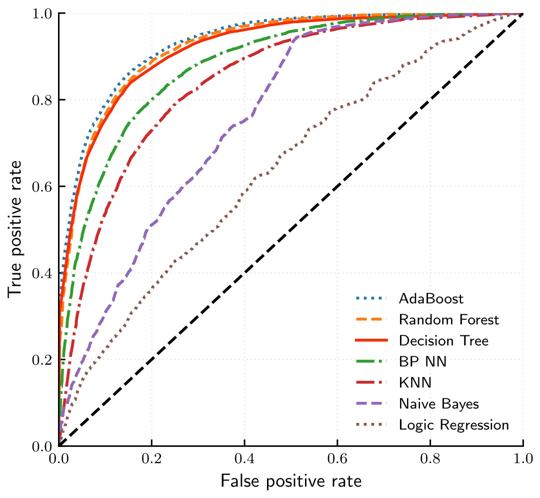


Fig. 4. ROC curve of different classifiers.

whose classification performance is almost the same as Random Forest but its complexity is much lower.

3.3.2 Configuration of the Classifying Algorithm. To avoid decreasing generalization performance caused by overfitting, we set the upper limit of splitting times to 30 for the decision tree, which is approximately three times the number of features. The prediction complexity of a decision tree classifier is highly related to the height of a tree. After comprehensive tests, we found that the height of a tree in our classifier model is 5 in most cases, and therefore, in the worst case, only five comparisons are needed to complete prediction.

3.4 Feature Extraction

The implementation effect of the prediction algorithm rests with feature extraction to a great extent, let alone that our prediction is non-history-oriented, which is much more difficult than existing history-based predictions.

3.4.1 Feature Overview.

(1) Photo owner's social information:

Active friends: The number of users who have interacted with the photo owner in the recent past.

Table 2. Confusion Matrix for Prediction Result

Reality	**Prediction Result**	
	Positive	Negative
Positive	TP (true positive)	FN (false negative)
Negative	FP (false positive)	TN (true negative)

Table 3. Definitions of Metrics in Machine Learning

Metric	**Definition**
Precision (P)	P = TP / (TP + FP)
Recall (R)	R = TP / (TP + FN)
Accuracy	The proportion of the number of correctly classified samples to the total number of samples
ROC curve	A curve whose vertical axis is the true-positive rate and lateral axis is the false-positive rate
AUC	Area under the ROC curve

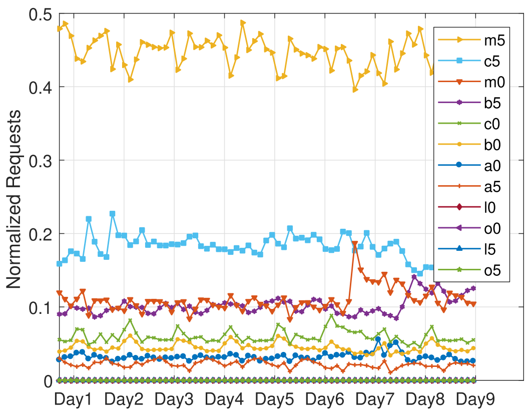


Fig. 5. Number of requests for different types of photos.

Average views of the owner's photo: Ratio of the total photo views of all images of the owner to the number of the user's photos.

(2) Photo information:

Photo type: The photo type is embodied by the resolution and specification of the photo. There are six resolutions (a, b, c, m, l, and o) and two picture specifications (i.e., png and jpg, represented by 0 and 5, respectively), the combination of which leads to a total of 12 photo types. There are significant gaps in the number of requests among different types of photos, as shown in Figure 5. Among them, *l5* has the most requests, accounting for about 45%. For a certain type of photo, the access probability is relatively stable in general, but it will fluctuate over time. This phenomenon implies that the time feature should be considered in the classifier.

Photo size: Photo size has strong correlation to the resolution of the image. In general, for the same image, the higher the resolution, the larger the photo.

Table 4. Feature Importance Measured by Decision Tree

Feature Category	Feature	Feature Importance
Social information	Average views of owner's photo	0.7288
	Active friends	0.0014
Photo information	Photo access recency	0.1580
	Photo age	0.0820
	Photo type	0.0271
	Photo size	0
Information of cache system	Access time	0.0025
	Terminal type	0.0002
	Recent requests	0

Photo age: The age of the picture measured by the time interval between the current time point and the uploaded time point of the given photo. Intuitively, newer pictures are more popular. Experimental results also confirm the intuition.

Recency: The time interval between the current photo access time and its last access time. If the picture has never been accessed, we use the time interval between the photo access time and its uploaded time.

(3) Information of the cache system:

Terminal type: The terminal type mainly includes the personal computer (PC) and mobile device.

Recent requests: The number of requests accepted by the system in the last minute. Recent requests can indirectly reflect the activity of the whole user group, and a higher number means a more active user group, whose average access probability of each photo is accordingly higher.

Access time: Time in a day when accesses happen. Users often use QQ at a relatively fixed time, generally at 8:00 PM, which also means that photos are accessed at different time intervals with different probabilities.

3.4.2 *Approach of Feature Extraction.* Initially, all n features form a full set, $\{a_1, a_2, \ldots, a_n\}$, and we use the information gain (the greater the information gain, the more helpful to classification) to quantify the quality of each feature, and then we choose the optimal feature, a_i, which has the biggest information gain, to construct the goal set, $\{a_i\}$, and remove a_i from the original full set. Again, the information gain of each feature in a full set is judged and the optimal feature, a_j, is determined and transferred from the full set to the goal set—that is, $\{a_i, a_j\}$. If the goal set $\{a_i, a_j\}$ is superior to the previous $\{a_i\}$, which means that the effect of the new classification is better, then the process will be repeated accordingly. Otherwise, the process stops.

Based on this method, we choose the average views of the owner's photo, photo access recency, photo age, photo type, and photo access time in our final feature set. Moreover, we can obtain the feature importance after constructing the decision tree with the whole feature set, and Table 4 shows the results.

Feature importance here is defined as the normalized total reduction of impurity brought by the given feature, which could be calculated by Equation (6). W_j is the weight of $node_j$ and denotes the number of samples reachable to $node_j$, and E_j is the impurity value of $node_j$ denoted by information entropy. The impurity reduction of $node_j$ can be expressed as the weighted impurity of $node_j$ minus its left child's and right child's weighted impurity. Finally, we could obtain the importance of $feature_i$ as the sum of impurity reduction of all nodes.

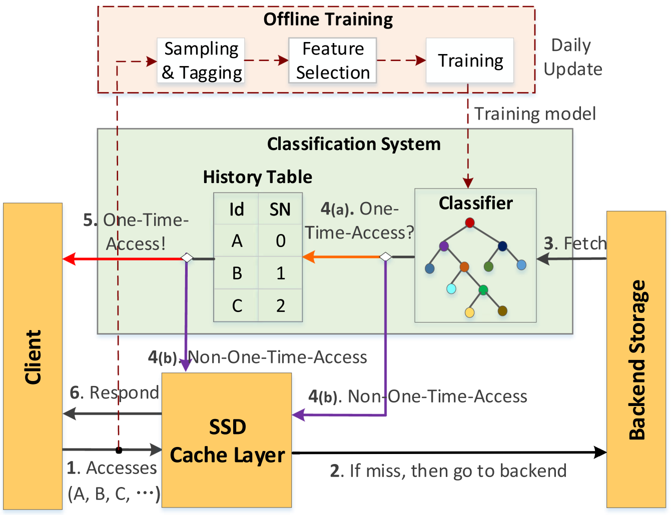


Fig. 6. Cache architecture with classification.

$$FI_i = \sum_j [W_j \cdot E_j - W_{left(j)}E_{left(j)} - W_{right(j)}E_{right(j)}] \tag{6}$$

3.4.3 Feature Processing. Feature processing comprises data tagging, discretized processing, and processing of time value.

Data tagging labels for all of the one-time-access files. The definition of one-time-access was explained in Section 3.2, in which the LRU algorithm is used. The definition is also applicable to other advanced algorithms, such as ARC, S3LRU, and FIFO.

Discretized processing includes two parts. One is mapping 12 types of photos (a0, a5, b0, b5, c0, c5, m0, m5, o0, o5, l0, l5) to 12 discrete values (1–12). The other is mapping the PC and mobile device to 0 and 1, respectively. Since decision tree models can handle these categorical variables, one-hot encoding them is unnecessary.

Processing of the time value mainly aims at access time and time interval. Access time is represented by hour of the day, because the workload in the social application is highly correlated to human activity and demonstrates some kind of periodicity. Photo age and recency are processed in the granularity of 10 minutes, which accelerates training process convergence.

4 INTELLIGENT CACHING ARCHITECTURE

4.1 Overall Architecture

To present more clearly, we abstract the Tencent photo cache (Figure 1) into three simple layers: client, cache, and backend storage, on which the additional components named as the classification system and offline training are supplemented. The whole architecture is shown in Figure 6. The classification system only records the accesses in the memory, and training is done offline. Therefore, it does not introduce noticeable impact to system performance. The process flow of client requests proceeds as follows:

(1) The client generates photo accesses (A, B, C, ...), which go to the cache layer. If hit, the requested photo is returned to the client and the request is completed. Otherwise, it goes to step 2.
(2) The request is forwarded to backend storage.

(3) The requested photo will be fetched from backend storage and sent to the classification system, where the classifier makes a prediction on the photo file to determine whether it is a one-time-access photo or not. If it is predicted to be one-time-access file, then it goes to step 4(a). Otherwise, it proceeds to step 4(b).
(4) (a) The history table is queried to determine if the photo is indeed one-time-access or not. If so, the photo will be returned directly to the client (step 5). If not, it proceeds to step 4(b).
(b) The requested photo is processed in a traditional way: cached and returned to the client (step 6).

It should be noted that when the photo is fetched from the backend storage, the corresponding features are packaged together, and then the returning of the photo to the client and the prediction of the photo to be one-time-access or not can be done in parallel, and the photos to be cached are temporarily saved in a reserved memory buffer, which will be flushed to the SSD cache when the buffer becomes full.

During the procedure, prediction conducted by the classifier is the most important, so an appropriate classifying model need to be employed, which is obtained through offline training. The offline training encompasses three functions: sampling and tagging, feature selection, and training. To guarantee prediction performance, the model is updated daily. Since the classifier is mainly based on an offline classification policy, the overhead of prediction is small and the influence to cache performance can be neglected.

4.2 History Table

The history table plays an important role in correcting any misclassifications made by the classifier. A misclassification here refers to the classifier false negatively classifying a photo file as one-time-access, which if not corrected would cause a cache miss for the photo file. The history table is implemented using the C++ HashMap structure, which uses chaining to resolve conflict internally. The history table stores information of the photos that were classified as one-time-access recently, and the information of each entry includes FileID (the key), PrevNode, NextNode, and AccessSequence, totaling 32 bytes. PrevNode and NextNode link the entry to a FIFO list. For a photo in the history table, if the reaccess distance is smaller than M, it means that the photo was misclassified into a one-time-access photo last time. Therefore, it will be regarded as a non-one-time-access photo this time and removed from the history table. We set the capacity of the history table stored in DRAM to $M(1-h)p \times 0.05$, nearly 2% to 5% of the SSD cache meta-data table, and evict items using a FIFO policy.

The history table keeps track of one-time-access files. If a file is accessed again, and its reaccess distance is less than M, then the previous classification is judged to a misclassification. Only if the decision tree classifies a file as a one-time-access file, and after a while the history table tells us that there is no misclassification, can the file be treated as a one-time-access file. The history table is more like performing a post-classification corrector, remedying misclassifications. If the history table is not used, the classification accuracy of the system will decrease, and the hit rate of the cache system will decrease. We quantitatively evaluate the impact of the history table on cache performance in Section 5.4.

4.3 Guarantee Mechanisms for Quality of Prediction

To ensure prediction quality, we adopt a cost-sensitive learning approach, the history table, and a daily updating model. The cost-sensitive learning approach and the history table try to reduce the negative impact of false predictions on the cache system. Both of these methods increase the

Table 5. Cost Matrix

Actual	**Predict**	
	Positive	Negative
Positive: one-time-access	0	1
Negative: non-one-time-access	v	0

confidence of the prediction results for the one-time-access photos that we care about. Meanwhile, the training model is updated daily to ensure that the classification aligns with the most recent user access pattern.

4.3.1 Cost-Sensitive Learning Approach. There are two kinds of errors of misclassification: one is classifying one-time-access into non-one-time-access (i.e., false negative), and the other is opposite, classifying non-one-time-access into one-time-access (i.e., false positive). The first error will lead to cache space wastage, whereas the second one will cause subsequent cache miss of the given file. To alleviate the cost caused by such errors, we employ a cost matrix [12], as shown in Table 5.

In a distributed cache system like QQPhoto, an even slightly higher hit rate has importance, since a higher hit rate leads to shorter response time and lower network traffic between the cache and backend storage, and as we pursue to minimize the unnecessary writes of SSD without sacrificing the cache hit rate, we believe that the second type of error causes a higher penalty than the first one, $v > 1$. When cache space is lower, the penalty of the second error is relatively higher and vice versa. The choice of v values is {1.5, 2, 2.5}. The method of selecting the value of v is introduced in the next section.

4.3.2 Selection of the Classifier's Working Point. The working point determines on which point the classifier is ultimately within the ROC curve. A common method of evaluation is to use the F_β function, whose computation method is shown as Equation (7):

$$F_\beta = (1 + \beta^2) \cdot \frac{precision \cdot recall}{\beta^2 \cdot precision + recall} \tag{7}$$

F_β represents the harmonic mean of precision and recall. If $\beta = 1$, it indicates that both precision and recall are of the same importance. If $\beta > 1$, it means that recall is more important than precision; otherwise, precision is of more importance. In our scenario, misclassification of a file as non-one-time-access file access only results in an increase in the number of SSD writes. However, if the file is wrongly classified as a one-time-access file, it will result in cache misses to later accesses, which decreases the hit ratio. We believe that the hit rate should have a higher priority than the amount of SSD writes. Therefore, the first case is acceptable and the second one should be avoided as much as possible. Since precision is more important, β is set to be smaller than 1.

Specifically, assigning a value to β should also consider cache space. When the cache space is small, on average, files can stay in the cache for a relatively shorter length of time. Therefore, the number of hits in a small space is lower than that in a large space. This also means that with a small cache space, the cache miss penalty is relatively low due to the misclassification of non-one-time-access files. Therefore, β is higher in a small space than that in a large cache space. By using the F_β function, we choose the value of v in the cost-sensitive matrix, and we set it as 1/1.5, 1/2, and 1/2.5 at 2 GB, 20 GB, and 40 GB, respectively. Please note that the cache space corresponds to 100 times the size of a realistic cache due to our 1:100 sample method in our experiments. The value of β in other spaces is calculated by the quadratic function determined by these three points.

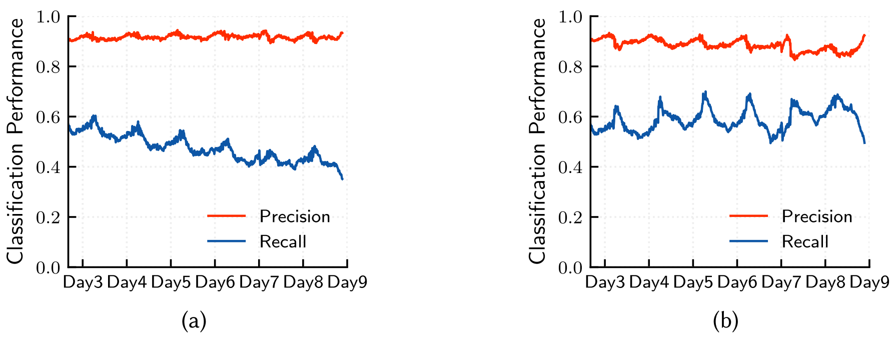


Fig. 7. Comparison of classifier performance using the single updated model (a) and the daily updated model (b).

4.3.3 Daily Updating Model. We adopt the cost matrix and implement training on the dataset described in Section 3.3.1, then apply the classifier into the whole trace. We find that classifying performance drops significantly over time, as shown in Figure 7(a), implying that the prediction accuracy is time bounded.

There are two solutions to this problem. One is incrementally updating the classification model in a real-time manner. The other is an offline learning manner, such as updating the classifier at intervals. We choose the second one, which minimally affects cache performance. To determine the update cycle, we analyzed the percentage of one-time-access, p, and found that p changes at daily periodicity, reaching the highest and the lowest at 5:00 AM and 8 PM, respectively, every day. Considering the system load and the time required for labeling, we train the classifier at 5:00 AM each day, using sampled data of the previous 24 hours before 5:00 AM. For example, on Day3, we use the model trained with the sampled data of Day1, because the labeling and training of the sampled data of Day1 is only completed on Day2. Due to sampling and efficiency of the CART decision tree algorithm, the entire training procedure takes only a few minutes. Then the classifier is used to classify data of the next day. Figure 7(b) shows the classifier performance using the daily updated model, and Figure 7(a) shows the classifier performance using a single training for the whole period of time. As can be observed, using the daily updated model keeps the recall rate stable during the whole trace time, whereas using the single updated model, the recall rate gradually decreases as the workloads deviate from the training dataset as time goes on.

5 EXPERIMENTAL RESULTS

5.1 Test Design

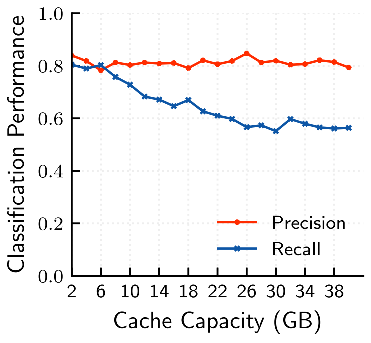
The original trace contains 5.8 billion records, with a total data volume of 50 TB. To simplify processing, we randomly sampled trace data using the following steps. First, distinct objects were extracted to construct an object set L. Second, random sampling at 1:100 was conducted on L, and object set L' is obtained. Then records included in original dataset and whose object id belong to L' were extracted to construct a new trace sequence according to the timestamp. Our final sample dataset comprises about 14 million objects. We conduct our experiments on the sampled trace and range the cache space from 2 GB to 40 GB, which correspond to about 200 GB and 4 TB realistic cache space, respectively. Our test environment is featured as follows: Intel Xeon CPU E5-2609@2.5 GHz, 32 GB of memory, a 4-TB SSD.

Before introducing the experimental results, we list some parameters (M, h, p) of our system under different cache capacities in Table 6, taking the LRU algorithm, for instance. We find that as the cache capacity increases, cache hit rate h increases, and the window for observing whether a

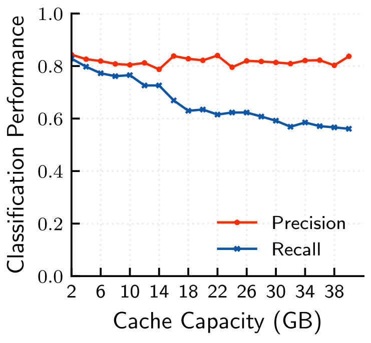
Table 6. Parameter Configurations of Different Cache Capacities

Cache Capacity (GB)	h	p	M	Size of History Table (KB)
2	49.46%	51.11%	265,226	107
6	59.04%	41.64%	822,514	219
10	62.90%	37.76%	1,419,116	311
14	65.19%	35.48%	2,042,682	394
18	66.77%	33.93%	2,686,562	473
22	67.95%	32.79%	3,346,550	550
26	68.88%	31.89%	4,019,695	623
30	69.65%	31.19%	4,707,415	696
34	70.28%	30.61%	5,402,522	768
38	70.83%	30.08%	6,104,562	837

Note: Detailed calculation methods for these parameters can be found in Sections 3.2 and 4.2.



(a) LRU



(b) LIRS

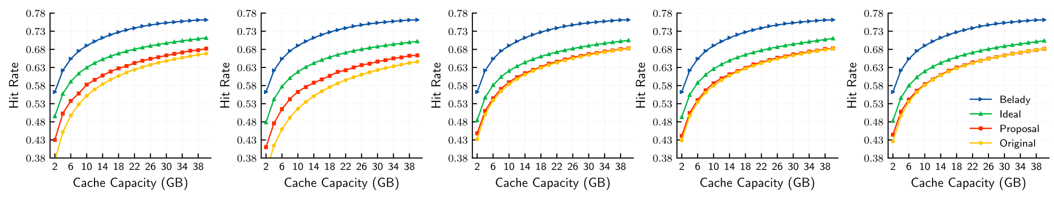
Fig. 8. Performance of the classification system.

file is one-time-access will also be larger, which means that the M value will also become larger, and it directly leads to more files being classified as non-one-time-access, so p gets smaller and smaller. In addition, it can be seen that the history table always occupies only a small amount of memory space of not more than 1 MB.

5.2 Experimental Results for the Classification System

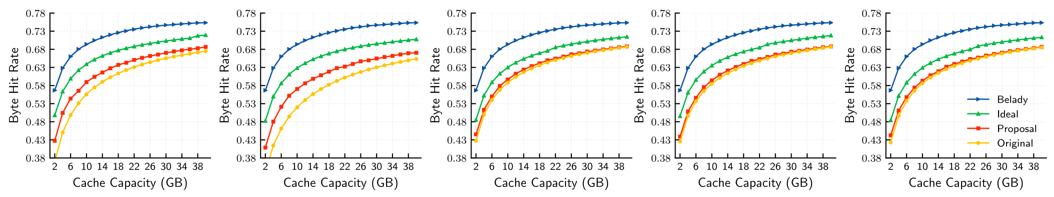
We choose the following cache algorithms in our experiments: LRU, ARC, S3LRU, LIRS, and FIFO. As discussed in Section 3.2, one-time-access criteria for LRU, ARC, S3LRU, and FIFO are the same, whereas the criteria for LIRS is somewhat different, for which $M_{LIRS} = M_{LRU} \times R_s$, where $R_s = C_s/C$. C_s means the cache space of STACK(S), reserving blocks with smaller inter-reference recency, and C means the whole cache space. Without loss of generality, we conduct experiments for two algorithms, LRU and LIRS, whose results are shown in Figure 8(a) and Figure 8(b), respectively.

We can see from the figure that the prediction accuracy of LIRS is slightly better than that of LRU. The reason is that $M_{LIRS} < M_{LRU}$, which means that the LRU algorithm needs to predict further into the future, causing it to be more difficult to guarantee prediction accuracy. However, in Section 5.3, we will find that the improvement for the LRU algorithm using classification prediction is even more pronounced than LIRS. That is attributed to two factors: one is that the LRU algorithm has more room for improvement, and the other is that advanced algorithms, such as LIRS and ARC,



(a) LRU (b) FIFO (c) S3LRU (d) ARC (e) LIRS

Fig. 9. File hit rate of cache replacement algorithms.



(a) LRU (b) FIFO (c) S3LRU (d) ARC (e) LIRS

Fig. 10. Byte hit rate of cache replacement algorithms.

have their own strategies in compensating the negative impacts of one-time-access files, and thus higher classification accuracy is required for further improvement.

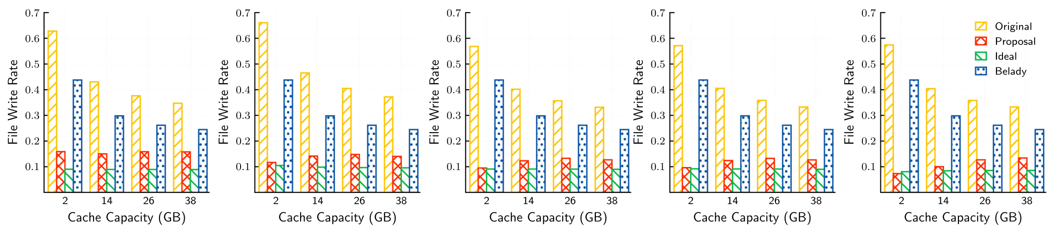
5.3 Experimental Results for the Caching System

In this section, we present the experimental results of the file hit rate, byte hit rate, write rate, byte write rate, and response time for LRU, FIFO, S3LRU, ARC, and LIRS, respectively. In each group of comparisons, *Belady* provides the upper limit of cache performance, *Original* means the corresponding cache replacement algorithm (e.g., LRU, FIFO, S3LRU, ARC, and LIRS), *Proposal* represents the cache system using our classifier, and *Ideal* denotes the cache system using ideal classifier with its classification accuracy being 100%.

5.3.1 File Hit Rate. When cache capacity ranges from 2 GB to 40 GB, we can see from Figure 9 that after using the classifier, the promotions of FIFO and LRU are the most significant, which are 2.7% to 20.9% and 1.9% to 15.5%, respectively, whereas for advanced algorithms, such as S3LRU, the improvement is much less, which is 0.1% to 3.8%.

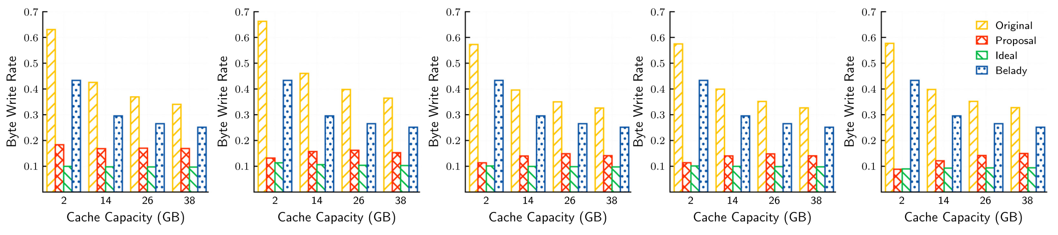
In addition, with the increase of cache capacity, the improvement of the hit rate become smaller. The reason is that when cache capacity increases, the judgment threshold (M) for the one-time-access file is raised and the classifying performance drops, which leads to lower improvement of the hit rate.

5.3.2 Byte Hit Rate. Different from the file hit rate, the byte hit rate takes the file size into account as well and represents the ratio of hit data volume to the whole accessed data, which can indicate the throughput performance of the cache system. Similar to the hit rate, using the classifier, the promotions of FIFO and LRU are quite noticeable, reaching 2.6% to 21.1% and 1.7% to 15.8%, respectively, but it is only 0.2% to 4.1% for S3LRU, as shown in Figure 10. Most photos of the QQ album are of approximately the same size, and meanwhile our classifier is not sensitive to the file size, and therefore we have not observed significant differences between the hit rate and byte hit rate.



(a) LRU (b) FIFO (c) S3LRU (d) ARC (e) LIRS

Fig. 11. File write rate of cache replacement algorithms.



(a) LRU (b) FIFO (c) S3LRU (d) ARC (e) LIRS

Fig. 12. Byte write rate of cache replacement algorithms.

5.3.3 File Writes of the SSD. With the traditional caching method, every file miss will cause the file to be written to the SSD cache. In our proposed cache system, once a file miss occurs, the file will be written to the SSD cache only when it is predicted not to be a one-time-access file, which can decrease the number of SSD writes. We can see from Figure 11 that the number of file writes drops significantly for all replacement algorithms if the classifier is applied. Among them, LIRS obtains the most write reduction of 59.7% to 87.3%. The write rate represents the written data to the SSD divided by the total amount of accessed data.

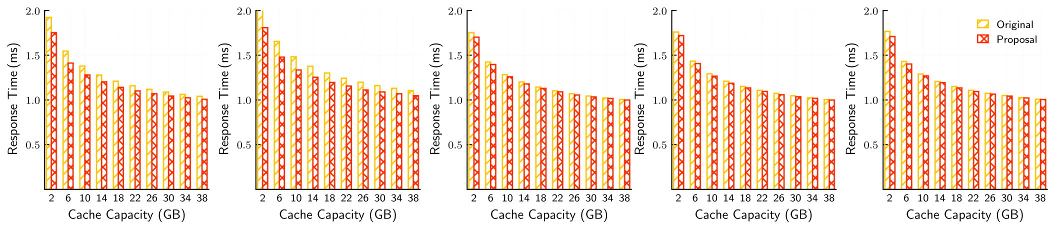
5.3.4 Byte Writes of the SSD. Byte writes consider the file size, representing the amount of total data written into the SSD. Similar to file writes, byte writes are reduced for all cache algorithms, achieving 54.2% to 84.6% for LIRS, as shown in Figure 12.

It can be noticed un-expectantly that in Figure 11 and Figure 12, the write rate of *Proposal* is slightly lower than that of *Ideal* on the LIRS algorithm. This is due to misjudgments of the classifier, which mistakenly regard some non-one-time-access files as one-time-access files, and rejects their entries into the cache, thereby reducing the amount of SSD writes, but it sacrifices certain cache hit rates. Moreover, as mentioned in Section 5.2, a single model has been trained for the LIRS algorithm, and therefore this phenomenon only occurs on LIRS but not on other replacement algorithms.

5.3.5 Comparison of Performance. Since the classification process will inevitably impact the response time of the cache system, now we discuss it qualitatively. For the sake of simplicity, notation $t_{classify}$ represents the execution time of the classification system consisting of the classifier and history table. Generally, the system-level average access latency in a cache system can be calculated using Equation (8), which combines the overall cost of the hit cases along with that of miss cases.

$$T = hit\ rate \times Hit\ cost + (1 - hit\ rate) \times Miss\ penalty \tag{8}$$

For a file access, if the file hits in the cache, the hit cost of the original caching system without prediction is the same as that of our proposed system, which is the sum of query time (t_{query}) and



(a) LRU (b) FIFO (c) S3LRU (d) ARC (e) LIRS

Fig. 13. Response time of cache replacement algorithms.

read time (t_{ssdr}) from the SSD and is expressed as follows:

$$Hit\ cost = t_{query} + t_{ssdr}. \tag{9}$$

In the case of a file miss, the miss penalty of the original system includes the time of querying the cache and reading data from the HDD to DRAM, which can be represented as Equation (10). Please note that writing data to the SSD is not taken into account since it can be done in the background:

$$Miss\ penalty_o = t_{query} + t_{hddr}, \tag{10}$$

where t_{hddr} represents the completion time of HDD reading.

In our proposed system, if the current file is predicted to be a one-time-access file, it will be read from the HDD and be forwarded to DRAM directly. Otherwise, it will be dealt with in the same way as the common system without classification. The missing penalty of the proposed system only includes the querying cache, classification, and reading data from the HDD to DRAM. Therefore, the missing penalty of the proposed system can be represented as Equation (11):

$$Miss\ penalty_p = t_{query} + t_{classify} + t_{hddr}, \tag{11}$$

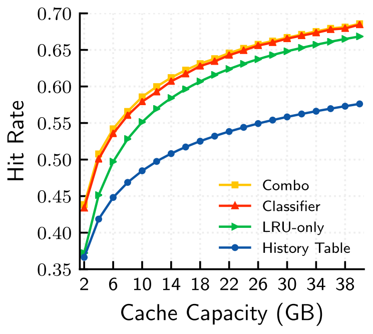
where $t_{classify}$ indicates the execution time of classification.

In this manner, we can compare the average access time for a photo of a no-classification system with that of our proposed system. For a 32-KB picture, we set $hddr = 3$ ms, $t_{query} = 1\ \mu$s, and $t_{classify} = 0.4\ \mu$s. Please note that t_{query} and $t_{classify}$ are the results of our experiment. We can see that *Proposal* outperforms *Original* in Figure 13, and the most improvement is shown with FIFO, 4.5% to 10.8%, the least is LIRS, 0.3% to 3.2%. Because there is a significant gap of read performance between the SSD and HDD, a higher hit rate will result in a lower average access latency, as shown in Equation (8), and the overhead of the classifier seems to be minimal. With the help of the classifier, the hit rate of the cache system is improved a little, as shown in Section 5.3.1, which ultimately leads to a better overall response time for the proposal system over a no-classification system.

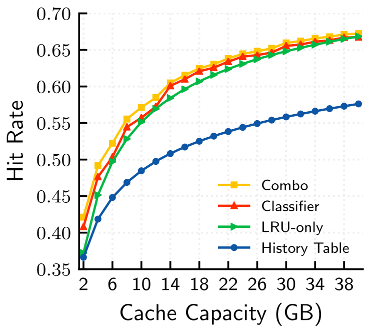
5.4 Breakdown of the Experimental Results for the Caching System

On first sight, readers may wonder why we need to recheck the one-time-access files classified by the classifier using a history table. Therefore, it would be interesting to see the effectiveness of the history table in ensuring cache performance. As mentioned in Section 4.2, the role of the history table is to reduce the impact of the classifier's misjudgment of one-time-access files on the cache hit rate. Intuitively, the higher the classification accuracy is, the less effective the history table would be.

To illustrate the effect of the history table in different classifiers, we employ two different decision trees with different classification performance. The accuracy of the better classifier is about 80%, whereas the poor classifier's accuracy is about 70%. Figure 14 compares the hit rates of the



(a) Hit Rates using a better classifier



(b) Hit Rates using a poor classifier

Fig. 14. Hit rates using two kinds of classifiers.

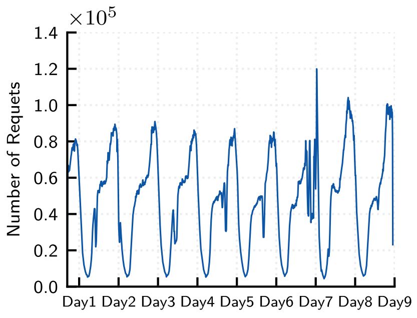


Fig. 15. Number of requests in every 10 minutes.

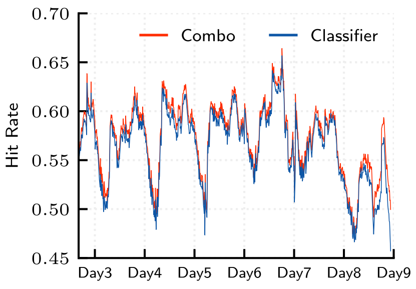


Fig. 16. The trend of hit rate in every 10 minutes.

two classifiers. The four curves in Figure 14 represent the original LRU algorithm (*LRU-only*), the LRU algorithm with the history table (*History Table*), the LRU algorithm with the classifier (*Classifier*), and the LRU algorithm with both the history table and the classifier (*Combo*), respectively. The history table improves the hit rate by 0.17% to 1.17% for the better classifier (Figure 14(a)), respectively, and 0.59% to 3.80% for the poor classifier (Figure 14(b)).

Furthermore, we use the daily update model, which is based on the condition that the daily access patterns are similar, and the 9-day access patterns shown in Figure 15 also coincide with this intuition, except for a burst in Day7 that could affect the performance of the classifier. Therefore, it is difficult to maintain the stability of the classifier's judgment accuracy all the time. To further illustrate how the history table helps alleviate this problem, we compared the real-time hit rate trends of *Combo* and $Classifier$ under poor classifiers, as shown in Figure 16. It can be found that the effect of the history table on Day9 is obvious and the corresponding hit rate is improved by 0.04% to 12.35% relatively. The reason is that the model used by Day9 was trained on the sampled data of Day7.

Therefore, we can see that using a marginal amount of memory could ensure that the overall performance of the cache system remains stable in both cases when the classifier has a good and poor predictive performance.

It is interesting to observe that the performance of *History Table* is the worst in Figure 14. This is because an admission policy is applied to the LRU algorithm, and then only objects in the history table can be admitted into the cache, and since the observation window of the history table is small, which is just about 2% to 5% of the meta-data table in the SSD cache, the update of cached objects is very slow. In other words, the history table acts just as a post-classification corrector instead of a pre-admission index, as in LARC [18].

6 RELATED WORK

Our work focuses on optimizing photo cache performance and extending SSD lifetime by filtering the one-time-access photos from the cache. In this section, we mainly discuss related work in three categories: the first is caching, especially photo caching in social networks; the second is image popularity prediction; and the last one is data access pattern analysis.

6.1 Caching

Caching has been an active research topic for many years. Traditional cache replacement algorithms are designed to make use of different characteristics of access patterns such as recency (LRU) and frequency (LFU), and some advanced replacement algorithms attempt to exploit both, such as SLRU [22] and ARC [27]. Based on these traditional algorithms, there has been much research on how to further optimize cache performance for specific scenarios [15]. Cidon et al. [8] and Yang et al. [43] discuss how to improve caching performance in the web environment. Zhou et al. [48] propose a lazy eviction scheme for a scenario where there are lots of files whose reuse distance is larger than the cache size. Keramidas et al. [23] and Kharbutli and Solihin [24] focus on predicting when data will be reused to optimize the cache performance on the CPU, whereas Jiménez and Teran [20], Peled et al. [28], and Teran et al. [34] leverage machine learning techniques to achieve similar goals. Their main idea is to imitate the Belady algorithm [35], which is estimating reuse distance to achieve a higher hit rate, whereas we filter unnecessary data to improve the utilization of the cache space. In addition, it is more difficult to predict reuse distance in our scenario because the data locality and the correlation between features and reuse distance are weak. Jiménez and Teran [20], Kharbutli and Solihin [24], and Qureshi et al. [29] investigate bypass cache policies to improve cache performance, whereas none of them provides criteria to determine what kind of data should not be put into the cache. Instead, the work in Eisenman et al. [11] is designed to look for objects that may be read frequently and store them on the SSD to minimize writes to the SSD. Huang et al. [18] record the access information through a ghost cache and dynamically adjust its size according to the hit rate to ensure that hot files are in the cache.

In recent photo caching research, Huang et al. [17] give a detailed analysis of a Facebook photo caching service stack. Bai et al. [2] discuss how to improve the performance of processing different resolution images in the photo cache. Tang et al. [32] try to solve the problem of random write to Flash when an advanced cache algorithm is applied to photo caching. Zhou et al. [47] analyze the access behaviors of QQPhoto in detail and propose a prefetching strategy to improve cache efficiency. We start from different perspectives and leverage the user's behavior to optimize the photo caching system, enriching the research work in this area.

6.2 Image Popularity Prediction

In recent years, popularity prediction of network content has attracted many research interests. Khosla et al. [25] point out that social information of users is of great significance for image popularity prediction. Based on their work [25], Gelli et al. [14] introduce sentiment and context features, and analyze the influence of different sentiments on image popularity. Most of those works focus on popular photos and assume that currently popular photos are more likely to be popular later. Unlike them, we try to identify cold photos and apply them to a real-world problem. It is worth mentioning that Tang et al. [33] apply popularity prediction to a large-scale production environment. In their work [33], the popularity of videos on Facebook is predicted so that the quality of most watch time can be improved, along with reduced overhead.

6.3 Data Access Pattern Analysis

Cache optimization and image popularity prediction heavily rely on the analysis of data access patterns. Breslau et al. [4] and Huang et al. [17] reveal that the access pattern of objects in the cloud environment shows Zipf-like or Pareto distribution. Yang and Zhu [41] found that Zipf's law also exists in write traffic through the analysis of write count distributions in real-world traces. Guo et al. [37] investigate media access pattern and find that media access workloads follow the stretched exponential (SE) distribution, instead of Zipf-like distribution, and analyze the effectiveness of media caching in detail. Fan et al. [36] propose the adaptive allocation cooperative caching (AACC) algorithm, which conducts cache partitioning and allocation according to the data access pattern to minimize total data access cost for cooperative caching in IMANETs. Samarasinghe et al. [38] predict the cache update interval on the basis of the user access pattern to improve access performance and the freshness of cached data. Shafiq et al. [39] and Sundarrajan et al. [39, 40] analyze the characteristics of CDN workloads and propose caching strategies such as N-hit caching and content-aware caching.

7 CONCLUSION

When SSD is deployed as a caching layer, write density will be high and its lifetime is threatened. Through the analysis of real-world photo cache traces, we find that it is possible to cut down on SSD writes in the first place. Take Tencent's QQ photo album, for instance: users' daily browsing is up to 50 billion, of which there is a large number of one-time-access photos. In the regular cache configuration, these photos will be swapped into the cache, which introduces a large number of unnecessary SSD writes and reduces cache utilization.

Motivated by this observation, we endeavored to predict future photo accesses and prevent one-time-access photos from entering the cache. Owning to the lack of history information, it is hard to make such predictions accurately using traditional cache algorithms. Therefore, we propose to leverage machine learning approaches to extract the most related features dynamically and optimize the training model. In our approach, we obtain a prediction accuracy of greater than 80% and achieve various extents of improvement in the hit rate, SSD writes, and access latency. Although our research aims at Tencent's photo caching, it is also applicable to other buffer caching in cloud storage environments.

REFERENCES

[1] Ethem Alpaydin. 2014. *Introduction to Machine Learning*. MIT Press, Cambridge, MA.

[2] Xiao Bai, B. Barla Cambazoglu, and Archie Russell. 2016. Improved caching techniques for large-scale image hosting services. In *Proceedings of the 39th International ACM SIGIR Conference on Research and Development in Information Retrieval.* ACM, New York, NY, 639–648.

[3] Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. 1984. *Classification and Regression Trees*. Wadsworth.

[4] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. 1999. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of the 18th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'99)*, Vol. 1. IEEE, Los Alamitos, CA, 126–134.

[5] Li-Pin Chang, Yu-Syun Liu, and Wen-Huei Lin. 2016. Stable greedy: Adaptive garbage collection for durable page-mapping multichannel SSDs. *ACM Transactions on Embedded Computing Systems* 15, 1 (Jan. 2016), Article 13, 25 pages.

[6] Feng Chen, Tian Luo, and Xiaodong Zhang. 2011. CAFTL: A content-aware flash translation layer enhancing the lifespan of flash memory based solid state drives. In *Proceedings of the 9th USENIX Conference on File and Storage Technologies (FAST'11)*, Vol. 11. 77–90.

[7] F. H. Chen, M. C. Yang, Y. H. Chang, and T. W. Kuo. 2015. PWL: A progressive wear leveling to minimize data migration overheads for NAND flash devices. In *Proceedings of the 2015 Design, Automation, and Test in Europe Conference and Exhibition (DATE'15).* 1209–1212.

[8] Asaf Cidon, Assaf Eisenman, Mohammad Alizadeh, and Sachin Katti. 2015. Dynacache: Dynamic cloud caching. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'15).*

[9] Riley Crane and Didier Sornette. 2008. Robust dynamic classes revealed by measuring the response function of a social system. *Proceedings of the National Academy of Sciences* 105, 41 (2008), 15649–15653.

[10] Zhaoxia Deng, Lunkai Zhang, Nikita Mishra, Henry Hoffmann, and Frederic T. Chong. 2017. Memory cocktail therapy: A general learning-based framework to optimize dynamic tradeoffs in NVMs. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, New York, NY, 232–244.

[11] Assaf Eisenman, Asaf Cidon, Evgenya Pergament, Or Haimovich, Ryan Stutsman, Mohammad Alizadeh, and Sachin Katti. 2019. Flashield: A hybrid key-value cache that controls flash write amplification. In *Proceedings of the 16th USENIX Conference on Networked Systems Design and Implementation (NSDI'19).* 65–78.

[12] Charles Elkan. 2001. The foundations of cost-sensitive learning. In *Proceedings of the International Joint Conference on Artificial Intelligence*, Vol. 17. 973–978.

[13] Eran Gal and Sivan Toledo. 2005. Algorithms and data structures for flash memories. *ACM Computing Surveys* 37, 2 (2005), 138–163.

[14] Francesco Gelli, Tiberio Uricchio, Marco Bertini, Alberto Del Bimbo, and Shih-Fu Chang. 2015. Image popularity prediction in social media using sentiment and context features. In *Proceedings of the 23rd ACM International Conference on Multimedia.* ACM, New York, NY, 907–910.

[15] Ping Huang, Wenjie Liu, Kun Tang, Xubin He, and Ke Zhou. 2016. ROP: Alleviating refresh overheads via reviving the memory system in frozen cycles. In *Proceedings of the 2016 45th International Conference on Parallel Processing (ICPP'16).* IEEE, Los Alamitos, CA, 169–178.

[16] Ping Huang, Pradeep Subedi, Xubin He, Shuang He, and Ke Zhou. 2014. FlexECC: Partially relaxing ECC of MLC SSD for better cache performance. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14).* 489–500.

[17] Qi Huang, Ken Birman, Robbert van Renesse, Wyatt Lloyd, Sanjeev Kumar, and Harry C. Li. 2013. An analysis of Facebook photo caching. In *Proceedings of the 24th ACM Symposium on Operating Systems Principles.* ACM, New York, NY, 167–181.

[18] Sai Huang, Qingsong Wei, Dan Feng, Jianxi Chen, and Cheng Chen. 2016. Improving flash-based disk cache with lazy adaptive replacement. *ACM Transactions on Storage* 12, 2 (2016), 8.

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

[20] Daniel A. Jiménez and Elvira Teran. 2017. Multiperspective reuse prediction. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture.* ACM, New York, NY, 436–448.

[21] Xavier Jimenez, David Novo, and Paolo Ienne. 2014. Wear unleveling: Improving NAND flash lifetime by balancing page endurance. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*, Vol. 14. 47–59.

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

[23] Georgios Keramidas, Pavlos Petoumenos, and Stefanos Kaxiras. 2007. Cache replacement based on reuse-distance prediction. In *Proceedings of the 2007 25th International Conference on Computer Design.* IEEE, Los Alamitos, CA, 245–250.

[24] Mazen Kharbutli and Yan Solihin. 2008. Counter-based cache replacement and bypassing algorithms. *IEEE Transactions on Computers* 57, 4 (2008), 433–447.

[25] Aditya Khosla, Atish Das Sarma, and Raffay Hamid. 2014. What makes an image popular? In *Proceedings of the 23rd International Conference on World Wide Web.* ACM, New York, NY, 867–876.

[26] Ren-Shuo Liu, Chia-Lin Yang, and Wei Wu. 2012. Optimizing NAND flash-based SSDs via retention relaxation. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12).* 11.

[27] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03).* 115–130.

[28] Leeor Peled, Shie Mannor, Uri Weiser, and Yoav Etsion. 2015. Semantic locality and context-based prefetching using reinforcement learning. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA'15).* ACM, New York, NY, 285–297.

[29] Moinuddin K. Qureshi, Vijayalakshmi Srinivasan, and Jude A. Rivers. 2009. Scalable high performance main memory system using phase-change memory technology. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09).* ACM, New York, NY, 24–33.

[30] Gokul Soundararajan, Vijayan Prabhakaran, Mahesh Balakrishnan, and Ted Wobber. 2010. Extending SSD lifetimes with disk-based write caches. In *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST'10)*. 101–114.

[31] Gabor Szabo and Bernardo A. Huberman. 2010. Predicting the popularity of online content. *Communications of the ACM* 53, 8 (2010), 80–88.

[32] Linpeng Tang, Qi Huang, Wyatt Lloyd, Sanjeev Kumar, and Kai Li. 2015. RIPQ: Advanced photo caching on flash for Facebook. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. 373–386.

[33] Linpeng Tang, Qi Huang, Amit Puntambekar, Ymir Vigfusson, Wyatt Lloyd, and Kai Li. 2017. Popularity prediction of Facebook videos for higher quality streaming. In *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*. 111–123.

[34] Elvira Teran, Zhe Wang, and Daniel A. Jiménez. 2016. Perceptron learning for reuse prediction. In *Proceedings of the 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*. IEEE, Los Alamitos, CA, 1–12.

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

[36] Xiaopeng Fan, Jiannong Cao, Haixia Mao, Weigang Wu, Yubin Zhao, and Chengzhong Xu. 2016. Web access patterns enhancing data access performance of cooperative caching in IMANETs. In *Proceedings of the 2016 17th IEEE International Conference on Mobile Data Management (MDM'16)*, Vol. 1. IEEE, Los Alamitos, CA, 50–59.

[37] Lei Guo, Enhua Tan, Songqing Chen, Zhen Xiao, and Xiaodong Zhang. 2008. The stretched exponential distribution of Internet media access patterns. In *Proceedings of the 27th ACM Symposium on Principles of Distributed Computing*. ACM, Los Alamitos, CA, 283–294.

[38] Rohan Samarasinghe, Yoshihiro Yasutake, and Takaichi Yoshida. 2005. Optimizing the access performance and data freshness of distributed cache objects considering user access pattern. In *Proceedings of the 19th International Conference on Advanced Information Networking and Applications (AINA'05)*, Vol. 2. IEEE, Los Alamitos, CA, 325–328.

[39] M. Zubair Shafiq, Amir R. Khakpour, and Alex X. Liu. 2016. Characterizing caching workload of a large commercial content delivery network. In *Proceedings of the 35th Annual IEEE International Conference on Computer Communications (INFOCOM'16)*. IEEE, Los Alamitos, CA, 1–9.

[40] Aditya Sundarrajan, Mingdong Feng, Mangesh Kasbekar, and Ramesh K. Sitaraman. 2017. Footprint descriptors: Theory and practice of cache provisioning in a global CDN. In *Proceedings of the 13th International Conference on Emerging Networking Experiments and Technologies*. ACM, New York, NY, 55–67.

[41] Yue Yang and Jianwen Zhu. 2016. Write skew and Zipf distribution: Evidence and implications. *ACM Transactions on Storage* 12, 4 (2016), 1–19.

[42] Guanying Wu and Xubin He. 2012. Delta-FTL: Improving SSD lifetime via exploiting content locality. In *Proceedings of the 7th ACM European Conference on Computer Systems*. ACM, New York, NY, 253–266.

[43] Qiang Yang, Haining Henry Zhang, and Tianyi Li. 2001. Mining web logs for prediction models in WWW caching and prefetching. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 473–478.

[44] Rui Ye, Wentao Meng, and Shenggang Wan. 2017. Extending lifetime of SSD in Raid5 systems through a reliable hierarchical cache. In *Proceedings of the 2017 International Conference on Networking, Architecture, and Storage (NAS'17)*. IEEE, Los Alamitos, CA, 1–8.

[45] Qingyuan Zhao, Murat A. Erdogdu, Hera Y. He, Anand Rajaraman, and Jure Leskovec. 2015. Seismic: A self-exciting point process model for predicting tweet popularity. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, New York, NY, 1513–1522.

[46] Ke Zhou, Shaofu Hu, Ping Huang, and Yuhong Zhao. 2017. LX-SSD: Enhancing the lifespan of NAND flash-based memory via recycling invalid pages. In *Proceedings of the 2017 IEEE 33rd Symposium on Massive Storage Systems and Technology*.

[47] Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, and Tianming Yang. 2019. Improving cache performance for large-scale photo stores via heuristic prefetching scheme. *IEEE Transactions on Parallel and Distributed Systems* 30, 9 (2019), 2033–2045.

[48] Ke Zhou, Yu Zhang, Ping Huang, Hua Wang, Yongguang Ji, Bin Cheng, and Ying Liu. 2018. LEA: A lazy eviction algorithm for SSD cache in cloud block storage. In *Proceedings of the 2018 IEEE 36th International Conference on Computer Design (ICCD'18)*. IEEE, Los Alamitos, CA, 569–572.

Received June 2019; revised February 2020; accepted April 2020