This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

Improving Cache Performance for Large-Scale Photo Stores via Heuristic Prefetching Scheme

Ke Zhou, Si Sun, Hua Wang, Ping Huang, Xubin He, Rui Lan, Wenyan Li, Wenjie Liu, Tianming Yang

Abstract—Photo service providers are facing critical challenges of dealing with the huge amount of photo storage, typically in a magnitude of billions of photos, while ensuring national-wide or world-wide satisfactory user experiences. Distributed photo caching architecture is widely deployed to meet high performance expectations, where efficient still mysterious caching policies play essential roles. In this work, we present a comprehensive study on internet-scale photo caching algorithms in the case of QQPhoto from Tencent Inc., the largest social network service company in China. We unveil that even advanced cache algorithms can only perform at a similar level as simple baseline algorithms and there still exists a large performance gap between these cache algorithms and the theoretically optimal algorithm due to the complicated access behaviors in such a large multi-tenant environment. We then expound the reasons behind this phenomenon via extensively investigating the characteristics of QQPhoto workloads. Finally, in order to realistically further improve QQPhoto cache efficiency, we propose to incorporate a prefetcher in the cache stack based on the observed immediacy feature that is unique to the QQPhoto workload. The prefetcher proactively prefetches selected photos into cache before they are requested for the first time to eliminate compulsory misses and promote hit ratios. Our extensive evaluation results show that with appropriate prefetching we improve the cache hit ratio by up to 7.4%, while reducing the average access latency by 6.9% at a marginal cost of 4.14% backend network traffic compared to the original system that performs no prefetching.

Index Terms—Caching Algorithm; Distributed Storage; Photo Storage; Cloud Computing;

1 Introduction

PHOTO sharing is a most common social network activity through which people communicates their daily life updates to friends and even strangers over the internet, where uploaded photos are typically viewed and commented more intensively right after their publishings and accesses to them gradually fade away as time goes. It has recently become a dominating web content generator [1], resulting in billions of photos being hosted by the photo service provider. Hundreds of millions of users are interacting with the photo store in a concurrent manner, which imposes significant management challenges to the photo store [2], [3], [4]. To efficiently support such a large-scale photo store and deliver satisfactory user experiences, photo service providers routinely build geographically distributed and hierarchically structured photo storage systems [4], [5], [6], [7], which consist of multiple layers along the access path, including client-end cache, edge cache, regional cache [2] and backend storage located in a number of data centers.

The deployment of multiple cache layers not only speeds up photo-access requests, but also reduces downstream traffics to the low-performance backend storage when the requested photos happen to have already been cached in the caches. For example, a 8.5% hit ratio improvement can reduce 20.8% downstream requests [2]. Therefore, as in traditional on-chip cache scenarios [8], [9], [10], it is important to improve the photo cache hit ratio. In fact, it is even more desirable to achieve a high hit ratio in the web photo cache case, since the miss penalty is more expensive as a missed access request might be routed through the network to locate the requested photo. Unfortunately, it is particularly challenging to design effective cache algorithms for photo caching workloads, as the access patterns are rather complicated and extremely difficult to predict due to the nature of multi-tenancy and extensive-sharing in the cloud web environment [11], [12], [13], [14], [15], [16]. Access requests from different clients are constantly intervened such that recency and frequency are often compromised, leading to relatively low hit ratios (compared to the Clairvoyant algorithm) of locality-based cache algorithms. Even though the Facebook's analysis work suggests that advanced cache algorithms are able to perform better [2], still simple algorithms are utilized in real-world deployed systems. For instance, Facebook's photo caching uses the FIFO algorithm and Tencent's QQPhoto employs the LRU algorithm. Particularly, for such photo caching workloads, it has been demonstrated that even advanced cache algorithms are able to improve the hit ratios only when the cache capacity is smaller than a certain size and there exist inflect points on the hit ratio curves beyond which advanced cache algorithms lose their advantageous benefits [2].

In order to further improve the cache efficiency for photo cache workloads, we comprehensively investigate the characteristics of photo cache workloads by analyzing a vast amount of realistic photo cache traces of the QQPhoto workload at Tencent Inc. [17], the largest social network

- *K. Zhou, S. Sun and H. Wang are with Wuhan National Laboratory for Optoelectronics (Huazhong University of Science and Technology), Key Laboratory of Information Storage System, Intelligent Cloud Storage Joint Research center of HUST and Tencent.*
- *P. Huang, X. He and W. Liu are with Temple University. P. Huang is the corresponding author. E-mail: templestorager@temple.edu*
- *R. Lan and W. Li are with Tencent Inc.*
- *T. Yang is with Huanghuai University.*

A preliminary version appeared in the proceedings of the 32nd ACM International Conference on Supercomputing (ICS), 2018.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

service provider in China. According to our analysis, we find that the major factor limiting hit ratio improvement is the compulsory misses or cold misses of the first accesses to photos, with which existing cache algorithms are incapable of dealing. Based on this observation, we identify that the key to improving hit ratio is to eliminate those cold misses as much as possible. Moreover, we have observed that a majority of photos become extensively interesting to users in a limited period of the time right after their uploadings, which we call the *immediacy* feature. Leveraging this feature, we propose to augment a prefetcher to the photo cache to reduce compulsory misses. The prefetcher proactively prefetches selected resolutions (Section 2) of freshly uploaded photos from the backend storage system to the photo cache in advance in anticipation that they will be accessed soon. It should be noted that the added prefetecher is tailored for the QQPhoto cache infrastructure in which client photos are directly uploaded to the backend storage system and photo read requests are serviced in a separate read channel via caching. In alternative infrastructures where photos are written to the photo cache layer when uploaded, photos can be deemed as prefetched upon uploading. However, in doing so, those infrastructures require excessively large amount of cache space, which could be too expensive to afford. In other words, in that infrastructure, we could utilize the *immediacy* feature to decide which photos are to be kept in the cache. Our evaluations show that the prefetcher is able to promote photo cache hit ratio at a marginal expense of network resources consumption.

In summary, we make the following main research contributions in this paper:

(1) We conduct a comprehensive investigation on a set of realistic large-scale photo cache traces and make several interesting and insightful observations.

(2) We perform extensive experiments with various cache algorithms using the photo cache traces and find that even advanced cache algorithms are only able to bring negligible benefits in the photo cache scenario due to excessive compulsory misses.

(3) We propose to augment a prefetcher to proactively bring selected resolutions of recently uploaded photos into the cache stack. To make the prefetcher more efficient, we also design two heuristics based cache algorithms.

(4) We have implemented a cache simulator framework to verify our designs. We evaluated the prefetcher's efficiency with the simulator thoroughly. Compared with the state-of-art cache policies, our prefetcher improves the hit ratios by up to 7.4%, while reducing the average access latency by 6.9% at a marginal cost of 4.14% backend network traffic.

The remainder of this paper is structured as follows. In Section 2, we present the background of this study with a focus on the QQPhoto architecture and trace methodology and our motivation to improve photo cache hit ratios. We then elaborate on the design details of our proposed photo cache prefetcher in Section 3. We experimentally evaluate the efficacy of our solution in terms of hit ratio improvement and request latency reduction in Section 4. Finally, we discuss related work on cache algorithms and photo storage in Section 5 and conclude the paper in Section 6.

2 Background and Motivation

2.1 QQPhoto Preliminaries

QQPhoto at Tencent Inc. [17] is the photo service factory that supports the popular instant message QQ application in China. QQ users upload photos to their respective photo albums. Depending on the chosen visibility protection scheme (including private, public, and friend-only), only friends of a photo album owner or all strangers on the internet can browse photos in the album. In 2017, QQPhoto hosted more than 2,000 billions of photos for 1 billion users, which amounted to a total storage capacity of 300PB, with 250 millions of photos uploaded and 50 billions of photos requested daily [18]. Nowadays, the magnitude of QQPhoto is much larger. It has been a tremendous challenge for Tencent to manage the photo store system at such a large scale.

In QQPhoto, photo writes and reads are routed through separate upload and download channels and the QQPhoto cache stack specifically refers to the download cache path. Figure 1 gives an overview of the QQPhoto cache stack. Figure 1 (a) depicts the upload and download channels. As it is indicated, the photo infrastructure comprises the backend storage where all photos are hosted and a cache layer consisting of SSDs. Photos are directly uploaded to the backend storage, while they are provided to users through the cache layer comprised of multiple Data Center Caches (DCs) and Outside Caches (OCs). Tencent adopts separate paths for reads and writes mainly based on the following two considerations. First, after uploading, each photo is resized to multiple versions corresponding to specified different specifications and formats, which is referred as a *physical* or *resolution* photo. The original photo is called a *logical* photo. Storing multiple physical photos for a logical photo is a common practice among web media content service providers to accommodate clients' varying display requirements, e.g., desktop setting or mobile terminal [5], [19]. Directly uploading photos to the backend storage relieves resizing computational burden from the cache servers. Second, uploading photos to and resizing them in the cache servers would pollute and quickly consume up the cache space, jeopardizing overall cache performance. In contrast, QQPhoto employs an SSD cache layer to fasten photo read requests and conserve network requirements, because reads are more popular than writes in photo caching workloads. The total SSD cache space in one data center cache is about 5TB and is managed using the Least Recently Used (LRU) cache policy.

A photo download request goes through both OC and DC. When a user requests photos, QQPhoto first sends the query to the OC which is geographically nearest to the requesting user or has the smallest network distance. If the requested photos are cached in the selected OC, they are returned to the user from the OC. Otherwise, the photo request is forwarded to a DC for further checking. If it misses in the DC again, the request continues to be passed on to the backend storage and the requested photos are populated to the DC and OC cache. Figure 1 (b) describes

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

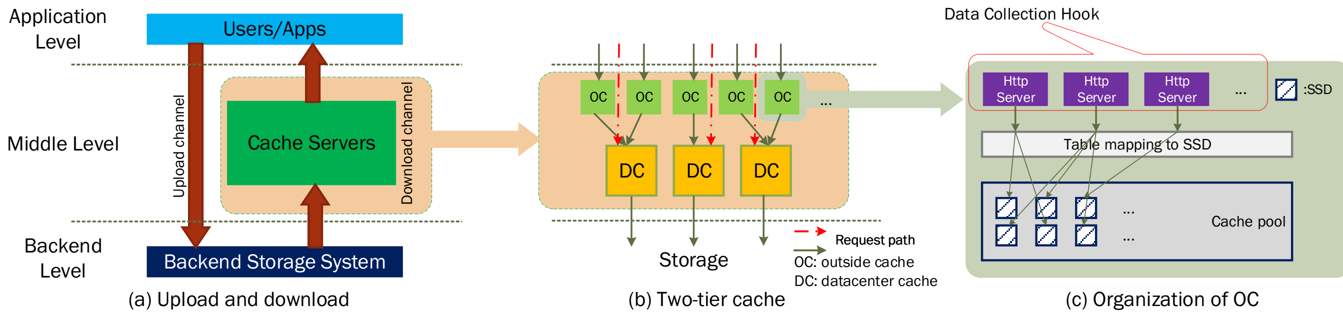


Fig. 1: The QQPhoto architectural view. Figure (a) shows that QQPhoto supports separate photo upload and download channels. Figure (b) shows that the photo download cache path includes outside caches (OCs) and data center caches (DCs). Figure (c) gives a detailed view of the internal organization of an OC.

this query procedure. In addition, QQPhoto also optimizes photo accesses using a heuristic based on the creation time of requested photos. If the requested photos were created a relatively long time ago, e.g., a week ago, QQPhoto simultaneously issues requests to the DC to reduce access latency because they may more likely not be cached in the OC, as it is indicated by the red arrow requests in Figure 1(b). Please be noted that QQPhoto allows multiple copies of a photo to be cached in different OCs, as OCs make cache decisions on behalf of their respective clients independently of other OCs. Figure 1 (c) gives a detailed internal structural view of an OC. Within an OC, there are hundreds of peer web servers responsible for handling photo requests arriving at the OC. Our photo traces were crawled from requests in such an OC.

2.2 Photo Traces and Sampling Method

The QQPhoto traces used in our study span a 9-day length period of duration and record photo requests occurring to an OC during that period of time. Each request log entry contains the following information: request timestamp, photo ID, image format (jpg, webp, etc.), specification (small, medium or large), handling time, return size, terminal type (PC or mobile) and some other auxiliary information which is irrelevant and thus ignored in our study. All physical photos corresponding to the same logical photo have the same photo ID. The whole trace contains a total of about 5.8 billion requests.

To efficiently experiment with such a large set of traces while without missing their original behavior characteristics, we select a representative subset of samples from the original traces for evaluations. Specifically, we first extract all unique photo IDs into an ID set and then use the reservoir sampling method [20] to sample out $\frac{1}{100}$ of the total photo IDs. We then obtain our experimental traces via extracting from the original traces the requests whose photo IDs belonging to the sampled set of photo IDs. In doing so, our sampled set of traces inherits the access characteristics in the original traces since it keeps the whole access history of any sampled logical photos. For instance, assume the original traces are $< P_6^2, P_2^1, P_4^2, P_1^1, P_6^1, P_1^2, P_5^2, P_2^3, P_3^1, P_4^1, P_5^1, P_2^2, P_3^2 >$, where P_i^j, represents the physical photo corresponding to the j^{th} resolution of the i^{th} (Photo ID) logical photo. If

TABLE 1: QQPhoto Original and Sampled Traces

	Original	Sampled
# of Requests	5,854,956,972	58,565,016
# of Logical Photos	801,498,523	8,014,985
# of Physical Photos	1,515,462,898	15,162,925
Data set (GB)	46,753	467
Total Traffic (GB)	186,712	1,802

we sample one third of photos IDs and the sampled photo ID set is $< P_1, P_2 >$, then our extracted traces will be $< P_2^1, P_1^1, P_1^2, P_2^3, P_2^2 >$, which reserves the whole access history of the photos $< P_1, P_2 >$. Table 1 gives a brief comparison of some key statistics between the original and sampled traces.

To verify the faithfulness of our sampling method, we evaluate both original and sampled traces with the LRU cache algorithm used by QQPhoto. For the original traces, we set the total cache size the same as in the real production system, denoted as size X (about 5TB). Correspondingly, we use one percent of the original cache size (i.e., 0.01X) for the sampled traces. We compare the photo hit ratio and byte hit ratio between the original and sampled traces. Table 2 lists the comparison results. The photo hit ratios of the original and sampled traces are 67.9% and 67.7% and the byte hit ratios are 69.6% and 68%, with a bias of -0.2% and -1.6%, respectively. To verify the accuracy of simulation results, we have also obtained the overall hit ratio of the entire original trace, which is calculated as the ratio of the number of photo hits to the total number of photo accesses. The trace hit ratio is 65.62%, which is quite near to our simulation result. The slight difference might be due to not all the cache space (5T) in the production system being used to cache photo files. Some cache space is used for management overhead, while all the cache is simulated to cache photo files in the simulation. As we are more concerned about the photo hit ratio, we believe that our sampled traces faithfully represent the original traces. Our following evaluations are all based on the sampled traces.

2.3 Advanced Algorithms

A straightforward approach to improving hit ratio is to employ advanced cache algorithms. In fact, the Facebook's photo analysis work has concluded that using advanced

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

TABLE 2: Hit Ratio Comparison

	Hit ratio	Byte hit ratio
Original Traces	67.9%	69.6%
Sampled Traces	67.7%	68.0%

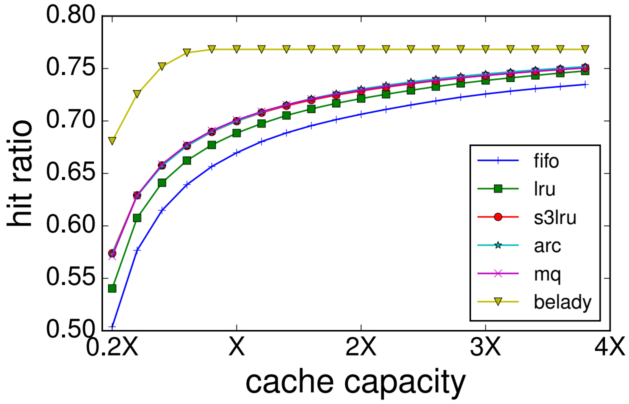


Fig. 2: The hit ratio curves of various cache algorithms on the QQPhoto cache workloads. The improvements of advanced cache algorithms over LRU are negligible and there is a big gap between the Belady algorithm and the advanced cache algorithms.

cache algorithms improves the hit ratios of Facebook's photo caching workloads [2], [4]. In this section, we apply advanced cache algorithms to QQPhoto workloads to examine their efficacies and investigate potential improvement space for the QQPhoto workloads beyond cache algorithms.

The fundamental rationale to improve hit ratios is to accurately cache the items that will be most likely used in the near future, which requires perfect knowledge of future access behaviors. Unfortunately, in most cases, it is almost always impossible to obtain the ideal insights into future access patterns. Therefore, existing cache algorithms make cache decisions based on recent historical patterns. Several commonly used heuristics include recency, frequency, and reuse distance. More specifically, the least recently accessed items will be less likely accessed in the near future (LRU), the less frequently accessed items will be less likely accessed (LFU), and cache items with smaller reuse distance are more likely to be reused again. Advanced cache algorithms may make cache decisions by taking into account multiple of those heuristics.

We implement three advanced cache algorithms, i.e., S3LRU, ARC, MQ [21], [22], and use them to experiment with the photo cache traces. In addition, we experiment the LRU algorithm which is employed in the QQPhoto production system and the FIFO algorithm which is used in Facebook's photo cache. Finally, we also experiment the optimal offline Belady's MIN algorithm for the best possible up-bound hit ratios. We vary the cache capacity in the range of $(0.2X, X, 2X, 3X, 4X)$. Figure 2 shows the hit ratios of the cache algorithms. We can make four interesting observations from this figure. First, for any cache algorithm, there exists an inflection point on the hit ratio curve. The effect of cache increase below the inflection point is apparent, while

TABLE 3: Hit Ratio Contribution Breakdown

Frequency	PoP	PoR	CtoHR
$f > 10000$	0.0006%	4.9926%	4.9925%
$1000 < f \leq 10000$	0.0081%	4.1116%	4.1096%
$100 < f \leq 1000$	0.1567%	9.4964%	9.4568%
$10 < f \leq 100$	5.9010%	36.2041%	34.7113%
$5 < f \leq 10$	5.6052%	10.6784%	9.2604%
$2 < f \leq 5$	12.5679%	11.7320%	8.5525%
$f = 2$	14.3027%	7.2368%	3.6184%
$f = 1$	61.4577%	15.5480%	0

Note: PoP, PoR, and CtoHR stand for the "percentage of photos", "percentage of requests", and "contribution to hit ratio", respectively.

it is rather limited once crossing the inflection point. Second, FIFO performs the worst, leading to other algorithms showing good improvement space. The fact that LRU performs better than FIFO agrees with the intuition of social network behaviors, i.e., recently uploaded photos are more likely to be visited. Third, there is still a big gap between advanced cache algorithms and the optimal curve. For example, at capacity X which is the realistic case, the hit ratio of LRU is 67.7% and the hit ratio of optimal Belady is 76.8%, resulting in a 9% difference. These three observations are in line with the findings from Facebook's photo cache study [2], [4]. Fourth, compared to the LRU algorithm, other advanced cache algorithms show very limited improvements (e.g., S3LRU improves only 1.24% at capacity X) and all perform at a comparable level as indicated by the overlapped curves.

To examine the behind reasons why advanced cache algorithms are not able to deliver better cache performance and find out any improvement opportunity, we investigate the frequency and reuse distance distributions. Both logical photos and physical photos exhibit zipf-like distribution. Table 3 gives more details about the relationship between different frequency photos and their respective contributions to the hit ratio. *PoP* is the ratio of the number of photos in each frequency group to the total number of photos and *PoR* is the ratio of the amount of requests in each group to the total requests. *CtoHR* is calculated via dividing the difference between the number of requests and the number of photos in each group by the total requests in the trace. The *CtoHR* assumes all requests are hits except for the first photo request and it reflects how much hit ratio the group contributes. As can be inferred from Table 3, highly frequent ($frequency$ >100) photos occupy a very minimal percentage of the total photos (1.6%) but make contributions to hit ratio commensurate with their occurrences in the requests. Medium frequent ($2 < frequency \leq 100$) photos account for a medium percentage (24%) but also contribute commensurately with their occurrences in the requests. However, the least-frequent (frequency is 1 or 2) photos account for 75.7% of the total photos and 22.8% of requests, but contribute negligibly to the hit ratio. Especially, 15.5% of the requests to photos of 1 frequency are all missed. Therefore, frequency-based cache algorithms are incapable of capturing those 22.8% requests as their requested photos' lower frequencies would cause them to be quickly evicted from the cache.

Figure 3 shows the cache reuse distance cumulative distribution functions grouped by photo frequency ranges. We define the *"reuse distance"* as the real time difference

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

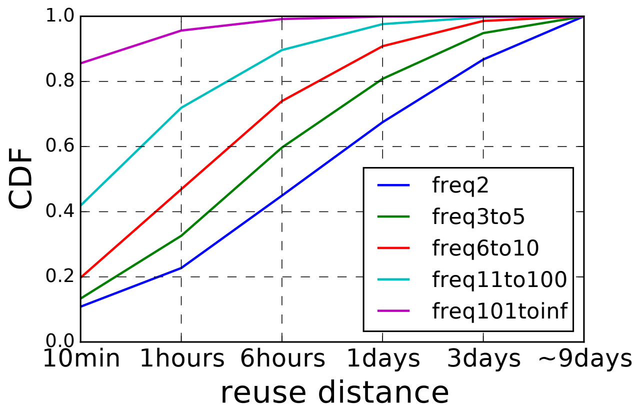


Fig. 3: The CDFs of photo reuse distance grouped by photo frequency. Higher frequency photos show smaller reuse distance and lower frequency have larger reuse distance, resulting in recency-based cache algorithms ineffective for lower frequency photos.

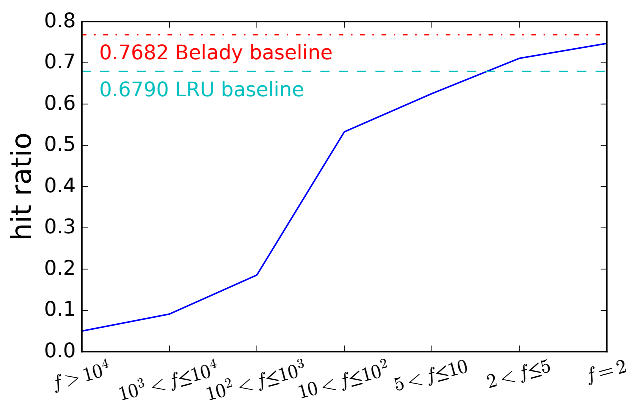


Fig. 4: The cumulative CtoHR of all groups with $freq > 2$.

between two consecutive accesses to a cached photo and thus it also includes the feature of *"recency"*. Note that there is no "freq1" curve. As shown in the figure, higher frequency photos exhibit smaller reuse distance, while lower frequency photos have larger reuse distance, which means recency-based cache algorithms are not able to capture low-frequency photos either.

2.4 Motivation

As discussed in preceding sections, we have analyzed the factors limiting advanced cache algorithms from promoting hit ratio for the photo cache workloads. Existing cache algorithms leverage either frequency or recency, or a combination of frequency and recency. However, as we have seen, the low-frequency photos are neither frequency-friendly nor recency-friendly. Still, they account for a decent amount of the total requests.

Calculating the cumulative CtoHR (Figure 4), it is apparent that to further improve the hit ratio over LRU, we need a mechanism that is able to improve the hit ratio of those low-frequency photos. Given the 1-frequency photos

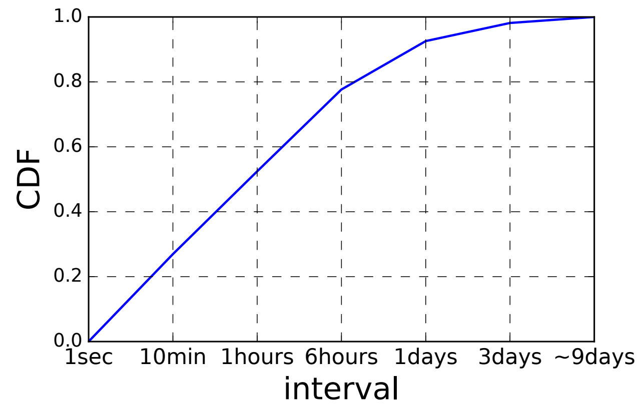


Fig. 5: The CDF of time interval between photos uploading time and their first request time.

are all compulsory misses and the 2-frequency photos have larger reuse distance (thus the second access is also likely a compulsory miss due to being evicted), we are motivated to employ prefetching to eliminate compulsory misses and improve overall hit ratio.

We leverage a common social network phenomenon to guide our prefetching design, i.e., recently uploaded photos are more likely to attract internet users' interest and attention, which we name as "immediacy". To put the phenomenon in perspective, we have analyzed the time intervals between photos uploading time and their first request time. Figure 5 shows the cumulative distribution function of the first request intervals. As can be seen from the figure, 30% photos are visited for the first time 10 minutes after their uploading and this number becomes 52% 6 hours after their uploading. Moreover, 90% photos are accessed within 1 day following their uploading [1]. Therefore, if we prefetch recently uploaded photos in the cache and keep it for at least 24 hours, we can eliminate compulsory misses to them at a probability of 0.9. Fortunately, QQPhoto's cache space is large enough to host one day's worth of uploaded photos, which makes our approach practically feasible.

3 Photo Cache Prefetching

As discussed before, advanced cache algorithms fail to capture the portion of low-frequency photos due to their peculiar access patterns of photo cache, leading to a large amount of compulsory misses. We propose to add a prefetcher to maximally eliminate compulsory misses and improve cache hit ratio via prefetching appropriate photos to the cache from the backend storage. We leverage photo cache characteristic to guide which photos to prefetch and when to perform prefetching. In this section, we elaborate on the prefetcher design, focusing on analyzing the popularities of photo resolutions (i.e., the physically stored versions of a photo) and discussing prefetching scheduling alternatives.

1. At first glance, it may sound strange to define "immediacy" in hours or a day. However, we assume it is reasonable in the social network context, particularly when considering time zone differences among the world-wide clients.

3.1 Prefetch Photo Resolutions

Clients of QQPhoto upload millions of photos daily and QQPhoto resizes each photo to multiple resolutions, i.e., combinations of specification and format, to be stored in the backend storage. To effectively prefetch photo candidates from such a vast amount of photos, we need a good heuristic to help guide our prefetch design. Leveraging characteristics of social network workloads and considering the separate upload and download path of QQPhoto architecture, we prefetch recently uploaded photos from the backend storage to the cache pool.

We observe that resolutions of the same photo show different access popularities. Some resolutions are much more intensively accessed, while others experience very few accesses. For example, with the largest majority of photo clients using mobile terminals to navigate QQ albums and perform related activities (like make comments on photos), the resolutions suitable for mobile settings are thus accessed more intensively. To reveal this phenomenon in the QQPhoto caching, we have performed an analysis on the access popularity of photo resolutions. To ensure information anonymity, we denote the photo resolutions as *"Rez1"*, *"Rez2"* and so on according to their popularity rank, i.e., *"Rez1"* is the most popular resolution, and *"Rez2"* is the next to *"Rez1"*. Figure 6 shows the results. As it is shown, the most popular resolution *"Rez1"* accounts for 34.78% and the next three resolutions stay around in the range of 13%-15%, with the remaining being less than 10%. This figure implies that prefetching the most popular photo resolutions can deliver reasonably good cache hit ratios. The more resolutions we prefetch, the better chances of eliminating compulsory misses. However, the probability of network wastage is also higher. Therefore, the choice of the number of prefetching resolutions (NPR) presents a trade-off point between cache efficiency and network overhead. We evaluate how various values of this parameter affect the trade-offs in Section 4. Algorithms 1 describes the procedure of the prefetching method. Line 1-7 describes the normal online caching service program. When a prefetching requirement is determined, our program spawns a new thread PREFETCHER to prefetch photos. The thread is scheduled to run in the background to minimize its impacts on the online photo service. PREFETCHER (Line24-26) prefetches logical photos in sequence according to their uploaded timestamp. The resolutions of a logical photo are prefetched in the order of their popularities until NPR resolutions are prefetched.

To prevent workloads variations from undermining the efficacy of prefetcher, the prefetcher can employ an on-line profiler which dynamically and periodically profiles workloads and outputs the popularity rank of photo resolutions. This information is used to determine photo resolutions for the next prefetching.

3.2 Prefetching Scheduling

Since QQPhoto is a 24×7 online service, meaning clients are continuously uploading and downloading photos. Therefore, we periodically perform prefetching to ensure that uploaded photos get chances to be prefetched and immediate accesses to them can be serviced from the cache. At every prefetch timepoint, we prefetch the most popular

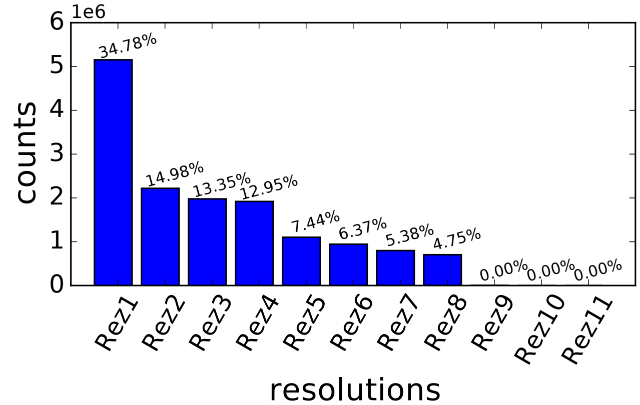


Fig. 6: Photo resolution distributions. The x-axis represents the photo resolutions ordered by their popularity rank. The y-axis denotes the number of respective resolutions. The numbers on top of the bars give their percentages. Prefetching only several highly ranked resolutions can provide a good prefetching coverage.

resolutions of all photos uploaded during the last period. For instance, assume we configure to prefetch 2 resolutions (based on the profiling results) and $[T_1, T_2]$ is a prefetch interval. Then, at time T_2, we prefetch 2 resolutions of all photos which were uploaded to the backend storage during $[T_1, T_2]$(Line3-5).

3.3 Smart Eviction

Another decision pertaining to prefetched photos is use what eviction policy to manage the cache pool containing both prefetched photos and cached photos. A preliminary idea is to treat prefetched and cached photos uniformly, i.e., once prefetched photos have been entered in the cache pool, they are managed using the LRU cache algorithm along with regular cached photos. However, taking into account the characteristics of workloads, we could benefit from two heuristic improvements on eviction policy. The first improvement is *time heuristic* that leverages the features of *immediacy* and *low-frequency* and instructs the prefetcher to proactively evict photos the two features of which have faded away due to time elapse. The second improvement considers different resolution popularities of the same logical photo and checks whether to evict other resolutions of the same photo when a resolution is being evicted, which is *logical photo heuristic*. We called these heuristic modules when applied to the eviction policy as *"smart eviction"*.

3.3.1 Time Heuristic

Using *time heuristic*, prefetched photos are timestamped to be distinct from cached photos and they are also put in the LRU queue along with cached photos. The heuristic tries to leverage the *immediacy* access characteristic as shown in Figure 5. When making an eviction decision, prefetched photos that have been in the cache longer than a configured length period of time are evicted because they lose *immediacy*. On the other hand, prefetched photos whose frequencies are less than a specified frequency will be evicted preferentially

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

Algorithm 1 Prefetching method

```
1:  procedure ONLINECACHINGSERVICE
2:      caching service ...
3:      if currenttime is prefetchingtime then
4:          CREATETHREAD(
5:                  PREFETCHER(NPR, interval, currenttime))
6:      CACHEPOLICY(targetphoto)
7:      caching service ...
8:
9:  function CACHEPOLICY(targetphoto)
10:     if targetphoto is in caching pool then
11:         return the photo
12:     else
13:         if caching pool is full then
14:             EVICT(photosize)                ▷ use one Evict!!
15:             or SMARTEVICT1(photosize)       ▷ use one Evict!!
16:             or SMARTEVICT2(photosize)       ▷ use one Evict!!
17:         insert the photo to the cache
18:
19: function PREFETCHER(NPR, interval, currenttime)
20:     timestamp ← currenttime − interval
21:     while timestamp < currenttime do
22:         for each uploaded logical photo at timestamp do
23:             n ← 1
24:             while n <= NPR do
25:                 prefetching resolution Rez-n
26:                 n ← n + 1
27:         timestamp ← timestamp + 1
28:
29: function SMARTEVICT1(photosize)
30:     maintain a ghost FIFO queue for prefetched photos
31:     while cachesize+photosize > cachecapacity and ghost
    queue is not empty do
32:         tmpphoto ← pop tail of ghost queue
33:         if tmpphoto.tt > TT and tmpphoto.ft < FT then
34:             evict tmpphoto
35:             cachesize ← cachesize − tmpphoto.size
36:     EVICT(photosize)
37:
38: function SMARTEVICT2(photosize)
39:     maintain a ghost hashmap mapping logical photos
    to physical photos in cache
40:     while cachesize + photosize > cachecapacity do
41:         tmpphoto ← pop tail of cache queue
42:         for each rez in tmpphoto.logical do
43:             if rez.freq ≤ LT then
44:                 evict tmpphoto.logical.rez
45:                 cachesize ←
46:                         cachesize − tmpphoto.logical.rez.size
```

since they are likely to be *low-frequency* photos and have already finished their accessing in lifetime. We use two thresholds to control the eviction decision named *timeout threshold* (TT) and *frequency threshold* (FT) respectively. Policies using *time heuristic* first evicts those prefetched photos whose request times are lower than a configured FT or TT. If there are no prefetched photos satisfying the conditions, then it falls through using FIFO cache algorithm. The *time heuristic* can detect and remove invalid prefetches efficiently since invalid prefetching photos receive no request. The procedure of *time heuristic* eviction policy is presented in Function SMARTEVICT1 at Line 29-36 in Algorithm 1.

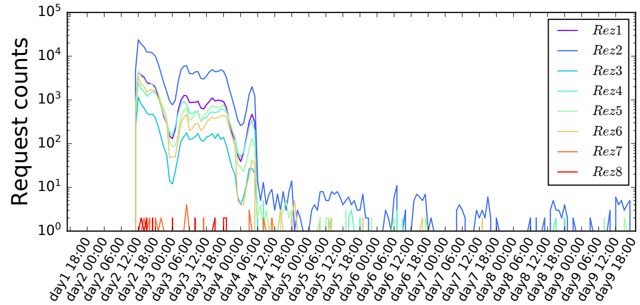


Fig. 7: Popularities of resolutions associated to a logical photo. The request is counted in intervals of one hour. Note that the y-axis is in log scale.

3.3.2 Logical Photo Heuristic

The *logical photo heuristic* is not restricted to prefetching method but a more general eviction strategy for photo cache. It leverages the fact that resolutions of the same logical photo exhibit similar popularity trend. As Figure 7 shows, all resolutions of the same photo reach peaks or troughs narrowly simultaneously despite they share different frequencies. Most of the requests to a photo happen within a short period of time after it is uploaded and all resolutions receive very few requests when the logical photo becomes cold. This trend reflects the reality that popularity goes down along with time. It is the logical photo that determines the popularity and the resolution only reflects the access distribution. The *logical photo heuristic* leverages such feature and evicts all homologous resolutions of the evicted photo simultaneously upon an eviction. However, we need to avoid evicting a hot resolution along the evicting of a cold resolution. To ensure that, we take an additional decision to check how many times resolutions have been requested. When evicting a specific resolution, policies with *logical photo heuristic* will evict other homologous resolutions whose frequencies fall below a certain threshold, named *logical threshold* (LT). Function SMARTEVICT2 of Algorithm 1) (Line38-46) depicts the procedure in detail. Moreover, the *logical photo heuristic* is compatible with the *time heuristic*, and they could be applied at the same time.

4 EVALUATION

In this section, we evaluate the efficacy of the prefetcher added to an OC. We write a simulator to drive our trace experiments. The simulator is written in Python and is open sourced on Github[2]. It provides a flexible framework to support various cache replacement policies, including FIFO, LRU, and SxLRU. The simulator takes the photo access traces as its input and uses the configured cache capacity and cache policy for simulation. As in the production system, the entire cache space is managed as a single cache

2. https://github.com/sunsihtf/simple-cache-policy-simulator

pool and the cache granularity is a variable-sized photo file. Individual photo files are replaced out from the cache space according to the replacement policy. In the end, it outputs various statistics including hit ratio and byte hit ratio. Otherwise specified, we use the LRU cache algorithm as the default algorithm, which is also used in the running QQPhoto system. We set the standard cache space as $0.01X$, where X is QQPhoto cache size, because we use one-hundredth of the original photos. We use NPR to represent the number of prefetching resolutions, e.g., $NPR = 1$ means we prefetch the most popular resolution of each uploaded photo. We compare different cache algorithms in terms of *hit ratio*, *average request latency*, and *network traffic*. To calculate the average request latency, we use the following equation:

$$latency = HR \times AVG_{hit\ latency} + MR \times AVG_{miss\ latency},$$

where $AVG_{hit\ latency}$ and $AVG_{miss\ latency}$ are 11.62ms and 127.0ms, respectively, according to our trace records. In addition, we implement an offline prefetching method that prefetches the exact resolutions by leveraging the future hint which is denoted as *offline* and the Belady's MIN algorithm for comparisons. The configurations of NPR and prefetch interval dictate which recently uploaded photo files are prefetched to the cache. At every prefetching time, the prefetcher prefetches the NPR most popular photo resolutions uploaded during the previous interval. If smart eviction policy is employed, the prefetched photos are tagged to support their early being evicted if they have stayed longer than a configured period of time. We use the first 5-days traces for warm-up and collect statistics of the next 4-days traces.

4.1 Hit Ratio Improvements

We explore the hit ratio improvements introduced by the prefetcher in cases of various values of the factors impacting hit ratios, mainly including the values of NPR and the time length of the prefetch interval. Figure 8 gives the hit ratio improvements introduced by the prefetcher relative to the original LRU cache algorithm at varying NPR values, cache sizes and prefetching intervals. We reveal the effects of NPR and prefetch interval on the hit ratio improvements via conducting three sets of experiments.

First, we vary the values of NPR while keeping the same prefetch interval value. Figure 8a, 8b and 8c depict different $NPRs$ under certain fixed prefetch intervals. As it is depicted, in almost all these cases, our prefetcher improves the hit ratios over the original LRU cache algorithm except for some large $NPRs$ at small cache capacities because aggressively prefetching photo resolutions leads to more pollutions. Similarly, at small caches, it is more likely for cache pollutions to happen. Therefore, we see smaller hit ratio improvements at small caches. Moreover, when NPR is bigger than 3, we observe no increased hit ratio improvements due to prefetching less popular resolutions. As the cache size increases, the improvements of smaller $NPR (= 1, 2, 3)$ values are slightly smaller than that of larger $NPRs$ values (up to 3%), which is consistent with the intuition that if there is more cache space then we can do prefetching more aggressively to gain more hit ratio improvements. Overall, compared to the original LRU, even the least aggressive prefetching with $NPR = 1, interval = 1h$ delivers an improvement of 3.9% at a cache capacity of X, and the most aggressive prefetching with $NPR = 5, interval = 1$ second achieves an improvement of 7.4%.

To illustrate the influence of prefetch interval more clearly, Figure 8d, 8e and 8f depict three different intervals of 1 second, 10 minutes and 1 hour under $NPRs$ of 1, 3 and 5. As it is shown in the figures, smaller intervals are more beneficial and the differences among prefetching intervals also increase as the value of NPR increases. However, there still exists a big gap between the prefetching method and the theoretically optimal offline prefetch method. But it is remarkable that the prefetching method surpasses the optimal Belady's MIN algorithm with relatively large $NPRs$ and high cache capacities since the prefetcher eliminates most of compulsory misses while Belady's MIN algorithm falls short at compulsory misses.

In previous sections, we rely on the intuition that the higher resolution we prefetch, the more benefits we are able to obtain. To validate this speculation, we evaluate the hit ratios of prefetching different resolutions. We set $NPR = 1$ and vary the prefetching popularity rank (see Figure 6). Figure 8g illustrates the results. The results are consistent with the resolution distribution pattern. As can be observed from the figure, prefetching the most popular resolution ($Rez1$) results in the highest hit ratios and the hit ratios decrease if prefetch less popular resolutions.

4.2 Latency and Network Traffic Trade-off

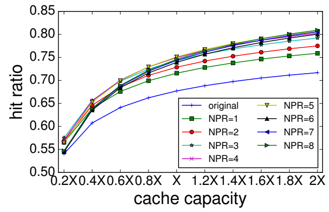
Though larger $NPRs$ values lead to higher hit ratios, aggressive prefetching also faces the problem of more severe cache pollutions and causes more bandwidth consumption. To quantify this trade-off, we investigate the average request latency and incurred network traffic in the condition of 10 minutes prefetch interval with varying NPR values, which are illustrated in Figure 9a and Figure 9b, respectively. As excepted, prefetching brings about additional network traffic and the bigger NPR values result in more network traffics. On the other hand, prefetching decreases average request latency as the value of NPR increases.

We therefore use a cost benefit analysis (CBA) to determine the optimal choice of NPR and prefetch interval, which can be calculated by $net\ benefits$ as

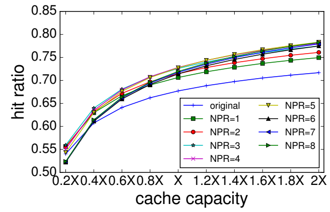
$$\begin{aligned} net\ benefits &= total\ benefit - total\ cost \\ &= W_{latency} \times Latency - W_{bandwidth} \times Bandwidth. \end{aligned}$$

where W is weight. However it's hard to determine values of the weight and thus impossible to calculate the deterministic $net\ benefits$. Fortunately, there are only 8 resolutions that we can prefetch in the QQPhoto system, which enables us to analyze the problem by enumerating them. We have investigated the case of capacity X and the results are listed in Table 4. As demonstrated in the table, as the value of NPR increases, the additional bandwidth cost becomes larger and larger while the latency gains get smaller and smaller. Carefully examining the trade-off table, we conclude that prefetching only the highest resolutions is a practically good trade-off point, which reduces the latency by 6.9% but consumes 4.14% extra network resources.

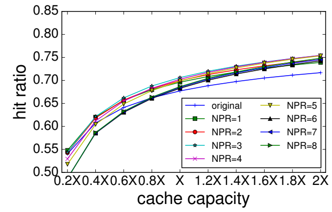
This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems



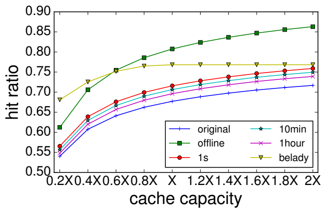
(a) $NPR = 1, ..., 8, interval = 1s$



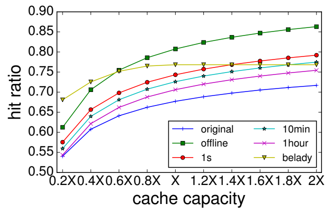
(b) $NPR = 1, ..., 8, interval = 10m$



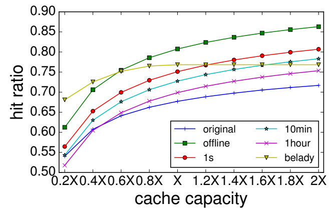
(c) $NPR = 1, ..., 8, interval = 1h$



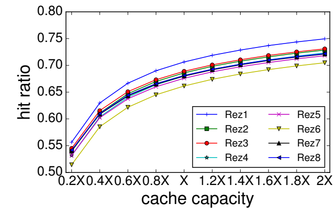
(d) $NPR = 1, interval = 1s, 10m, 1h$



(e) $NPR = 3, interval = 1s, 10m, 1h$

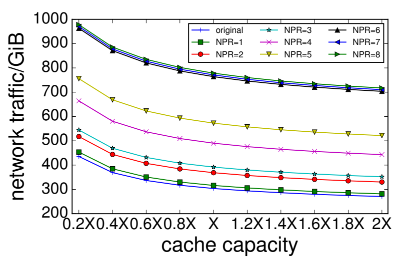


(f) $NPR = 5, interval = 1s, 10m, 1h$

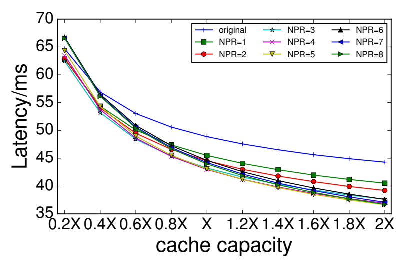


(g) $NPR = 1, interval = 10m$

Fig. 8: Hit ratios of prefetching at various NPR values and prefetch intervals. Original is the LRU cache replacement algorithm without prefetching. Belady is the optimal algorithm with no prefetching. NPR is number of prefetching resolutions.



(a) $NPR = 1, ..., 8, interval = 10m$



(b) $NPR = 1, ..., 8, interval = 10m$

Fig. 9: Network traffic and latency of $NPR = 1, ..., 8$ at the 10 minutes prefetch interval.

TABLE 4: Network traffic and latency trade-offs at cache capacity of X

NPR	network traffic	latency
1	4.14%	-6.90%
2	21.41%	-8.85%
3	29.00%	-11.54%
4	61.41%	-12.06%
5	88.59%	-11.89%
6	151.60%	-8.74%
7	153.92%	-9.54%
8	156.33%	-10.01%

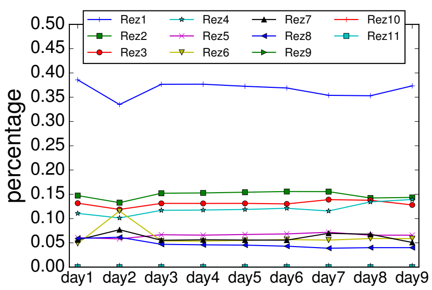


Fig. 10: Resolution popularity evolution in the traces.

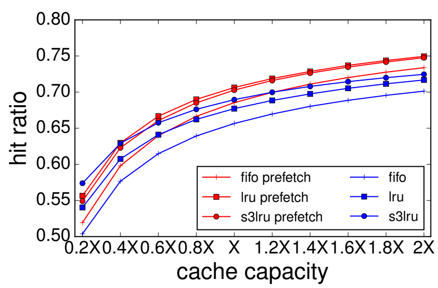


Fig. 11: Hit ratio of FIFO, LRU and S3LRU with and without a prefetcher at $NPR = 1, interval = 10m$.

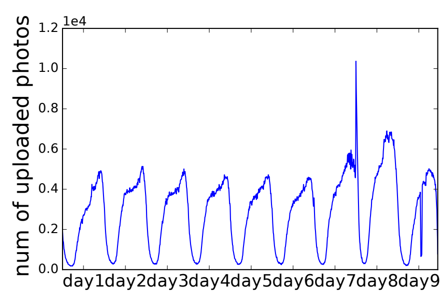


Fig. 12: The amount of uploaded photos every 10 minutes.

4.3 Resolution Popularity Evolution

In the preceding section, we have seen that prefetching only the highest resolutions is a suitable choice. However, we cannot ensure a resolution is always the most popular resolution. To solve the problem, we study how resolution popularity changes overtime. Figure 10 plots resolution distribution evolution in the 9-days traces. As can be seen, all the resolutions remain at a relatively stable state. $Rez1$

always occupies the highest proportion. The second tier $Rez2$, $Rez3$ and $Rez4$ are much lower and stays stable most of the time except for a few occasional fluctuations and the remaining resolutions also exhibit similar tendency. As we discussed before, an on-line profiler can help if there are many fluctuations.

4.4 Optimal Prefetch Interval

We have examined so far three different intervals (1s, 10m, 1h) for our evaluations. However there might exist other better intervals than the current best choice. Time-varying workload dictates that it is impossible to find a consistently optimal interval. We subjectively suppose the max hit ratio loss (the bias between actual interval and real time (1s) interval) should not exceed 1%. Thereby, the problem is transformed to figure out the max interval in which the hit ratio loss is less than 1%. It turned out that $interval = 10min$ was the optimal resolution whose maximum loss is 0.95%. On the other hand, to estimate the impacts to backend, the quantity of uploaded photos during every $interval = 10min$ is counted as Figure 12 depicts. The day7 bursts into a very high number which is an exception and will not be considered [3]. Generally speaking, the peak number of uploaded photos is approximately 5000 at late afternoon everyday. Taking sampling into account, the peak prefetch will exceed 500,000.

4.5 Applicability

To validate that our prefetching method has wide applicability, we integrate it with the basic FIFO and an advanced S3LRU to see how it improves over other cache algorithms. Figure 11 depicts the hit ratios of FIFO and S3LRU with prefetcher as well as LRU for comparison. The results indicate that our prefetcher improves the hit ratios for all three algorithms. Similar to previous results, the improvement is smaller at small cache capacities and goes higher at large capacities. It should be noted that the prefetcher causes negative impacts on S3LRU at small cache capacities because the lowest queue in S3LRU is too small to hold both prefetched photos and cached photos, causing excessive cache pollutions.

4.6 Cache Lifetime Distribution

In this section, we investigate the cache lifetime of prefetched photos, which provides the insights into prefetching efficiency. Three kinds of photos are measured, namely, the prefetched and non-prefetched (due to demand cache misses) photos in prefetch method and all the photos in the method without an prefetcher. Figure 13 depicts the cumulative distribution function of their access frequency in the cache. In the non-prefetch method, nearly 75% uploaded photos receive only one request and about 97% receive no more than 10 requests. In the prefetch method, non-prefetched photos take up 84.0% of all uploaded photos and their frequency distribution is similar to that in the non-prefetch method and prefetched photos only take up 16%. What's more, 51.7% of them receive no requests and are invalid prefetches. The low prefetch efficiency is reasonable because the proportion of prefetched resolutions is less than 40% and the more than 60% rest are wasted. For the same reason, the ratio of non-prefetched photos in cache is relatively high.

3. It was the Chinese largest festival—the Spring Festival

4.7 Smart eviction

In Section 3, we have discussed the two smart eviction strategies for prefetching method. The *time heuristic* intend to improve evictions by adjusting TT and FT. Figure 14 illustrates hit ratios under various TT and FT. While the hit ratio differences are small, still it's clear that the correlation between TT and hit ratio is positive and it is negative between FT and hit ratio. It's because a higher TT results in prefetched photos residing longer in the cache to receive more hits. By contrast, a lower FT prevents prefetched photos from staying in cache longer and therefore their chances to be evicted increase. Besides, no matter how to adjust the two parameters, smart eviction cannot outperform the normal LRU policy, i.e., treating prefetched photos the same as cached photos.

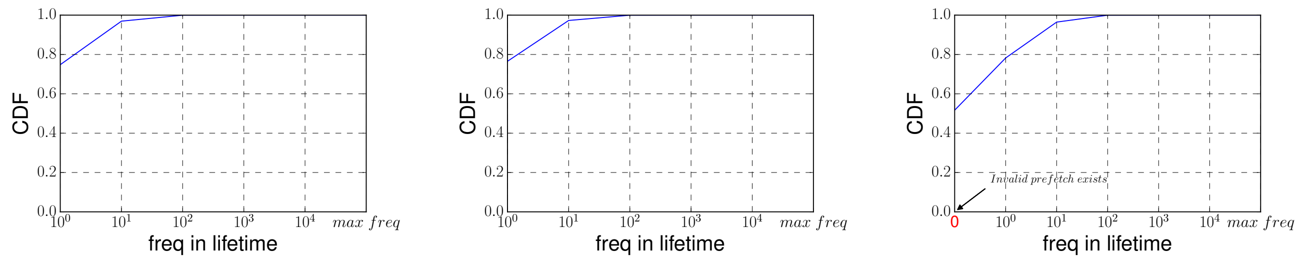
Logical photo heuristic tries to evict several resolutions simultaneously on a replacement. Since most of photos only receive very limited requests, to make every possible request being hit, we set the LT at a very low value, i.e., $LT = 1$. $LT = 1$ indicates a prefetched photo receives no request or only one request will be evicted with priority. Thus we can filter invalid prefetches and only one-time-request photos. Figure 15 illustrates hit ratios and network traffic traffic under various $NPRs$ at cache capacity of X and $interval = 10m$ and $LT = 1$. The heuristic helps improving hit ratios over the original prefetching method at a small margin at small $NPRs$. Nevertheless, the gain gets smaller as NPR increases and eventually becames negative. The reason is because higher NPR results in higher chances to be evicted as more resolutions are detected on evictions, thereby the long reuse distance photos get higher chances to be evicted. The network consumption under the heuristic is nearly the same as before since the hit ratio brings no explicit change as all savings are derived from hit ratio improvement, and the negligible change results in no waves on network traffic.

5 Related Work

Cache is a classical while still ever-appealing research topic as its main philosophical principle is universally applicable as long as there exist differences between two neighboring levels in terms of performance. Caching is ubiquitously existent in almost all computer systems and it has been a primary technique to improve system performance in a cost-effective manner. During the past decades, various cache algorithms have been proposed and researched in a wide range of contexts including traditional, main memory cache, flash cache in the secondary storage.

Traditional cache algorithms typically aim to achieve global efficiency via taking advantage of temporal or spatial locality. Widely used cache algorithms include First-In-First-Out (FIFO), Least Recently Used (LRU), Least Frequently Used (LFU), LRU-K [23], ARC [21], SxLRU, LIRS [24],

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems



(a) *photos in original method* (b) *non-prefetched photos in prefetching method* (c) *prefetched photos in prefetching method*

Fig. 13: CDF of photos frequency during cache lifetime in different methods at $NPR = 1, interval = 10m$

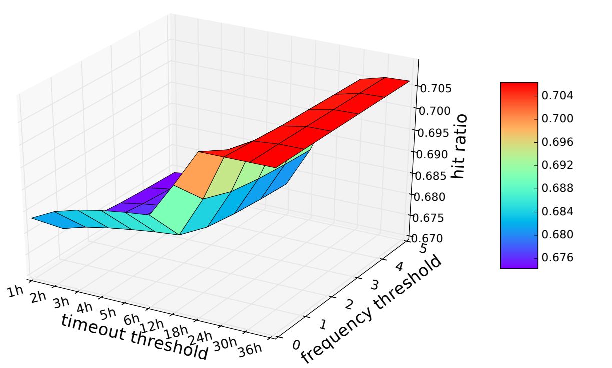


Fig. 14: Hit ratio of LRU with prefetcher using *time heuristic* at cache capacity of X. Large timeout thresholds help improve hit ratios of prefetched photos, while high frequency thresholds lead to lower hit ratios.

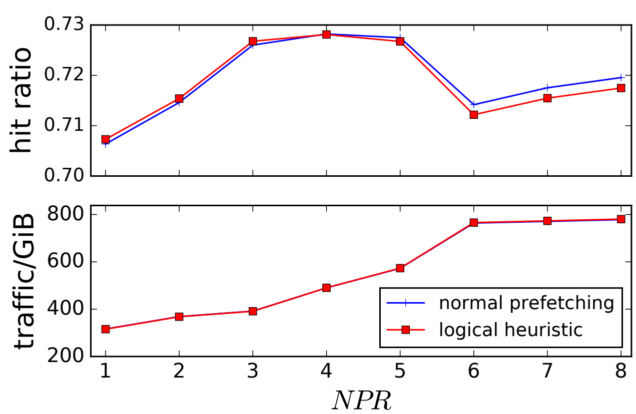


Fig. 15: Hit ratio and network traffic of LRU with prefetcher using *logical photo heuristic* under various $NPRs$ at cache capacity of X, $interval = 10m$, $LT = 1$.

Multiple Queue [22], etc. Some of these cache algorithms simply use recency and frequency, while others are more adaptable to workloads. An offline optimal cache algorithm called Belady's MIN [25], [26] is often used as the theoretical up-bound limit to judge a cache algorithm. This optimal algorithm is based on the good foreknowledge about future access patterns, which are hard to obtain in practice. Cache modeling and simulation is a commonly used means to comparatively evaluate various cache algorithms [27].

Flash-based SSDs are gaining increasing popularity in storage systems and are commonly used as a cache front-end of HDD-based storage [28], [29], [30], [31], [32]. There also exist numerous works studying cache algorithms for flash cache. Most of the cache algorithms in this line focus on reducing write amplification via employing flash-friendly cache designs. Nitro [33] and Pannier [34] reduce flash cache writes via employing de-duplication/compression and container-based cache granularity, respectively. Kim et al. [35] implement a write admission policy for non-volatile memory cache, which does not admit all writes to the cache but critical writes in terms of application performance. CloudCache [36] uses Reuse Working Set (RWS) cache model to estimate workloads cache demand and admits RWS in the cache. CacheDedup [37] improves on the LRU and ARC cache algorithms to make them deduplication-aware. LARC [38] a variation of ARC algorithm, filters low popular blocks and prevents them entering the cache to reduce cache replacements. Cheng et al. [26] explore the offline optimal flash cache algorithms that consider both cache performance and flash lifespan. RIPQ [4] enables advanced cache algorithms to efficiently work in Facebook's photo cache by eliminating small random writes to flash cache.

A lot of more recent cache research interest has been devoted to the key value caches, as key value systems have become widespread in the cloud [15], [16], [39]. Cliffhanger [15] constructs and leverages the hit rate curve gradient of local eviction queue by leveraging shadow queues and dynamically adjust cache resource allocations according to workloads changes. Memshare [16] reserves a minimal amount of memory for applications and dynamically partitions the remaining pooled memory resources among multiple-tenant applications using a log-structured memory design. Hyperbolic caching [13] decays cached items priorities at various rates and continuously reorders many items at once, taking in account multiple factors when determining the priorities of cached items. MIMIR [14] adds a lightweight online profiler which hooks to the cache replacement policy and obtains the graphs of overall cache hit ratio. It then uses that information to dynamically right-size adjust cache memory size. Moirai [40] is a software defined caching architecture that allows cache operators to flexibly control cache resources in multi-tenant data centers. Dual cost cache [41] is a cache algorithm that aims to minimize the sum of read-miss and write-hit cost when making cache replacement decisions.

Huang et al. [2] presents a comprehensive analysis on the characteristics of Facebook's photo caching workloads and investigates the effects of different cache algorithms on an Edge cache. They find that using advanced algorithms can improve hit rate and reduce a significant amount of downstream request. However, we present a deep analysis on the photo cache from the largest Chinese social network company from the cache algorithm prospective and propose to add prefetching to improve photo cache efficiency based on social network workloads access immediacy.

Prefetching is also a widely used technique to improve performance by prefetching the likely requested data into the cache stage in advance. It has been demonstrated effective in LLC and main memory prefetching [42], [43], [44], while a recent study has shown that prefetching may negatively impact cloud workloads with long temporal reuse patterns [45]. Our prefetcher in this work performs prefetching based on the immediacy feature of cloud photo caching.

6 Conclusion

Tencent's QQPhoto employs photo caching to speed up photo access requests and the cache performance affects user experiences. We present a comprehensive analysis on the QQphoto workloads and find that a key factor limiting cache hit ratio is compulsory misses of the first time accesses to photos. Moreover, photo accesses exhibit immediacy characteristic, i.e., photos tend to be more likely accessed in a recent period of time immediately following uploading. To improve QQPhoto caching efficiency, we propose prefetching the most popular resolutions of recently uploaded photos to the cache. Our results have shown that with the prefetcher photo access latency is cut by an average of 6.9% while sacrificing only 4.14% additional network cost. As the future work, we plan to improve prefetching accuracy using machine learning methods. We are also closely working with the QQPhoto engineer team to explore the possibility of materializing our solution in the production system.

Acknowledgment

This work is supported in part by the National Natural Science Foundation of China under Grant No. 61502189, the National Key Research and Development Program of China under Grant No. 2016YFB0800402, and the U.S. National Science Foundation under Grants CCF-1717660 and CNS-1702474. Ping Huang is the corresponding author.

References

[1] S. Nikolaou, R. V. Renesse, , and N. Schiper, "Proactive Cache Placement on Cooperative Client Caches for Online Social Networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 27, no. 4, pp. 1174–1186, April 2016.

[2] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li, "An Analysis of Facebook Photo Caching," in *Proceedings of the 24^{th} ACM Symposium on Operating Systems Principles(SOSP'13)*, Farminton, Pennsylvania, November 2013, pp. 167–181.

[3] X. Bai, B. B. Cambazoglu, and A. Russell, "Improved Caching Techniques for Large-Scale Image Hosting Services," in *Proceedings of the 39^{th} International ACM SIGIR Conference on Research and Development in Information Retrieval(SIGIR'16)*, Pisa, Italy, July 2016, pp. 639–648.

[4] L. Tang, Q. Huang, W. Lloyd, S. Kumar, and K. Li, "RIPQ: Advanced Photo Caching on Flash for Facebook," in *Proceedings the 13^{th} USENIX Conference on File and Storage Technologies(FAST'15)*, Santa Clara, CA, February 2015, pp. 373–386.

[5] D. Beaver, S. Kumar, H. C. Li, J. Sobel, and P. Vajgel, "Finding a Needle in Haystack: Facebook's Photo Storage," in *Proceedings of the 9^{th} USENIX Conference on Operating Systems Design and Implementation(OSDI'10)*, Vancouver, BC, Canada, October 2010, pp. 47–60.

[6] S. Muralidhar, W. Lloyd, S. Roy, C. Hill, E. Lin, W. Liu, S. Pan, S. Shankar, V. Sivakumar, L. Tang, and S. Kumar, "f4: Facebook's Warm BLOB Storage System," in *Proceedings of the 11^{th} USENIX Symposium on Operating Systems Design and Implementation(OSDI'14)*, Broomfield, CO, October 2014, pp. 383–398.

[7] N. Bronson, Z. Amsden, G. Cabrera, P. Chakka, P. Dimov, H. Ding, J. Ferris, A. Giardullo, S. Kulkarni, H. Li, M. Marchukov, D. Petrov, L. Puzar, Y. J. Song, and V. Venkararamani, "TAO: Facebook's Distributed Data Store for the Social Graph," in *Proceedings of the 2013 USENIX Annual Technical Conference (USENIX ATC'13)*, San Jose, CA, June 2013, pp. 49–60.

[8] N. Beckmann and D. Sanchez, "Talus: A Simple Way to Remove Performance Cliffs in Caches," in *Proceedings of the IEEE 21^{st} International Symposium on High Performance Computer Architecture(HPCA'15)*, San Francisco, CA, February 2015, pp. 64–75.

[9] N. Beckmann and D. Sanchez, "Modeling Cache Performance Beyond LRU," in *Proceedings of the 22^{nd} International Symposium on High Performance Computer Architecture(HPCA'16)*, Barcelona, Spain, March 2016, pp. 225–236.

[10] N. Beckmann and D. Sanchez, "Maximizing Cache Performance Under Uncertainty," in *Proceedings of the 23^{rd} International Symposium on High Performance Computer Architecture (HPCA'2017)*, Austin, TX, February 2017, pp. 109–120.

[11] Q. Pu, H. Li, M. Zaharia, A. Ghodsi, and I. Stoica, "FairRide: Near-Optimal, Fair Cache Sharing," in *Proceedings of the 13^{th} USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, Santa Clara, CA, March 2016, pp. 393–406.

[12] L. Guo, E. Tan, S. Chen, Z. Xiao, and X. Zhang, "The Stretched Exponential Distribution of Internet Media Access Patterns," in *Proceedings of the 27^{th} Annual ACM Symposium on Principles of Distributed Computing (PODC'08)*, Toronto Canada, August 2008, pp. 283–294.

[13] A. Blankstein, S. Sen, and M. J. Freedman, "Hyperbolic Caching: Flexible Caching for Web Applications," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, July 2017, pp. 499–511.

[14] T. Saemundsson, H. Bjornsson, G. Chockler, and Y. Vigfusson, "Dynamic Performance Profiling of Cloud Caches," in *Proceedings of the 5^{th} ACM Symposium on Cloud Computing(SoCC'14)*, Seattle, WA, November 2014, pp. 28:1–28:14.

[15] A. Cidon, A. Eisenman, M. Alizadeh, and S. Katti, "Cliffhanger: Scaling Performance Cliffs in Web Memory Caches," in *Proceedings of the 13^{th} USENIX Symposium on Networked Systems Design and Implementation (NSDI'16)*, Santa Clara, CA, March 2016, pp. 379–392.

[16] A. Cidon, D. Rushton, S. M. Rumble, and R. Stutsman, "Memshare: a Dynamic Multi-tenant Key-value Cache," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, July 2017, pp. 321–334.

[17] "Tencent," http://www.tencent.com/, 2017, [Online].

[18] "How QQPhoto works(Chinese)," https://cloud.tencent.com/developer/article/1005732, 2017, [Online].

[19] L. Tang, Q. Huang, A. Puntambekar, Y. Vigfusson, W. Lloyd, and K. Li, "Popularity Prediction of Facebook Videos for Higher Quality Streaming," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, July 2017, pp. 111–123.

[20] J. S. Vitter, "Random Sampling with a Reservoir," *ACM Transactions on Mathematical Software*, vol. 11, no. 1, pp. 37–57, March 1985.

[21] N. Megiddo and D. S. Modha, "ARC: A Self-Tuning, Low Overhead Replacement Cache," in *Proceedings of the 2^{nd} USENIX*

Conference on File and Storage Technologies(FAST'03), San Francisco, CA, March 2003, pp. 115–130.

[22] Y. Zhou and J. F. Philbin, "The Multi-Queue Replacement Algorithm for Second Level Buffer Caches," in *Proceedings of the 2001 USENIX Annual Technical Conference(USENIX ATC'01)*, Boston, MA, June 2001, pp. 91–104.

[23] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering," in *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data(SIGMOD'93)*, New York, NY, USA, May 1993, pp. 297–306.

[24] J. Song and X. Zhang, "LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance," in *Proceedings of the 2002 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems(SIGMETRICS'02)*, Marina del Rey, CA, June 2001, pp. 31–42.

[25] L. A. Belady, "A Study of Replacement Algorithms for a Virtual Storage Computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, 1966.

[26] Y. Cheng, F. Douglis, P. Shilane, G. Wallace, P. Desnoyers, and K. Li, "Erasing Belady's Limitations: In Search of Flash Cache Offline Optimality," in *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*, Denver, CO, June 2016, pp. 379–392.

[27] C. Waldspurger, T. Saemundson, I. Ahmad, and N. Park, "Cache Modeling and Optimization using Miniature Simulations," in *Proceedings of the 2017 USENIX Annual Technical Conference (USENIX ATC'17)*, Santa Clara, CA, July 2017, pp. 487–498.

[28] P. Huang, G. Wu, X. He, and W. Xiao, "An Aggressive Worn-out Flash Block Management Scheme to Alleviate SSD Performance Degradation," in *Proceedings of the 9th European Conference on Computer Systems(Eurosys'14)*, Amsterdam, The Netherlands, April 2014, pp. 22:1–22:14.

[29] M. Saxena, M. M. Swift, and Y. Zhang, "FlashTier: a Lightweight, Consistent and Durable Storage Cache," in *Proceedings of the* 7^{th} *ACM European Conference on Computer Systems(Eurosys'12)*, Bern, Switzerland, April 2012, pp. 267–280.

[30] P. Huang, P. Subedi, X. He, S. He, and K. Zhou, "FlexECC: Partially Relaxing ECC of MLC SSD for Better Cache Performance," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, Philadelphia, PA, June 2014, pp. 489–500.

[31] K. Zhou, S. Hu, P. Huang, and Y. Zhao, "LX-SSD: Enhancing the Lifespan of NAND Flash-based Memory via Recycling Invalid Pages," in *Proceedings of the IEEE* 33^{rd} *Symposium on Mass Storage Systems and Technologies(MSST'17)*, Santa Clara, CA, May 2017, pp. 28:1–28:13.

[32] Z. Shen, F. Chen, Y. Jia, and Z. Shao, "DIDACache: A Deep Integration of Device and Application for Flash Based Key-Value Caching," in *Proceedings of the* 15^{th} *USENIX Conference on File and Storage Technologies(FAST'17)*, Santa Clara, CA, February 2017, pp. 391–405.

[33] C. Li, P. Shilane, F. Douglis, H. Shim, S. Smaldone, and G. Wallace, "Nitro: A Capacity-Optimized SSD Cache for Primary Storage," in *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC'14)*, June 2012, pp. 501–512.

[34] C. Li, P. Shilane, F. Douglis, and G. Wallace, "Pannier: A Container-based Flash Cache for Compound Objects," in *Proceedings of the* 16^{th} *Annual Middleware Conference(Middleware'15)*, New York, NY, USA, December 2015, pp. 50–62.

[35] S. Kim, H. Kim, S.-H. Kimg, and J. L. adn Jinkyu Jeong, "Request-Oriented Durable Write Caching for Application Performance," in *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*, Santa Clara, CA, June 2015, pp. 193–206.

[36] D. Arteaga, J. Cabrera, J. Xu, S. Sundararaman, and M. Zhao, "Cloud Cache: On-demand Flash Cache Management for Cloud Computing," in *Proceedings of the* 14^{th} *USENIX Conference on File and Storage Technologies(FAST'16)*, Santa Clara, CA, February 2016, pp. 355–369.

[37] W. Li, G. Jean-Baptise, J. Riveros, G. Narasimhan, T. Zhang, and M. Zhao, "CacheDedup: In-line Deduplication for Flash Caching," in *Proceedings of the* 14^{th} *USENIX Conference on File and Storage Technologies(FAST'16)*, Santa Clara, CA, February 2016, pp. 301–314.

[38] S. Huang, Q. Wei, D. Feng, J. Chen, and C. Chen, "Improving Flash-Based Disk Cache with Lazy Adaptive Replacement," in *Proceedings of the IEEE* 29^{th} *Symposium on Mass Storage Systems and Technologies(MSST'13)*, Long Beach, CA, May 2013, pp. 28:1–28:10.

[39] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani, "Scaling Memcache at Facebook," in *Proceeding of the* 10^{th} *USENIX Symposium on Networked Systems Design and Implementation (NSDI'13)*, Lombard, IL, April 2013, pp. 385–398.

[40] I. Stefanovici, E. Thereska, G. O'hea, B. Schroeder, H. Ballani, T. Karagiannis, A. Rowstron, and T. Talpey, "Software-Defined Caching: Managing Caches in Multi-tenant Data Centers," in *Proceedings of the* 6^{th} *ACM Symposium on Cloud Computing(SoCC'15)*, Hawaii, August 2015, pp. 174–181.

[41] A. Dasgupta, R. Kumar, and T. Sarlós, "Caching with Dual Costs," in *Proceedings of the* 26^{th} *International Conference on World Wide Web(WWW'17)*, Perth, Australia, April 2017, pp. 643–652.

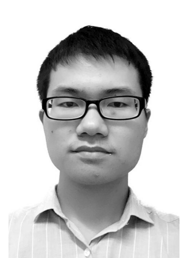
[42] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently Prefetching Complex Address Patterns," in *Proceedings of the* 48^{th} *Annual IEE/ACM International Symposium on Microarchitecture(MICRO'15)*, Waikii, HI, December 2015, pp. 141–152.

[43] J. Kim, S. H. Pugsley, P. V. Gratz, A. L. N. Reddy, C. Wilkerson, and Z. Chishti, "Path Confidence Based Lookahead Prefetching," in *Proceedings of the* 49^{th} *Annual IEE/ACM International Symposium on Microarchitecture(MICRO'16)*, Taibei, Taiwan, December 2016, pp. 1–12.

[44] A. Bhattacharjee, "Translation-Triggered Prefetching," in *Proceedings of the* 22^{nd} *International Conference on Architectural Support for Programming Languages and Operating Systems(ASPLOS'17)*, Xi'an, China, March 2017, pp. 63–76.

[45] J. Wang, R. Panda, and L. K. John, "Prefetching for Cloud Workloads: An Analysis Based on Address Patterns," in *Proceedings of the 2017 IEEE International Symposium on Performance Analysis of Systems and Software(ISPASS'17)*, Santa Rosa, CA, April 2017, pp. 163–172.

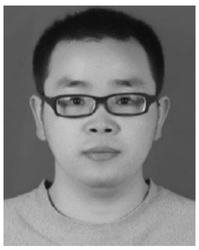
Ke Zhou received the BE, ME, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1996, 1999, and 2003, respectively. He is a professor of the School of Computer Science and Technology and Wuhan National Laboratory for Optoelectronics, HUST. His main research interests include computer architecture, cloud storage, parallel I/O, and storage security. He has more than 50 publications in journals and international conferences, including the IEEE Transactions on Parallel and Distributed Systems, the Performance Evaluation (PEVA), FAST, USENIX ATC, MSST, ACM MM, INFOCOM, SYSTOR, MASCOTS, ICC, etc. He is a member of the IEEE and a member of the USENIX.



Si Sun received the bachelor degree in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 2014. He is currently a PhD candidate at Wuhan National Laboratory of Optoelectronics (WNLO), HUST.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TPDS.2019.2902392, IEEE Transactions on Parallel and Distributed Systems

Hua Wang received the BS, MS, and PhD degrees in computer science and technology from Huazhong University of Science and Technology (HUST), China, in 1996, 2000 and 2009, respectively. She is an associate professor of Wuhan National Laboratory for Optoelectronics, HUST. Her main research interests include computer architecture, cloud storage and intelligent storage. She has published papers in various international conferences and journals, including ICPP, ICS, ICCD, MSST, CIKM, SYSTOR, MASCOTS, NAS, the Journal of Systems and Software, and etc.



Ping Huang received the PhD degree from Huazhong University of Science and Technology, in 2013. He is currently a research assistant in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His main research interest includes nonvolatile memory, operating system, distributed systems, DRAM, GPU, Key-value systems, etc. He has published papers in various international conferences and journals, including SYSTOR, NAS, MSST, USENIX ATC, Eurosys, IFIP Performance, INFOCOM, SRDS, MASCOTS, ICCD, the Journal of Systems Architecture (JSA), the Performance Evaluation (PEVA), the Sigmetrics, ICPP, the IEEE Transactions on Parallel and Distributed Systems (TPDS), the ACM Transactions on Storage, etc.

Xubin He received the BS and MS degrees in computer science from Huazhong University of Science and Technology, China, in 1995 and 1997, respectively, and the PhD degree in electrical engineering from the University of Rhode Island, Kingston, Rhode Island, in 2002. He is currently a professor in the Department of Computer and Information Sciences, Temple University, Philadelphia, Pennsylvania. His research interests include computer architecture, data storage systems, virtualization, and high availability computing. He received the Ralph E. Powe Junior Faculty Enhancement Award in 2004 and the Sigma Xi Research Award (TTU Chapter) in 2005 and 2010. He is a senior member of the IEEE, a member of the IEEE Computer Society and USENIX.

Lan Rui graduated from Huazhong University of Science and Technology, China, in 2012. He is now a software engineer at Tencent Corporation.

Wenyan Li is currently a senior staff engineer at Tencent Corporation. His main researh includes masive data storage and Content Distribution Network optimization.

Wenjie Liu received the master's degree from Nanjing University, China, in 2013. He is currently working toward the PhD degree at Temple University, USA.

Tianming Yang Tianming Yang received his BS in Department of Computer Science from Zhengzhou Institute of Technology, China in 1991, and PhD in Computer Science and Technology from Huazhong University of Science and Technology (HUST), China in 2010. Currently, he is an associate professor in the Department of Information Engineering, Huanghuai University, China. His current interests include data backup, networking storage, parallel file systems, disk array and solid state disk. He has more than 20 publications in international conferences and journals, including IPDPS, GRID, NAS ICCD, IPCCC, ICS, Journal of Zhejiang University-SCIENCE C, etc.