Pacaca: Mining Object Correlations and Parallelism for Enhancing User Experience with Cloud Storage

Binbing Hou
Computer Science and Engineering
Louisiana State University
bhou@csc.lsu.edu

Feng Chen
Computer Science and Engineering
Louisiana State University
fchen@csc.lsu.edu

***Abstract*—Object-based cloud storage presents an unconventional storage model. Exploiting its unique characteristics, such as the strong semantic correlations among objects and the high I/O parallelism potential, can greatly enhance user experience. Unfortunately, current storage optimization techniques, such as the caching and prefetching schemes, are designed for conventional storage and thus are sub-optimal for cloud storage services.**

In this paper, we propose a client-side cache management framework, called *Pacaca*, which integrates object clustering, parallelized prefetching, and cost-aware caching to exploit I/O parallelism and object correlations on cloud storage. We first develop an efficient mining scheme, called Frequent Cluster Mining (FCM), to discover object correlations from the access sequence, and then build a prefetching scheme to fetch the correlated objects in parallel. These two schemes are closely coordinated for achieving high prefetching accuracy, proper control on parallelism degree, and effective mis-prefetching detection and handling. After studying the impact of parallelized prefetching on cache management, we further present a cost-aware caching scheme to differentiate low-cost and high-cost objects for efficient caching by leveraging the awareness of parallelism and object correlations. Our experimental results show that our optimization schemes can effectively reduce the access latency, outperforming traditional schemes by up to 58%.

I. Introduction

Object-based cloud storage, such as Amazon S3 and Dropbox, has gained a huge popularity in recent years. As an Internet-based service, cloud storage maintains user data in the service provider's data center, and clients use an HTTP-based REST protocol to access their data through the network. Such a storage model brings numerous technical advantages, such as platform independence, usage-based pricing, cross-system shareability, and potentially unlimited capacity. These merits contribute to a quick and wide adoption of cloud storage in our daily computing.

Compared with conventional direct-attached storage, which has dominated computer systems for decades, object-based cloud storage is radically different: its "storage media" is a large-scale storage cluster, in which user data are distributed to a massive number of parallelized machines; its "I/O bus" is the world-wide Internet, which connects the client to the data center that provides the services; its "I/O protocol" is an HTTP-based REST protocol rather than a strictly defined command set; its "addressable unit" is a variable-size data object rather than a fixed-size sector. As a result, many of our long-held common-sense understandings and assumptions for storage system optimizations may not remain valid in the scenario of cloud storage. Here we list three critical issues:

• Critical Issue 1: I/O parallelization, rather than sequentiality, is key to optimizing user-perceived performance of cloud storage. In cloud storage, data are stored in a large-scale storage cluster, which is designed to simultaneously process a huge number of independent parallel requests. For example, Amazon S3 and Microsoft Azure Blob Storage are able to handle millions of requests per second [2], [6]. Such an inherent capability of processing parallel I/Os has a strong implication—creating parallel I/Os should be given a top priority for enhancing user experience with cloud storage; on the contrary, organizing sequential I/Os, which is a classic approach for optimizing traditional storage, becomes less rewarding with cloud storage.

We have conducted experiments on Amazon S3 and compared the time of downloading one thousand 4KB objects in different orders from an EC2 instance. The observed performance difference was almost non-existent. The same observations were obtained in the experiments with the object size varying from 16 KB to 4 MB. By contrast, properly parallelizing I/O jobs (e.g., downloading the objects of 4 KB with four parallel threads) can significantly improve the bandwidth and shorten the overall I/O completion time. This indicates that the existing schemes designed for optimizing rotating media, such as organizing sequential I/Os through caching [46] and prefetching [30], become less effective for cloud storage; instead, I/O parallelization is more important to user-perceived performance.

• Critical Issue 2: The object abstraction of cloud storage enables rich opportunities to explore the semantic relationships among objects. Unlike conventional block-based storage, which provides a simple *Logical Block Address* (LBA) interface, cloud storage presents an object-based abstraction. In object-based cloud storage [1], [5], [9], [13], the basic entity of user data is an *object*, which is associated with certain *metadata*. Objects are further organized into logical groups, called *buckets* or *containers*, forming a flat namespace.

Such an object-based storage model can carry much richer semantic information. A particularly useful knowledge is the relationship among objects. For example, when a Netflix movie trailer is accessed, the full movie is likely to be downloaded soon. Another example is compiling a programming project. The source code files have inherent logical dependencies: the related header files need to be read together with the main source code files. Therefore, compared to obtaining the relationships from the block layer [54], mining

the relationships among objects is more effective and much simpler. This opportunity will enable numerous unprecedented optimization opportunities for caching, prefetching, compression, scheduling, and many others.

• **Critical Issue 3: Accessing cloud storage objects may result in drastically different access costs.** For direct-attached storage, such as HDDs and SSDs, access costs (I/O latencies) vary in a relatively small range (at the level of milliseconds, or smaller) and can be accurately modeled [59], [60]. Comparatively, access costs for cloud storage are more variable. First, user-perceived cloud I/O latencies largely depend on object sizes and network conditions, such as the network speed and the geographic distance, and thus may vary significantly (from milliseconds to seconds). Second, parallelized accesses can significantly change the access cost. For example, our experiments show that downloading four small objects (e.g., 4 KB) in parallel demands almost identical time as downloading an individual one.

A strong implication to us is that we cannot continue to assume that the data access cost is a constant value. For many system schemes, such as object-based caching, we need to differentiate the miss penalties of different objects and design a cost-aware caching scheme. Unfortunately, many traditional caching schemes (e.g., [45], [47], [57], [62], [69]) are cost-unaware, and thus unsuitable for cloud storage; the classic cost-aware caching schemes, such as GreedyDual-Size (GDS) [25], are not designed for recognizing the change of access costs caused by cloud I/O parallelization.

The above-said issues have motivated us to revisit the existing system design for cloud storage. In particular, we focus on *caching* and *prefetching* on the client side (i.e., clients or client-side proxies/gateways) and present a cache management framework, called *Pacaca*, aiming to enhance user experience with cloud storage, especially reducing user-perceived access latencies.

Pacaca is a unified cache management framework integrating a set of optimization schemes designed particularly for cloud storage, including *object clustering*, *parallelized prefetching*, and *cost-aware caching*. Specifically, we first develop an efficient mining scheme to discover object correlations and then build a prefetching scheme to fetch correlated objects in parallel; we further develop a cost-aware caching scheme to differentiate high-cost and low-cost objects, leveraging the awareness of access cost changes caused by parallelized prefetching. Our contributions in this work are summarized as follows:

1) To accurately identify the most appropriate candidates for prefetching, we have designed an efficient mining scheme, called *Frequent Cluster Mining* (FCM), to discover object correlations, and group the correlated objects into clusters. FCM adopts a "black-box" approach, without relying on application-specific knowledge or data semantics, to fit the cloud environment.
2) To properly exploit the great parallelism potential of cloud storage, we have developed a parallelized prefetching scheme, which is closely coordinated with the mining scheme FCM for achieving high prefetching accuracy, proper control on parallelism degree, and effective misprefetching detection and handling.
3) To improve the caching efficiency, we have studied the impact of parallelized prefetching on the access costs of objects, and further developed a cost-aware caching scheme to leverage the awareness of object correlations and parallelized prefetching.
4) To evaluate the efficiency of client-side caching and prefetching, we have designed a cache management framework. Besides the caching scheme of Pacaca, this framework supports three traditional cache replacement policies, including LRU, ARC [57], and GreedyDual-Size (GDS) [25], which are integrated with our prefetching scheme. The experiments on Amazon S3 show that our optimization schemes can effectively reduce cloud I/O latencies, outperforming traditional schemes by up to 58%.

The rest of the paper is organized as follows. Section II and III present the mining and prefetching schemes. Section IV presents our caching scheme. Section V describes the cache management framework, which integrates all the schemes together. Section VI gives the evaluation results. Related work is presented in Section VII. Section VIII concludes the paper.

II. FCM: Frequent Cluster Mining

In this section, we present our mining scheme, called *Frequent Cluster Mining* (FCM), which is designed to obtain useful object correlations to provide guidance for effective prefetching on cloud storage. We first present the design goals, then describe the mining scheme in detail, and finally analyze the mining efficiency.

A. Design Goals

The purpose of cluster mining is to obtain object correlations to direct parallelized prefetching in the scenario of cloud storage. We have two main design goals.

First, the scheme should be *efficient and application-independent*. Cloud storage serves different applications and maintains a large number of highly diversified objects from various applications. Therefore, the mining scheme should not assume certain application-specific knowledge about data in cloud environments.

Second, the scheme should be *prefetching-focused*. For the purpose of prefetching, we are interested in object correlations that are accurate, stable, and up-to-date, and the correlated objects should be accessed in a small access distance. With the knowledge of such object correlations, we can proactively submit cloud I/Os to prefetch the correlated objects in parallel.

These two goals, unfortunately, cannot be properly satisfied by using conventional approaches. For example, the graph-based schemes [35], [37], [38], [50], [51], though effective for small data sets, are difficult to efficiently present the correlations involving many objects and could suffer scalability issues [54]. Some correlation mining schemes are application-specific. For example, Dependency Graphs used

in Web mining [61] assumes and relies on link dependencies among web pages. SEER [50] partially relies on file attributes to determine the importance of different files. Finally, the methods that are designed for other purposes, such as analyzing user behaviors [36] and making hoarding decisions [50], are not optimized for prefetching.

To achieve our design goals, we propose an efficient mining scheme, called *Frequent Cluster Mining* (FCM). FCM adopts a "black-box" approach and does not rely on assumptions of application-specific semantics or knowledge to fit the cloud environment. It considers the recency, frequency, and accuracy of object correlations, for the purpose of prefetching, and also utilizes a set of optimizations to improve the mining efficiency.

B. Mining Frequent Clusters

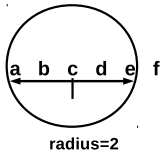


Fig. 1: An illustration of the look-around circle

The basic methodology of FCM is to mine frequent clusters from the access sequence. A *frequent cluster* is a set of objects that are frequently accessed together. In this section, we first introduce the constraints to determining a qualified cluster and then describe the mining procedures.

1) Determining Qualified Clusters: To determine a qualified cluster, FCM sets several constraints to quantitatively evaluate the strength of relationships among objects and to reduce the mining overhead.

Search scope: radius. To identify the correlated objects to a given object, we first define a search scope (i.e., the neighbor accesses in the access stream). We use a "look-around circle", whose size is determined by *radius* (see Figure 1). When setting *radius* to two, for example, FCM searches the correlated objects for a given object in the scope of the past two neighbor accesses and the following two neighbor accesses. The rationale is that the correlated objects for a given object may appear in the access sequence before or after that object. Some objects are correlated but do not necessarily have strict semantic dependencies, and thus they can be accessed in different orders. By restricting *radius*, FCM can discover the correlated objects that are closely accessed within a small access distance.

In general, we can set the search scope to a reasonably small value, such as 64 in C-Miner [54] or 20 in SEER [50]. In our design, since the search scope also determines the potential cluster size, we set *radius* to half of the upper bound of a proper cluster size, which is estimated by the parallelism control for prefetching (see Section III-A).

Search depth: search_limit. Another important factor is the recency of the identified correlations—a recent object correlation is more useful than an outdated one for prefetching. Thus, different from prior methods [50], [54], FCM particularly considers the recency of object correlations. For each object, when identifying its correlated objects, we use a threshold, *search_limit*, to restrict the maximum backward search distance (how far we look back in the access sequence). For example, if the *search_limit* is 100 for a given object, FCM only examines the past 100 accesses to the object and searches the potentially correlated objects in its look-around circles. This brings two benefits. First, it ensures the identified correlation clusters to be strong and up-to-date. Second, it limits the search depth and reduces the mining time. We will further discuss the effects of the *search_limit* threshold in Section VI-C2.

Metrics: support and confidence. To evaluate the accuracy of object correlations, we introduce the association rules and metrics, which are widely used in correlation mining [54], [65]. We use the association rule $x \rightarrow y$ to present the correlation that if object x is accessed, object y is likely to be accessed before or after the access to object x. We use two metrics, *confidence* and *support*, to estimate the accuracy and repeatability of an identified correlation. Specifically, if object x appears N times in the access sequence, and object y appears M times within the look-around circles of object x, then we have the association rule $x \rightarrow y$, and its $confidence = \frac{M}{N}$ and $support = M$. A high confidence means that two objects have a high possibility to be correlated, and a high support means that such a correlation is highly repeatable. So, we set the thresholds for both metrics, $min\_support$ and $min\_confidence$, to filter out weak and rare correlations. The effects of these two thresholds will be further discussed in Section VI-C2.

Cluster definition. By using *confidence* and *support* to measure object correlations, we can ensure the accuracy and repeatability of a cluster, in which any two objects are closely accessed and tightly correlated. Assuming a cluster c has a $min\_support$ threshold and a $min\_confidence$ threshold, it should satisfy the following two rules:

Rule 1: $\forall\ c_i \in c$ and $\forall\ c_j \in c$, the *confidences* of the association rules $c_i \rightarrow c_j$ and $c_j \rightarrow c_i$ are both no smaller than $min\_confidence$.

Rule 2: $\forall\ c_i \in c$ and $\forall\ c_j \in c$, the *supports* of the association rules $c_i \rightarrow c_j$ and $c_j \rightarrow c_i$ are both no smaller than $min\_support$.

2) Mining Procedures: FCM identifies clusters in three phases: (1) determining the search depth for frequent objects; (2) generating candidate association rules; and (3) generating final clusters from the obtained candidate association rules.

Phase 1: FCM scans the access sequence to count the frequency of each object. The purpose is to determine the proper search depth and to remove rarely accessed objects. First, if the frequency of an object is smaller than the default setting of *search_limit*, its *search_limit* threshold will be updated with its frequency. Any object with a frequency smaller than *min_support* is regarded as an infrequent object and will be discarded in the process of generating the candidate association rules.

Phase 2: FCM does a backward scan on the access sequence to generate the candidate association rules. Each time when object j appears in the look-around circle of object i, FCM increases the $support$ of the association rule $i \rightarrow j$ by 1; if the association rule does not exist, a new one is created. FCM only examines a limited number (*search_limit*) of recent accesses to the object. If the *search_limit* threshold of object i is reached, in the remaining process of the backward scanning, FCM will skip this object and not further update the association rules for it.

In addition to discarding infrequent objects (with the *min_support* threshold) and limiting the search depth (with the *search_limit* threshold), another technique for FCM to improve the mining efficiency is to early prune the "unpromising" objects and the association rules that are predicted to be impossible to satisfy the rules in the remaining process of searching.

To better explain this, we can consider a simple scenario, where FCM is searching for the correlated objects of object i and the $search\_limit$ threshold is 100. If the current $support$ of the association rule $i \rightarrow j$ is 9, and object i will be searched for 40 times in the remaining process, then the maximum possible $support$ of the association rule $i \rightarrow j$ will be 49, which is the sum of the current support (9) and the maximum possible increment of the $support$ value in the remaining process (40). Thus, the maximum possible $confidence$ is 0.49 (=49/100), which makes it impossible to satisfy the *min_confidence* threshold, 0.5. Therefore, object j is considered "unpromising" for object i, and we do not need to proceed further.

Phase 3: FCM generates clusters based on the cluster definition (see Section II-B1). The association rules that do not satisfy the definition are removed first. Then FCM scans each object to find potential clusters based on its association rules. For example, for an object a, which has the association rules $a \rightarrow b$ and $a \rightarrow c$, FCM checks the association rules in the descending order of their *confidence* values. For the association rule $a \rightarrow b$, if $b \rightarrow a$ also exists, the two objects are grouped as a cluster $\{ab\}$. Then, FCM continues to check $a \rightarrow c$. Object c can be added to cluster $\{ab\}$ if and only if the association rules $c \rightarrow a$, $c \rightarrow b$, and $b \rightarrow c$ also exist, which is based on the definition of a cluster that any two objects in a cluster should be correlated to each other. Once an object is added into a cluster, it is not further considered in the remaining process of clustering. This procedure is iterated over all the remaining objects until completion.

Discussion: Similar to prior work [54], FCM has time complexity $O(n)$. In practice, the efficiency of FCM is further optimized with several important measures. First, FCM only focuses on the correlations of frequent objects, rather than the semantic relationships of all the objects. Second, FCM particularly considers the *recency* by using the $search\_limit$ threshold to limit the search depth, which together with other thresholds constrains the number of candidate association rules for an object. Third, FCM prunes the "unpromising" objects and the association rules as early as possible.

III. Parallelized Prefetching

With the knowledge of object correlations, the basic prefetching scheme is straightforward: since any two objects in an identified cluster are tightly correlated (see Section II-B1), we can prefetch the correlated objects in parallel. Two main challenges are how to properly decide the parallelism degree and how to detect and handle mis-prefetching.

A. Parallelism Control

To reduce the interference between parallel requests, we propose a method to adjust the parallelism degree of prefetching by restricting the size of obtained clusters. The key idea is to determine the upper bound of a proper cluster size based on the system performance potential, so that downloading all the objects of a cluster in parallel would consume comparable time as downloading an individual object.

To achieve this, we need to understand the system capability, which can be characterized by the upper bound of the parallelism degree with which the client can download objects of a certain size without causing a latency increase. The knowledge about such parallelism degrees can be obtained by running simple tests on the client with a range of typical object sizes (e.g., from 16 KB to 1 MB) and parallelism degrees (e.g., from 1 to 64). For example, on our platform, such an upper-bound parallelism degree for downloading 64KB objects is observed to be 32, which means that downloading 64KB objects with a parallelism degree larger than 32 would lead to over-parallelization. Therefore, if the objects in a cluster are larger than 64 KB, the number of objects in the cluster should be no larger than 32, which is the upper bound of a proper cluster size.

To properly restrict the cluster size, we set the search scope *radius* in our mining scheme (see Section II-B1) to half of the upper bound of a proper cluster size, which ensures that the size of the obtained clusters would not be excessively large. With such a setting, fetching objects in parallel would not cause significant over-parallelization (see Section VI-C3 for further discussion).

B. Handling Mis-prefetching

To alleviate the possible cache pollution caused by mis-prefetched objects, we adopt a correlation-aware method to proactively detect mis-prefetching. We maintain a *logical clock*, which ticks upon each on-demand request. For each prefetched object, we set its *expiration time* as the current clock time plus the diameter (i.e., 2×*radius*) of the look-around circle of the cluster. If a prefetched object runs out of the assigned time and is still not accessed, it is considered as a mis-prefetched object. The rationale is that if an object fails to be accessed in the pre-defined access distance, it is very likely that this object is uncorrelated to the object accessed by the on-demand request. In this case, we should quickly evict such objects and reclaim their space (see Section VI-C3 for further discussion).

Step	Access Stream	LRU	Latency	Pacaca	Latency
1	A	[**A**]	7	[**A**]	7
2	B1	[**B1, B2, B3, B4**, A]	2	[A, {**B1, B2, B3, B4**}]	2
3	B2	[B2, B1, B3, B4, A]	0	[A, {B1, B2, B3, B4}]	0
4	B3	[B3, B2, B1, B4, A]	0	[A, {B1, B2, B3, B4}]	0
5	B4	[B4, B3, B2, B1, A]	0	[A, {B1, B2, B3, B4}]	0
6	C1	[**C1, C2, C3, C4**, B4, B3, B2, B1]	2	[A, {**C1, C2, C3, C4**}]	2
7	C2	[C2, C1, C3, C4, B4, B3, B2, B1]	0	[A, {C1, C2, C3, C4}]	0
8	C3	[C3, C2, C1, C4, B4, B3, B2, B1]	0	[A, {C1, C2, C3, C4}]	0
9	C4	[C4, C3, C2, C1, B4, B3, B2, B1]	0	[A, {C1, C2, C3, C4}]	0
10	B1	[B1, C4, C3, C2, C1, B4, B3, B2]	0	[A, {**B1, B2, B3, B4**}]	2
11	B2	[B2, B1, C4, C3, C2, C1, B4, B3]	0	[A, {B1, B2, B3, B4}]	0
12	B3	[B3, B2, B1, C4, C3, C2, C1, B4]	0	[A, {B1, B2, B3, B4}]	0
13	B4	[B4, B3, B2, B1, C4, C3, C2, C1]	0	[A, {B1, B2, B3, B4}]	0
14	A	[**A**, B4, B3, B2, B1]	7	[A, {B1, B2, B3, B4}]	0
Total Time			18		13

TABLE I: An example illustrating the advantages of the caching scheme of Pacaca over the traditional LRU caching scheme in the scenario of parallelized prefetching, in which all the objects of a cluster are downloaded in parallel upon related cache misses. In this example, the cache space is set to 16, and the cache is empty before Step 1. The objects shown in the cache from left to right have caching priorities from high to low. The objects of the lowest caching priority have the least "value" to be held in cache. The objects downloaded from the cloud are boldfaced. The sizes, downloading latencies, and costs of the objects and clusters are shown in Table II.

IV. Parallelization-aware Caching

In this section, we first analyze the impact of parallelized prefetching on caching with an illustrative example and then describe our cache replacement policy.

A. Impact of Parallelized Prefetching

Parallelized prefetching can change the relative costs of accessing objects from the cloud. Specifically, for the correlated objects that can be prefetched in parallel, the access cost is amortized, and thus the relative cost is lower than fetching each object individually. A direct implication to caching is that the relative cost of fetching an object in a cluster upon a cache miss would be significantly smaller (i.e., a lower miss penalty). This would change the equation for making a caching decision—evicting a low-cost object is a wise choice. Without such awareness, simply combining parallelized prefetching with traditional caching algorithms, such as LRU, ARC [57], and GreedyDual-Size (GDS) [25], would be sub-optimal.

B. An Illustrative Example

To illustrate the impact of parallelized prefetching on caching, we give a simple example in Table I to show the difference between the caching scheme of Pacaca and the traditional LRU caching scheme, which is widely adopted in current cloud-based storage systems [3], [4], [7], [8], [10], [12], [22], [66]. In the example, both schemes handle the same access stream in the scenario of parallelized prefetching (downloading all the objects of a cluster in parallel upon a related cache miss). Table II describes the sizes, latencies, and the access costs of the objects and clusters.

As shown in Table I, Pacaca has resulted in a lower aggregate latency than LRU (13 time units vs. 18 time units). Initially, the two caching algorithms have the same content. At Step 6, Pacaca and LRU begin to make distinct caching decisions. Since LRU makes the caching decisions only based on the recency of each object and finds that object A has a lower recency than other objects; consequently, LRU decides to evict object A. This decision leads to a cache miss of object A at a later time (Step 14), causing a high miss penalty (7 time units). By contrast, knowing that objects B1, B2, B3, and B4 are correlated and could be fetched in a cluster {B1, B2, B3, B4} in a parallelized manner, Pacaca estimates that the miss penalty of the cluster is lower than that of object A (0.25 cost unit vs. 0.875 cost unit). Thus, Pacaca decides to evict the cluster {B1, B2, B3, B4}, which leads to a relatively lower penalty (2 time units) at Step 10.

This example clearly illustrates the impact of parallelized prefetching on caching and demonstrates the importance of considering parallelism and object correlations when deciding the victim objects.

Object/Cluster	Size	Latency	Latency/Size
A	8	7	0.875
{B1, B2, B3, B4}	8	2	0.25
B1	2	2	1
B2	2	2	1
B3	2	2	1
B4	2	2	1
{C1, C2, C3, C4}	8	2	0.25
C1	2	2	1
C2	2	2	1
C3	2	2	1
C4	2	2	1

TABLE II: Access costs of the objects/clusters. {B1, B2, B3, B4} denotes the cluster containing objects B1, B2, B3, and B4; {C1, C2, C3, C4} denotes the cluster containing objects C1, C2, C3, and C4. The latency of a cluster is the time units of downloading the objects of the cluster in parallel. The cost of each object or cluster is calculated by $latency/size$.

C. Cache Replacement Policy

Pacaca adopts a cost-aware cache replacement algorithm based on GDS [25]. Our augmented algorithm is capable of

recognizing clusters of objects. The objects in a cluster are fetched together in parallel, when a related cache miss happens. We use a *cluster* as the basic unit for cost estimation. An object that does not have any correlated objects is considered as a special cluster containing a single object.

```
1  initialize L = 0
2  upon the request of object x
3  let c be the cluster containing x

4  if cache hit
5    H(c) = L + Lat(c)/Size(c)

6  if cache miss
7    while not enough cache space
8       update L = min(H)
9       evict cluster d such that H(d) = L
10   parallelized prefetching for cluster c
11   H(c) = L + Lat(c)/Size(c)
```

Fig. 2: Cache replacement algorithm

Figure 2 shows the algorithm of the caching scheme. Each cluster is associated with a value H to determine the caching priority (lines 5 and 8). The cluster with the lowest H value is selected as the victim and will be evicted first (lines 7-9). The H value is calculated as $H(c) = L + \frac{Lat(c)}{Size(c)}$, which includes two components:

- L is a global inflation value, tracking the H value of the most recently evicted cluster. Since the cluster having the lowest H value is always selected as the victim cluster (lines 7-9), L keeps growing and indicates the access recency of the clusters. Thus, a low L value means that the cluster has not been accessed recently.
- $\frac{Lat(c)}{Size(c)}$ evaluates the cost of the cluster, considering the miss penalty of the cluster per size unit. It incorporates the time of fetching the cluster in a parallelized way, $Lat(c)$, and the size of the cluster, $Size(c)$.

From this function, we can see that the cluster that has not been accessed for a long time and has a lower miss penalty is of less value for caching. Such a caching policy incorporates different factors, including not only access recency but also parallelization-aware miss penalty and cluster size.

It is worth noting that the latency function, $Lat(c)$, and the size function, $Size(c)$, here should only involve the objects that have been accessed on demand rather than the entire originally identified cluster. This is because some prefetched objects could be evicted earlier due to mis-prediction, or have not reached its *expiration time* and are waiting to be accessed (see Section III-B). Therefore, when calculating the cost of a cluster, we only consider the objects that have been accessed on demand. Similarly, when evicting a victim cluster, only the objects that have been accessed on demand will be evicted. The prefetched objects that are detected to expire will be evicted by the mis-prefetching handler (see Section III-B and Section V).

V. Putting It All Together

After describing each of the schemes above, we are in the position to present the architecture of *Pacaca*, which is a cache management framework that incorporates these schemes to provide client-side (i.e., clients or client-side gateways/proxies) caching and prefetching for cloud storage. Figure 3 shows the architecture of Pacaca, which integrates our proposed schemes. Since the details of the three schemes have been presented in prior sections, in this section we particularly focus on the integration of these components.

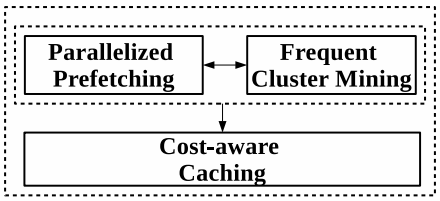


Fig. 3: An illustration of the architecture of Pacaca

Integration of cluster mining (FCM) and prefetching. In the framework, FCM is responsible for discovering object correlations, and the prefetching scheme exploits the correlations to make prefetching decisions. These two schemes are closely coordinated for achieving high prefetching accuracy, proper control on parallelism degree, and effective mis-prefetching detection and handling (see Section II and Section III).

Integration of caching and prefetching. To properly integrate the caching and prefetching schemes, an important issue is to manage the cache space for caching (to store the on-demand objects) and prefetching (to store the prefetched objects). In our design, we *logically* divide the local cache into two parts: a *demand cache* for caching on-demand data and a *prefetch cache* for holding prefetched data. These two areas use different management schemes: the prefetch cache manages objects in an LRU list, and the demand cache manages objects with our caching scheme (see Section IV-C).

Unlike prior methods that separate the cache space into two fixed-size partitions (e.g., [54]), in our design, caching and prefetching share the cache space. This is for two practical considerations. First, without static partitioning, the cache space can be sufficiently utilized, even when prefetching does not happen frequently. For example, a cache miss to an independent object that does not have correlated objects would not trigger prefetching at all. Second, our prefetching scheme is optimized with high prefetching accuracy and correlation-aware detection for handling mis-prefetching (see Section III-B). Thus we do not have to isolate the prefetched objects in a fixed-size area to reduce the cache pollution.

Cache space management. Figure 4 shows the core space management flow. Upon a request of object x, which is associated with cluster c (lines 1-2), for a cache hit in the *prefetch cache*, the object is promoted to the *demand cache* (lines 4-5); for a cache hit in the *demand cache*, the caching priority of the object is refreshed by the caching scheme (lines 6-7). If a cache miss happens (line 8), the prefetching scheme will be triggered to fetch the correlated objects in parallel

```
1  upon the request of object x
2  let c be the cluster containing x

3  if cache hit
4    if hit in prefetch cache
5       promote x to demand cache
6    if hit in demand cache
7       refresh caching priority

8  if cache miss
9    while not enough cache space
10      reclaim mis-prefetched objects
11   while not enough cache space
12      reclaim on-demand objects
13   while not enough cache space
14      reclaim prefetched objects in LRU order
15   parallelized prefetching for cluster c
16   add x to demand cache
17   add prefetched objects to prefetch cache
```

Fig. 4: Integration of caching and prefetching

(line 15), after which the on-demand object is added to the *demand cache* (line 16) and the other prefetched objects are added to the *prefetch cache* (line 17). If the cache space is not enough, the reclaiming priority of the objects from high to low is: (1) the mis-prefetched objects (lines 9-10); (2) the on-demand objects selected by the caching scheme (lines 11-12); and (3) the prefetched objects in the LRU order (lines 13-14). With such a policy, we first evict the mis-prefetched objects identified by the mis-prefetching handler and give a higher priority to protect the on-demand objects and the recently prefetched objects, which are likely to be accessed soon.

VI. Evaluation

A. Methodology

Emulation. We have implemented an emulator to evaluate the performance of our client-side cache management framework *Pacaca*. The prototype simulates a client for cloud storage similar to S3FS [12]. It provides POSIX-like APIs for users to access data stored on Amazon S3 buckets and has the support of a client cache to enable sophisticated caching and prefetching schemes. Particularly for dirty data, since a write-through policy would cause significant performance degradation [18], a write-back policy is often adopted by the cache solutions to optimize user-perceived performance with cloud storage [42], [66], [68]. In our prototype, we adopt a write-back policy similar to that of Linux memory management mechanism: we use a background daemon to synchronize the dirty objects to the cloud storage repository when they are *aged* (older than 30 seconds) or evicted.

Platform. In our experiments, we use Amazon S3 located in Oregon (s3-us-west-2.amazonaws.com) as the cloud. We also set up an Amazon EC2 instance (`m1.large`) in North California as the client to run our prototype. The client is configured with 2 processors, 7.5 GB memory, and 410 GB disk. The Round Trip Time between the client and the cloud is measured 28 milliseconds. The network bandwidth is tested to be 80 MB/sec.

Scheme comparisons. Besides the schemes of Pacaca, in our prototype, we have also implemented three classic caching algorithms: (1) LRU, a popular caching algorithm used in current cloud-based storage systems in both academia [12], [22], [66] and industry [3], [4], [7], [8], [10]; (2) ARC [57], one of the advanced caching algorithms that is recently adopted by gateway caching for cloud storage [68]; and (3) GreedyDual-Size (GDS) [25], a classic cost-aware caching algorithm.

We have integrated all the three caching algorithms with our parallelized prefetching schemes. With such comparisons, we can not only investigate the capability of our parallelized prefetching scheme to improve different caching algorithms but also evaluate the advantages of our caching scheme, which is aware of the parallelized prefetching. We note that *sequential prefetching* is a technique used by some cloud storage systems [11], [68]. However, it is only applicable to *block-based* cloud storage, in which the block sequentiality is visible to the block-level caching layer. For a cache serving object-based cloud storage, in which an object is the basic caching entity, sequential prefetching is not applicable due to the lack of object correlations. This has motivated us to develop the FCM mining scheme to discover object correlations. Therefore, we do not further use sequential prefetching as a comparison scheme.

Traces. Since web services and filesystem services are two typical and popular object-based storage services provided by cloud storage, we use the object-based traces converted from two web traces (Calgary and NASA) and two filesystem traces (Deasna2 and Home02) collected from the real-world storage systems:

- *Calgary* contains the logs of HTTP requests to the servers of Department of Computer Science of University of Calgary at Calgary, Canada [15], [17].
- *NASA* has 2-month HTTP requests to the web servers of NASA Kennedy Space Center in Florida [16], [17].
- *Deasna2* is an NFS trace of a general workload from the Department of Engineering and Applied Sciences at Harvard University. This trace is a mix of research, web, and email workloads [56], [63].
- *Home02* is another NFS trace collected in the main network of Harvard University, which serves 10,000 active user accounts from the colleges, the graduate school, and the administration [56], [64].

• **Trace pre-processing.** For the filesystem traces, we convert the NFS requests by extracting `file_id`, `offset`, and `length` from the `read` and `write` requests. For the web traces, we focus on `GET` and `PUT` requests (corresponding to `read` and `write` requests) and use the original link as `file_id`. Another pre-processing on the filesystem workloads is to split large files into smaller segments (1 MB). It simulates *chunking*, which is widely adopted in cloud storage clients (e.g., Dropbox) for various purposes such as deduplication, compression, delta encoding, and partial updating [24], [33]. Our framework leverages local storage as the client-side cache, and all the traces are first filtered with a memory cache. The memory cache has the size of 0.1% of the data set and uses the LRU cache replacement algorithm.

Training Trace	Length	Num. of Objects	Num. of Clusters	Avg. Cluster Size	Time (seconds)
Calgary (3 months)	218,519	5,133	549	4.7	4
NASA (1 month)	1,556,258	11,068	1,187	3.3	146
Deasna2 (1 day)	547,295	40,961	2,592	3.4	23
Home02 (1 day)	143,180	13,067	1,593	3.9	5

TABLE III: The details of the training traces and mining results. The cluster size refers to the number of objects in a cluster.

Testing Trace	Length	Num. of Objects
Calgary (3 months)	238,519	7,913
NASA (1 month)	1,305,596	10,093
Deasna2 (1 day)	488,145	36,532
Home02 (1 day)	135,363	12,559

TABLE IV: The details of the testing traces.

• **Trace splitting: training and testing.** To fairly test the effectiveness of identifying object correlations, we split each trace into two parts: one for training (denoted as *training trace*) and the other for testing (denoted as *testing trace*). For Calgary, we use the first three-month trace as the training trace and the next three-month trace as the testing trace; for NASA, the first-month and the second-month data are used for training and testing, respectively. For the filesystem traces, Deasna2 and Home02, we use one-day trace for training and the next-day trace for testing. The details of the testing traces are shown in Table IV. The details of the training traces and mining results are shown in Table III (see Section VI-B).

B. Mining Correlations in Real Traces

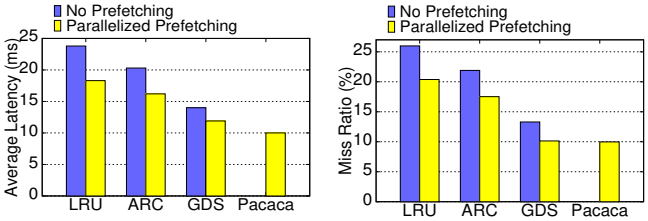
1) Parameter Settings: The main purpose of mining object correlations is for parallelized prefetching. When using FCM to cluster correlated objects, we set the *search_limit* as 10,000, $min\_confidence$ as 0.5, and $min\_support$ as 3. The setting of these thresholds can affect the performance of Pacaca, which will be further discussed in Section VI-C2.

As for the look-around circle, we need to set a relatively small *radius* to restrict the search scope for each object so that we can find correlated objects that are accessed within a small time frame; such object correlations are useful to direct prefetching (see Section II-B1). Specifically, in our experiments, we set the *radius* to 16 for mining web traces and 8 for mining filesystem traces. This setting considers the possible cluster size. Considering the 90th percentile of the object sizes in the web traces is smaller than 32 KB, based on the method of restricting the cluster sizes (see Section III-A), the proper number of objects for parallel accesses is tested to be 32 in our systems, so setting *radius* to no larger than 16 is a sound choice. For the same reason, we set the *radius* for the filesystem traces to 8.

2) Mining Results: The efficiency of mining object correlations determines its practicality. Table III shows the object clusters obtained from the training traces on our platform and the related overhead. Generally, the time overhead of FCM is reasonably low. For example, it takes only 4 seconds to complete searching for object correlations in Calgary. The overhead for mining filesystem traces is also small. For example, it takes only 5 seconds to mine object correlations in Home02. Even for the most costly one, NASA, it takes only 146 seconds to cluster correlated objects from the training trace which contains the log of one-month accesses.

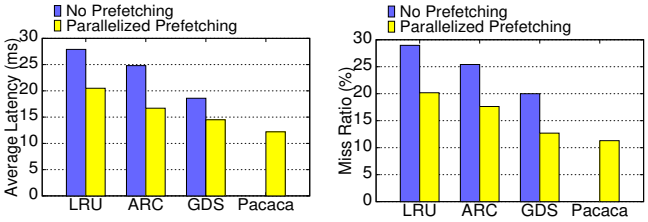
From Table III, we can also find that the average sizes of the object clusters are larger than 3, which means that the correlations involving multiple objects are abundant in real-system traces. This demonstrates the capability of FCM to find the correlations of multiple objects.

C. Performance Evaluation



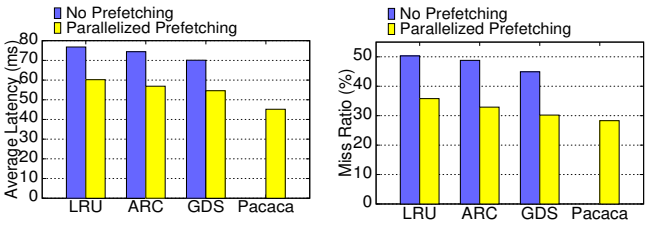
(a) Average Latency (b) Miss Ratio

Fig. 5: Performance for Calgary



(a) Average Latency (b) Miss Ratio

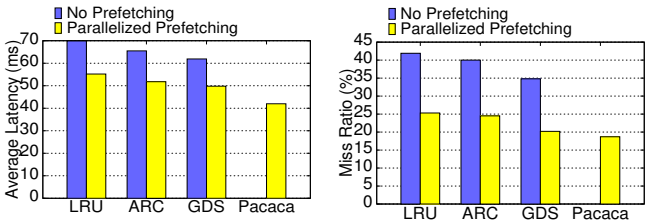
Fig. 6: Performance for NASA



(a) Average Latency (b) Miss Ratio

Fig. 7: Performance for Deasna2

1) Performance Comparison: In this experiment, we set the entire cache size as 5% of the working set (i.e., the total size of the unique objects). The prefetching scheme is directed by the object correlations obtained from the traces (see Section VI-B). Figures 5-8 show the performance for different optimization methods working with the four workloads. For a complete comparison, we also enhanced three traditional caching algorithms (LRU, ARC, and GDS) with the same prefetching scheme as Pacaca.



(a) Average Latency (b) Miss Ratio

Fig. 8: Performance for Home02

Overall performance improvement. Compared to the traditional caching algorithms without prefetching (LRU, ARC, and GDS), Pacaca can significantly improve the performance. Compared to LRU and ARC, Pacaca can reduce the average latencies by up to 58%. For example, for the Calgary workload, the average latency achieved by Pacaca is 10 milliseconds while the time used by LRU is 23.8 milliseconds, and by ARC is 20.3 milliseconds (see Figure 5). Even compared to GDS, Pacaca can also reduce the average latency by up to 35.5% with different workloads. This demonstrates the effectiveness and efficiency of Pacaca.

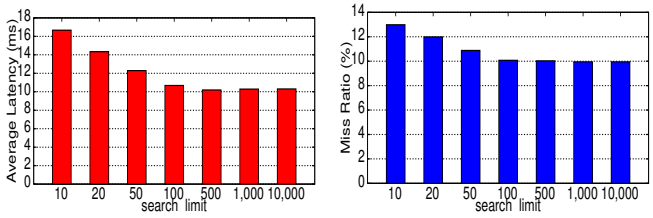
Efficiency of parallelized prefetching. In the figures, it is clear that the prefetching scheme can substantially improve all the traditional caching algorithms. For example, for the NASA workload, our prefetching scheme can reduce the average latency of LRU by 26.5%, ARC by 32.7%, and GDS by 22% (see Figure 6). This means that the obtained object clusters are effective for improving performance through parallelized prefetching. We also note that the mis-prefetching ratio (i.e., the percentage of the mis-prefetched objects of all the prefetched objects) is about 4%-15% with current settings (see Section VI-B1). The effects of mis-prefetching with different settings will be discussed in Section VI-C2.

Efficiency of cost-aware caching. Compared to the traditional caching algorithms that are enhanced with the same prefetching scheme, Pacaca can further improve system performance. Impressively, Pacaca can significantly outperform LRU with parallelized prefetching, reducing the average latencies by up to 45.4% (see Figure 5). Compared to ARC with the same prefetching scheme, Pacaca can reduce the average latencies by up to 38.3% with different workloads. Specifically, for example, for the Calgary workload, Pacaca outperforms ARC with parallelized prefetching by 38.3% (see Figure 5). The performance improvement of Pacaca is due to its cost-aware caching scheme. Both LRU and ARC are cost-unaware, thus they select the victim objects without considering the different miss penalties of the objects. By contrast, Pacaca makes caching decisions based on the access costs of the objects and prefers to evict the low-cost objects first, especially the correlated objects that can be prefetched in parallel, which leads to a lower average latency.

Compared to GDS with parallelized prefetching, Pacaca can reduce the average latencies by up to 17.2% with different workloads. Note that in this comparison, GDS is enhanced with our proposed prefetching scheme. If compared to GDS without prefetching, Pacaca can reduce the average latencies by up to 35.5% (see Figure 7). As stated in Section IV, the difference between the caching policies of Pacaca and GDS is that GDS only considers the access cost of each individual object, while Pacaca can further recognize the cost changes caused by parallelized prefetching. Therefore, the advantage of Pacaca over GDS with parallelized prefetching demonstrates the benefits of making caching decisions with the awareness of parallelism and object correlations.

This is consistent with our observation that Pacaca has similar miss ratios as GDS with parallelized prefetching but it achieves lower average latencies. For example, for the Calgary workload (see Figure 5), the miss ratios of Pacaca and GDS with parallelized prefetching are comparable (about 10%), but Pacaca can reduce the average latency by 16%. It is because evicting correlated objects, which have relatively lower access costs than individual objects, does not necessarily reduce miss ratios but can achieve lower overall miss penalties.

2) Sensitivity Study of Parameters: The performance of Pacaca can be affected by several parameters. In this section, we discuss the effects of three critical parameters: *search_limit*, *min_confidence*, and *min_support* (see Section II-B). Figures 9-11 show the performance achieved by Pacaca for the testing trace of Calgary with object correlations obtained from the training trace using different thresholds.

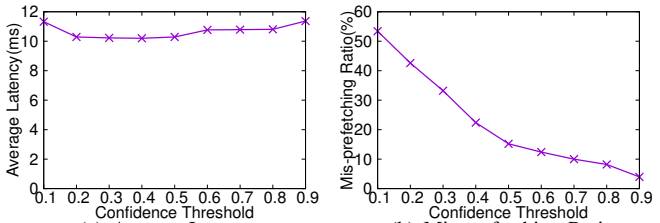


(a) Average Latency (b) Miss Ratio

Fig. 9: Pacaca performance for Calgary with different settings of *search_limit*.

Effects of the search_limit thresholds. Figure 9 shows the performance of Pacaca for Calgary using different *search_limit* thresholds from 10 to 10,000. When *search_limit* increases from 10 to 100, the performance is significantly improved, and after that, the performance gains diminish and the average latency even slightly increases when *search_limit* exceeds 500. This is because with a reasonable search scope, the object correlations become relatively stable, and further increasing the search depth cannot find more useful object correlations and could lead to performance loss. As for the mining overhead, reducing the *search_limit* from 10,000 to 100 results in 29% less time consumption. For the Calgary trace, using a *search_limit* of 100 can uncover 90% of the object occurrences. In our experiments, setting the *search_limit* close to the 90th percentile of the object occurrences also works well for other traces.

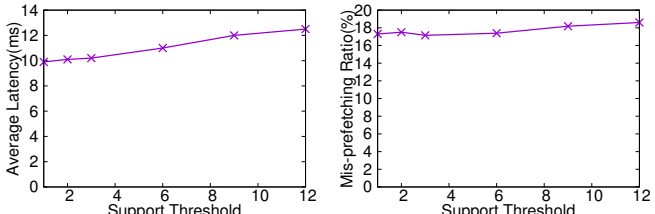
Effects of the min_confidence thresholds. Figure 10 shows the average latencies and mis-prefetching ratios achieved



(a) Average Latency (b) Mis-prefetching Ratio

Fig. 10: Pacaca performance for Calgary with different settings of *min_confidence*.

by Pacaca for Calgary using the confidence thresholds (*min_confidence*) ranging from 0.1 to 0.9. When the confidence threshold increases from 0.1 to 0.2, the average latency decreases by 8.8%; after that, the performance remains stable; when the confidence threshold exceeds 0.5, the average latency increases slightly. This can be explained from two aspects. First, a lower confidence threshold is helpful to find more object correlations but may suffer a higher mis-prefetching ratio. When the confidence threshold increases from 0.1 to 0.9, shown as Figure 10(b), the mis-prefetching ratio decreases from 53% to 4%, demonstrating that prefetching accuracy is determined by the confidence threshold. Since Pacaca can evict the mis-prefetched objects as early as possible and fetch the objects with proper parallelization (see Section III and Section VI-C3), the negative effect of mis-prefetching on performance is largely mitigated. However, considering intensive mis-prefetching would waste the system resources (e.g., network bandwidth), we find that a relatively higher confidence threshold (e.g., between 0.5 and 0.8) is more desirable.

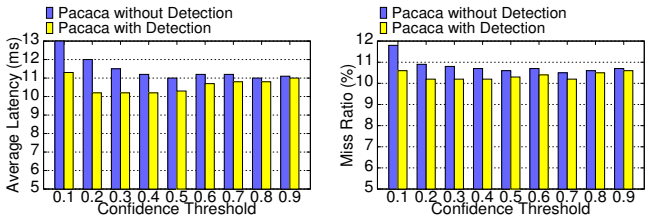


(a) Average Latency (b) Mis-prefetching Ratio

Fig. 11: Pacaca performance for Calgary with different settings of *min_support*.

Effects of the min_support thresholds. Figure 11 shows the average latencies and mis-prefetching ratios achieved by Pacaca for Calgary using the support thresholds (*min_support*) ranging from 1 to 12. When the support thresholds are smaller than 3, the average latencies are comparable. After that, the performance degradation is significant as the support threshold increases. Although a low support threshold may introduce some object correlations with weak repeatability, a high support threshold would filter out a lot of useful object correlations and thus lose many opportunities for prefetching. As shown in Figure 11(b), the support thresholds do not have obvious effects on mis-prefetching ratios. That is because the prefetching accuracy is mainly determined by the confidence threshold. Therefore, setting the support threshold to a reasonably small value (e.g., 3 in our experiments) is generally appropriate for performance.

3) Impact of Optimization Methods: In this section, we evaluate the impact of optimization methods, including the correlation-aware mis-prefetching detection and restricting cluster sizes for parallelism control.



(a) Average Latency (b) Miss Ratio

Fig. 12: Pacaca performance for Calgary with and without the correlation-aware mis-prefetching detection.

Impact of correlation-aware mis-prefetching detection. To reduce the possible cache pollution, we adopt a correlation-aware approach to detect the mis-prefetched objects, and quickly reclaim their space for efficiently utilizing the cache space (see Section III-B and Section V). Figure 12 shows the performance for the Calgary workload with and without the correlation-aware detection for handling mis-prefetching, setting the confidence thresholds ranging from 0.1 to 0.9. From Figure 12(a), we can see that the correlation-aware approach can reduce the average latency by up to 13.1%. This is because the approach can quickly remove the mis-prefetched objects to reclaim their space, resulting in the reduced miss ratios, as shown in Figure 12(b). In addition, since a higher confidence threshold leads to a lower mis-prefetching ratio (see Section VI-C2), the impact of the mis-prefetching detection diminishes as the confidence threshold increases.

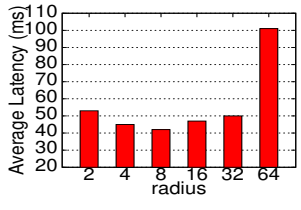


Fig. 13: Average Latency vs. Radius

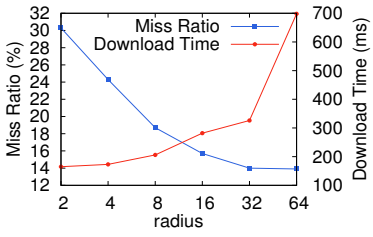


Fig. 14: Miss ratio and Download Time vs. Radius

Impact of restricting cluster sizes for parallelism control. To avoid over-parallelization, we set the search scope (*radius*) to restrict the cluster sizes for prefetching (see Section III-A). Based on the performance capabilities of the client, we set the *radius* to search for object clusters from the filesystem

traces and the web traces to 8 and 16, respectively (see Section VI-B1). Figure 13 shows the performance for the Home02 workload with different search scopes. Our setting (*radius* set to 8) achieves the lowest average latency.

For setting a smaller *radius* (2 or 4), the performance loss is caused by a higher miss ratio (see Figure 14). That is because a smaller search scope would limit the obtained object clusters in a smaller size, resulting in a smaller prefetching granularity and lower prefetching efficiency. For setting a higher *radius* (16, 32, or 64), the miss ratio decreases due to a larger prefetching granularity; however, the latency increase caused by the interference among over-parallelized requests can offset the benefits of a lower miss ratio, resulting in performance degradation. In particular, when setting *radius* to 64, we find that 8.4% of the obtained clusters contain more than 16 objects; 22.3% of these large clusters have even more than 32 objects. Downloading such large clusters in parallel would cause significant over-parallelization, impairing system performance, as shown in Figure 13 and Figure 14.

As for the web traces, since the object sizes are relatively small, the negative effect of over-parallelization is less severe. However, increasing the current *radius* setting from 16 to 32, for example, with the Calgary trace, can still lead to latency increase by about 10%.

VII. Other Related Work

In this section, we present other related work that has not been discussed previously. Several prior measurement work on cloud storage focuses on investigating the performance and behaviors of cloud storage services [21], [28], [52], [58] and client applications [20], [31], [32], [33], [42], [43], [55], [67]. These prior studies lay a foundation for us to understand cloud storage services. Our work particularly focuses on exploiting I/O parallelism, a unique characteristic of cloud storage.

Caching is widely adopted in cloud environments [19], [27], [44]. This work particularly focuses on the client-side caching and prefetching for cloud storage. LRU is a popular caching algorithm adopted in cloud-based storage systems [12], [22], [66] and also being used in many commercial products [3], [4], [7], [8], [10]. Tombolo [68] implements a sequential prefetching scheme integrated with the SARC [23] cache replacement algorithm in a gateway simulator. This scheme provides block-based optimizations and is thus not applicable in object-based cloud storage. Our efforts include clustering semantically correlated objects, prefetching objects in a properly parallelized manner, and making cost-aware caching decisions, which also make our work different from other approaches of integrating caching and prefetching [14], [29], [34], [39], [40], [49].

Our caching scheme is a type of GDS-based cost-aware caching optimized for parallelized prefetching in cloud storage. GreedyDual-Size (GDS) [25] is originally designed for web caching, which considers locality with cost and size in cache replacement. Other prior GDS-based caching algorithms [26], [46], [48], [53] also introduce additional factors, such as power and spatial locality. Our algorithm particularly recognizes the change of access costs caused by I/O parallelization in cloud storage, and it makes caching decisions by leveraging the awareness of parallelism and object correlations.

Our prior work, GDS-LC [41], is another client caching scheme designed for cloud storage. It aims to improve the cost efficiency by considering the difference in latencies and monetary costs of various cloud storage I/Os in caching. By contrast, Pacaca focuses on exploiting the parallelism potential in cloud storage by mining the relationships among objects and prefetching correlated objects in a parallelized manner, and its caching scheme is accordingly designed and optimized to be aware of such parallelization. As a flexible client cache management framework, Pacaca can include GDS-LC and other caching schemes, such as ARC and GDS, well integrating them with our parallelized prefetching scheme.

VIII. Conclusions

In this paper, we present a client-side cache management framework, called *Pacaca*, to optimize user experience with cloud storage. In this framework, we first design a cluster-based mining scheme *FCM* to obtain object correlations, based on which an optimized prefetching scheme preloads correlated objects in parallel. A cost-aware caching scheme further leverages the awareness of parallelism and object correlations to optimize the local cache management. The experimental results show the effectiveness and efficiency of our proposed schemes, which also demonstrates that it is important to consider the unique characteristics of cloud storage, such as parallelism potential and object correlations, to achieve the desired optimization goals.

Acknowledgments

We thank the anonymous reviewers for their constructive feedback and insightful comments. This work was supported in part by Louisiana Board of Regents under Grant LEQSF(2014-17)-RD-A-01 and in part by the U.S. National Science Foundation under Grants CCF-1453705 and CCF-1629291.

References

[1] Amazon S3. https://aws.amazon.com/s3/.
[2] Amazon S3 - Two Trillion Objects, 1.1 Million Requests / Second. https://aws.amazon.com/blogs/aws/amazon-s3-two-trillion-objects-11-million-requests-second/.
[3] EMC Elastic Cloud Storage (ECS) Overview and Architecture. https://www.emc.com/collateral/white-papers/h14071-ecs-architectural-guide-wp.pdf.
[4] ESG Microsoft Azure StorSimple White paper. http://www6.nasuni.com/rs/445-ZDB-645/images/CacheConfig.pdf.
[5] Google Cloud Storage. https://cloud.google.com/storage/.
[6] Microsoft Azure Blob Storage Handles Millions of Average Requests per Second. https://azure.microsoft.com/en-us/services/storage/blobs/.
[7] Nasuni Cache Configuration. http://www6.nasuni.com/rs/445-ZDB-645/images/CacheConfig.pdf.
[8] NetApp SteelStore Cloud Integrated Storage 3.2 Deployment Guide. https://library.netapp.com/ecm/ecm_download_file/ ECMP12031272.
[9] OpenStack Swift. http://www.openstack.org/.
[10] Panzura Debuts Version 3.0 of Its Global Cloud Storage System. http://panzura.com/press-releases/panzura-debuts-version-3-0-of-its-global-cloud-storage-system/.
[11] S3Backer. https://code.google.com/p/s3backer/.

[12] S3FS. https://code.google.com/p/s3fs/.

[13] Windows Azure Blob Storage. https://azure.microsoft.com/en-us/services/storage/blobs/.

[14] S. Albers and M. Büttner. Integrated Prefetching and Caching in Single and Parallel Disk Systems. In *Proceedings of the 15th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA'03)*, pages 109–117, San Diego, CA, June 7-9 2003.

[15] I. T. Archive. Calgary-HTTP. http://ita.ee.lbl.gov/html/contrib/Calgary-HTTP.html.

[16] I. T. Archive. NASA-HTTP. http://ita.ee.lbl.gov/html/contrib/NASA-HTTP.html.

[17] M. F. Arlitt and C. L. Williamson. Web Server Workload Characterization: The Search for Invariants. *ACM SIGMETRICS Performance Evaluation Review*, 24(1):126–137, 1996.

[18] D. Arteaga and M. Zhao. Client-side Flash Caching for Cloud Systems. In *Proceedings of International Conference on Systems and Storage (SYSTOR'14)*, pages 1–11, Haifa, Israel, June 30-July 2 2014. ACM.

[19] 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 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*, pages 47–60, 2010.

[20] A. Bergen, Y. Coady, and R. McGeer. Client Bandwidth: The Forgotten Metric of Online Storage Providers. In *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim'11)*, Victoria, BC, Canada, August 23-26 2011.

[21] I. Bermudez, S. Traverso, M. Mellia, and M. Munafo. Exploring the Cloud from Passive Measurement: The Amazon AWS Case. In *Proceedings of The 32nd IEEE International Conference on Computer Communications (INFOCOMM'13)*, Turin, Italy, April 14-19 2013.

[22] Bessani, Alysson, R. Mendes, T. Oliveira, N. Neves, M. Correia, M. Pasin, and P. Verissimo. SCFS: A Shared Cloud-backed File System. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*, Philadelphia, PA, June 19-20 2014.

[23] Binny S. Gill and Dharmendra S. Modha. SARC: Sequential Prefetching in Adaptive Replacement Cache. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05)*, pages 293–308, Anaheim, CA, April 10-15 2005.

[24] E. Bocchi, I. Drago, and M. Mellia. Personal Cloud Storage Benchmarks and Comparison. *IEEE Transactions on Cloud Computing*, 99:1–14, 2015.

[25] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and System (USITS'97)*, Monterey, CA, December 8-11 1997.

[26] F. Chen, S. Jiang, and X. Zhang. SmartSaver: Turning Flash Drive into a Disk Energy Saver for Mobile Computers. In *Proceedings of 2006 International Symposium on Low Power Electronics and Design (ISLPED'06)*, Tegernsee, Germany, October 4-6 2006.

[27] F. Chen, M. Mesnier, and S. Hahn. Client-aware Cloud Storage. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*, Santa Clara, CA, June 2-6 2014.

[28] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and V. colleagues. Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC'10)*, Indianapolis, IN, June 10–11 2010.

[29] M. S. Day. Client Cache Management in a Distributed Object Database. Technical report, Cambridge, MA, USA, 1995.

[30] X. Ding, S. Jiang, F. Chen, K. Davis, and X. Zhang. DiskSeen: Exploiting Disk Layout and Access History to Enhance I/O Prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC'07)*, Santa Clara, CA, June 17-22 2007.

[31] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking Personal Cloud Storage. In *Proceedings of the 2013 ACM Conference on Internet Measurement Conference (IMC'13)*, Barcelona, Spain, October 23-25 2013.

[32] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Modeling the Dropbox Client Behavior. In *Proceedings of the 2014 IEEE International Conference on Communicationsconference (ICC'14)*, Sydney, Australia, June 10-14 2014.

[33] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 Internet Measurement Conference (IMC'12)*, New York, NY, November 14-16 2012.

[34] J. Dwiartanto and P. Watson. Exploiting Method Semantics in Client Cache Consistency Protocols for Object-Oriented Databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Engineering (IKE'04)*, pages 467–473, Las Vegas, Nevada, June 21-24 2004.

[35] P. R. Eaton, D. Geels, and G. Mori. Clump: Improving File System Performance Through Adaptive Optimizations. *http://citeseerx.ist.psu.edu/viewdoc/download? doi=10.1.1.23.227*, 1999.

[36] M. Eirinaki and M. Vazirgiannis. Web Mining for Web Personalization. *ACM Transactions on Internet Technology*, 3(1):1–27, 2003.

[37] J. Griffioen and R. Appleton. Reducing File System Latency using a Predictive Approach. In *Proceedings of the USENIX Summer 1994 Technical Conference*, pages 197–207, Boston, Massachusetts, June 6-10 1994.

[38] J. Griffioen and R. Appleton. Performance Measurements of Automatic Prefetching. In *Proceedings of International Conference on Parallel and Distributed Computing Systems (PDCS'95)*, pages 165–170, 1995.

[39] T. W. Guang, C. C. Yue, and C. M. Syan. Integrating Web Caching and Web Prefetching in Client-side Proxies. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):444–455, 2005.

[40] W. S. Han, K. Y. Whang, and Y. S. Moon. A Formal Framework for Prefetching Based on the Type-Level Access Pattern in Object-Relational DBMSs. *IEEE Transactions on Knowledge and Data Engineering*, 17:1436–1448, 2005.

[41] B. Hou and F. Chen. GDS-LC: A Latency- and Cost-Aware Client Caching Scheme for Cloud Storage. In *ACM Transactions on Storage*, volume 13(4), pages 40:1–40:33, November 2017.

[42] B. Hou, F. Chen, Z. Ou, R. Wang, and M. Mesnier. Understanding I/O Performance Behaviors of Cloud Storage from a Client's Perspective. In *Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST'16)*, Santa Clara, CA, May 2-6 2016.

[43] W. Hu, T. Yang, and J. N. Matthews. The Good, the Bad and the Ugly of Consumer Cloud Storage. *ACM SIGOPS Operating Systems Review*, 44(3):110–115, Aug. 2010.

[44] 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 24th ACM Symposium on Operating Systems Principles (SOSP'13)*, pages 167–181, November 3-6 2013.

[45] S. Jiang, F. Chen, and X. Zhang. CLOCK-Pro: An Effective Improvement of the CLOCK Replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05)*, Anaheim, CA, April 10-15 2005.

[46] S. Jiang, X. Ding, F. Chen, E. Tan, and X. Zhang. DULO: An Effective Buffer Cache Management Scheme to Exploit Both Temporal and Spatial Localities. In *Proceedings of the 4th USENIX Conference on File and Storage Technologies (FAST'05)*, San Francisco, CA, December 12-16 2005.

[47] S. Jiang and X. Zhang. LIRS: An Efficient Low Inter-reference Recency Set Replacement Policy to Improve Buffer Cache Performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02)*, Marina Del Rey, CA, June 15-19 2002.

[48] S. Jin and A. Bestavros. Popularity-aware GreedyDual-Size Web Proxy Caching Algorithms. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS'00)*, pages 254–261, April 10-13 2000.

[49] T. Kimbrel, P. Cao, E. W. Felten, A. R. Karlin, and K. Li. Integrated Parallel Prefetching and Caching. In *ACM SIGMETRICS Performance Evaluation Review*, volume 24, pages 262–263. ACM, 1996.

[50] G. H. Kuenning. Design of the SEER Predictive Caching Scheme. In *Workshop on Mobile Computing Systems and Applications (WMCSA'94)*, December 8-9 1994.

[51] G. H. Kuenning and G. J. Popek. Automated Hoarding for Mobile Computers. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP'97)*, pages 264–275, Saint Malo, France, October 5-8 1997.

[52] A. Li, X. Yang, and M. Zhang. CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10)*, Melbourne, Australia, November 1–3 2010.

[53] C. Li and A. L. Cox. GD-Wheel: A Cost-aware Replacement Policy for Key-value Stores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15)*, Bordeaux, France, April 21-24 2015.

[54] Z. Li, Z. Chen, and Y. Zhou. Mining Block Correlations to Improve Storage Performance. *ACM Transactions on Storage*, 1(2):213–245, 2005.

[55] T. Mager, E. Biersack, and P. Michiardi. A Measurement Study of the Wuala On-line Storage Service. In *Proceedings of the IEEE 12th International Conference on Peer-to-Peer Computing (P2P'12)*, Sophia Antipolis, France, September 3-5 2012.

[56] P. Malkani, D. Ellard, J. Ledlie, and M. Seltzer. Passive NFS Tracing of Email and Research Workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*, pages 203–216, Berkeley, CA, March 31-April 2 2003.

[57] N. Megiddo and D. S. Modha. Outperforming LRU with an Adaptive Replacement Cache Algorithm. *IEEE Computer Magazine*, 37(4):58–65, April 2004.

[58] X. Meng, Y. Chen, J. Xu, and J. Lu. Benchmarking Cloud-based Data Management Systems. In *Proceedings of the 2nd international workshop on Cloud data management (CloudDB'10)*, Toronto, ON, October 26–30 2010.

[59] M. Mesnier, B. Salmon, M. Wachs, and G. Ganger. Relative Fitness Models for Storage. *ACM SIGMETRICS Performance Evaluation Review (PER)*, 33(4):23–28, June 2006.

[60] M. Mesnier, M. Wachs, R. R. Sambasivan, A. Zheng, and G. R. Ganger. Modeling the Relative Fitness of Storage. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'07)*, San Diego, CA, June 12-16 2007.

[61] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A Data Mining Algorithm for Generalized Web Prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, 2003.

[62] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-Based Replacement. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'90)*, Boulder, CO, May 22-25 1990.

[63] SNIA. Deasna2. http://iotta.snia.org/traces/102.

[64] SNIA. Home02. http://iotta.snia.org/traces/35.

[65] R. Srikant and R. Agrawal. Mining Quantitative Association Rules in Large Relational Tables. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD'96)*, pages 1–12, June 4-6 1996.

[66] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12)*, San Jose, CA, February 14-17 2012.

[67] H. Wang, R. Shea, F. Wang, and J. Liu. On the Impact of Virtualization on Dropbox-like Cloud File Storage/Synchronization Services. In *Proceedings of International Workshop on Quality of Service (IWQoS'12)*, Coimbra, Portugal, June 4-5 2012.

[68] S. Yang, K. Srinivasan, K. Udayashankar, S. Krishnan, J. Feng, Y. Zhang, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Tombolo: Performance Enhancements for Cloud Storage Gateways. In *Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST'16)*, Santa Clara, CA, May 2-6 2016.

[69] Y. Zhou, J. F. Philbin, and K. Li. The Multi-Queue Replacement Algorithm for Second Level Buffer Caches. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC'01)*, Boston, MA, June 25-30 2001.