GDS-LC: A Latency- and Cost-Aware Client Caching Scheme for Cloud Storage

BINBING HOU and FENG CHEN, Louisiana State University

Successfully integrating cloud storage as a primary storage layer in the I/O stack is highly challenging. This is essentially due to two inherent critical issues: the high and variant cloud I/O latency and the per-I/O pricing model of cloud storage. To minimize the associated latency and monetary cost with cloud I/Os, caching is a crucial technology, as it directly influences how frequently the client has to communicate with the cloud. Unfortunately, current cloud caching schemes are mostly designed to optimize miss reduction as the sole objective and only focus on improving system performance while ignoring the fact that various cache misses could have completely distinct effects in terms of latency and monetary cost.

In this article, we present a cost-aware caching scheme, called *GDS-LC*, which is highly optimized for cloud storage caching. Different from traditional caching schemes that merely focus on improving cache hit ratios and the classic cost-aware schemes that can only achieve a single optimization target, GDS-LC offers a comprehensive cache design by considering not only the access locality but also the object size, associated latency, and price, aiming at enhancing the user experience with cloud storage from two aspects: access latency and monetary cost. To achieve this, GDS-LC virtually partitions the cache space into two regions: a high-priority latency-aware region and a low-priority price-aware region. Each region is managed by a cost-aware caching scheme, which is based on GreedyDual-Size (GDS) and designed for a cloud storage scenario by adopting clean-dirty differentiation and latency normalization. The GDS-LC framework is highly flexible, and we present a further enhanced algorithm, called *GDS-LCF*, by incorporating access frequency in caching decisions. We have built a prototype to emulate a typical cloud client cache and evaluate GDS-LC and GDS-LCF with Amazon Simple Storage Services (S3) in three different scenarios: local cloud, Internet cloud, and heterogeneous cloud. Our experimental results show that our caching schemes can effectively achieve both optimization goals: low access latency and low monetary cost. It is our hope that this work can inspire the community to reconsider the cache design in the cloud environment, especially for the purpose of integrating cloud storage into the current storage stack as a primary layer.

CCS Concepts: • **General and reference** → **Design**; **Performance**; • **Information systems** → **Cloud based storage**; *Distributed storage*; *Hierarchical storage management*; • **Computer systems organization** → *Cloud computing*;

Additional Key Words and Phrases: Cloud storage, storage systems, caching algorithms

ACM Reference format:
Binbing Hou and Feng Chen. 2017. GDS-LC: A Latency- and Cost-Aware Client Caching Scheme for Cloud Storage. *ACM Trans. Storage* 13, 4, Article 40 (November 2017), 33 pages.
https://doi.org/10.1145/3149374

This work was supported in part by the Louisiana Board of Regents under grant LEQSF(2014-17)-RD-A-01 and the National Science Foundation under grants CCF-1453705 and CCF-1629291.

Authors' addresses: B. Hou and F. Chen (Corresponding author), Louisiana State University, Department of Computer Science and Engineering, Baton Rouge, LA, 70803, USA; emails: {bhou, fchen}@csc.lsu.edu.

1 INTRODUCTION

Cloud storage, such as Amazon Simple Storage Services (S3) and Dropbox, is becoming increasingly popular. As a cloud-based service, cloud storage allows users to store data in the service provider's data centers and access data via the network using an HTTP-based protocol. This new type of storage model provides a platform-independent storage abstraction and enables a set of highly desirable technical merits to end users, such as efficiency, elasticity, and flexibility. Despite these benefits, integrating cloud storage truly as a primary storage layer in computer systems still remains highly challenging. This is due to two critical issues inherent in cloud storage systems.

First, cloud I/Os suffer high and variant latencies. An essential weakness of cloud storage is the slow, unstable, and sometimes expensive network connection between the client and the cloud. If we consider a coast-to-coast connection, dozens of network components could be involved, and each contributes a nontrivial delay. As a cloud I/O may travel an excessively long distance (e.g., thousands of miles) to the service provider's data center, a cloud storage I/O latency could be hundreds of milliseconds or even higher. In contrast, local storage, even spinning disks, can easily complete an I/O in several milliseconds. Such a 100x longer I/O latency is intolerably slow for interactive operations. Even worse, as the latency is highly dependent on the client-to-cloud distance, it could be highly variant depending on where the data are stored and where the data are accessed. In heterogeneous cloud systems where data are stored in multiple distant data centers, such a situation is even more complex. Therefore, directly using cloud storage as a primary storage without proper optimization could incur high I/O latencies and cannot provide a satisfactory user experience for practical use.

Second, cloud storage adopts an unconventional pricing model, which is usage based. The pricing of cloud storage is normally a combination of three components: storage cost, which is based on the amount of data stored in the cloud; request cost, which is based on the number of I/O requests (e.g., `GET` and `PUT`) issued to the cloud; and data transfer cost, which is based on the volume of actual data transfer out from the cloud. With such a pricing model, each cloud storage I/O is associated with a certain amount of monetary cost, and thus the user's I/O activities would directly impact the operation cost. This is completely different from conventional storage, which is typically priced based on capacity and only involves a one-time expense for the initial installation. As such, the monetary cost of I/Os during runtime is not an issue with conventional storage but a must-have consideration with cloud storage. Without appropriate optimization, naively using cloud storage as a primary storage may incur undesirable economic loss.

Caching is a classic technique to address the two issues stated previously. By using local storage to temporarily reserve a copy of the most "valuable" data, most I/O requests can be served locally, so we can effectively reduce the I/O requests issued to the cloud and consequently lower both the access latency and monetary cost for using cloud storage services. Unfortunately, current caching schemes are suboptimal in the cloud environment. Despite being widely adopted in cloud-based storage systems (e.g., BlueSky [71] and S3FS [63]), conventional caching schemes, such as least recently used (LRU), can only exploit the access pattern (e.g., temporal locality) of the workloads and do not have the capability of differentiating the miss penalties associated with different objects. Cost-aware caching schemes, such as GreedyDual-Size (GDS) [14], are able to make caching decisions based on both temporal locality and other factors, including object size and access cost. However, they can only focus on minimizing one target (generally access latency) and thus cannot satisfy the requirements of minimizing both access latency and monetary cost, requiring further optimization in the cloud environment.

In this article, we present a caching scheme for cloud storage, called *GDS-LC*, aiming to enhance the user experience with cloud storage from two aspects: access latency and monetary cost. The key idea of GDS-LC is to label each cloud storage object by its value, in terms of the access locality, object size, retrieving latency, and monetary cost from the cloud, and offer high priority to protect the high-value objects in the client cache while aggressively evicting the low-value objects (i.e., the objects accessing that incurs relatively low latency and cost). To achieve this, GDS-LC virtually partitions the cache space into two regions: a high-priority latency-aware region, and a low-priority price-aware region. Each region is managed by a cost-aware caching algorithm, which is based on GDS [14] and designed for the cloud storage scenario by adopting clean-dirty differentiation and latency normalization. The objects of high locality and high value in terms of latency and price are identified for being kept in the cache, which allows us to reshape the cloud I/O streams to the desired low-latency and low-cost pattern. With such a two-region design, GDS-LC well balances several key factors for cloud storage caching, namely locality, size, latency, and price, which helps improve overall system performance and cost. Our solution can also be flexibly extended by considering other factors for caching. For example, by incorporating *frequency* in the caching decision, we further present a scheme called *GDS-LCF*, which gives a relatively high caching priority to frequently accessed objects.

To evaluate the effectiveness and efficiency of the caching schemes, we have built a prototype to emulate a typical cloud client cache. We choose Amazon Simple Storage Services (S3), one of the most popular cloud storage service providers, as the target cloud. Considering the diversity of the use cases of cloud storage, we set up three different working scenarios: local cloud, Internet cloud, and heterogeneous cloud, which feature different access latencies and pricing models. The experimental results show that compared to traditional caching schemes that solely focus on locality and the classic cost-aware caching schemes that can only achieve a single optimization target, our caching schemes can successfully achieve both optimization goals: low access latency and low monetary cost. It is our hope that this work can inspire the community to reconsider the cache design in the cloud environment, especially for the purpose of integrating cloud storage into the current storage stack as a primary layer.

The rest of the article is organized as follows. Section 2 gives the background. Section 3 analyzes the challenges of making an effective caching design in the cloud environment. Section 4 describes the design of our caching schemes. Section 5 gives the experimental evaluation. Section 6 presents other related work. Section 6 concludes this article.

2 BACKGROUND

In this section, we briefly introduce the two components of cloud storage—*cloud storage* services and *client applications*—and also describe the classic cost-aware caching scheme GDS [14].

2.1 Cloud Storage Services

Cloud storage can provide efficient object-based storage services. The basic data unit is an *object*, which is conceptually similar to a file in file systems. An object can carry certain metadata in the form of key-value pairs. Objects are generally organized into logical groups, called *bucket* or *container*. A bucket is akin to a directory in file systems, but buckets cannot be nested, meaning that the namespace is a flat one-level structure. To provide a high guarantee of reliability and availability, the objects can be further distributed to different geographic regions. With such an organization, an object can be referred to via a URL, similar to a Web link. The URL generally consists of the service domain name (or IP address), bucket name, and object name. For example, `https://s3-us-west-2.amazonaws.com/foo_bucket/foo` refers to a cloud storage object `foo`, which is in the bucket `foo_bucket` and located in the data centers of Amazon S3 in U.S. West region (Oregon) `s3-us-west-2.amazonaws.com`.

Typically, cloud storage service providers provide an HTTP-based representational state transfer (REST) interface for accessing cloud storage objects. Two important operations are `PUT`

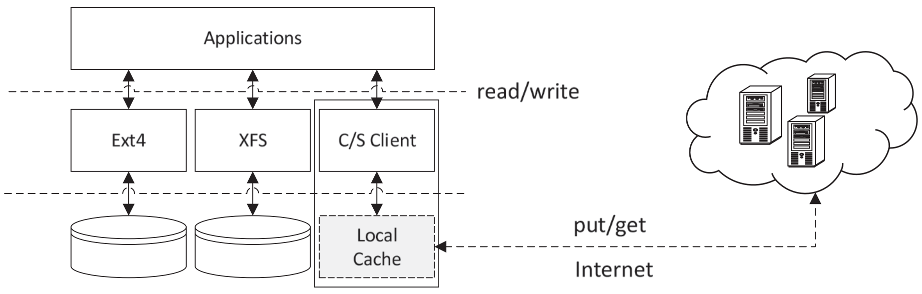


Fig. 1. Client-side caching for cloud storage.

(uploading) and GET (downloading), which are akin to `write` and `read` in file operations. The APIs also provide DELETE, HEAD, and POST to allow users to remove objects, and retrieve and change object metadata. For each operation, the target URL must be presented to specify the target in the cloud storage.

2.2 Cloud Storage Clients

Cloud storage services typically provide a POSIX-like interface to client applications, allowing end users to access cloud storage in a way that is similar to accessing local file systems. Under the hood, the client application translates the file system–level `read` and `write` to cloud storage GET and PUT requests. To reduce the I/O traffic to the cloud, there are two typical ways to enhance user experience: syncing and caching.

With the syncing approach, the client needs to mirror a complete copy of the cloud storage data in local devices. A syncer thread monitors the changes made to the local copies and periodically flushes the changes back to the on-cloud repository. The merit of such a syncing approach is its simplicity. Almost all read I/Os can be satisfied locally, but it demands a large amount of local storage space to manage the entire dataset. Essentially, the role of the cloud storage in the syncing mode is more like a *secondary* storage. In practice, this approach is often adopted by personal cloud storage (e.g., Dropbox [24], Google Drive [29], and OneDrive [51]) to provide data services, such as sync-and-share, collaborative editing, data recovery, or data archiving.

The second approach is caching, which only maintains the most important data in local cache, and only cache misses result in on-demand cloud I/Os. Such an approach is sophisticated but potentially could result in more cloud I/O activities, and therefore a well-designed caching policy is the key to its success. Currently, the caching mode is adopted by some open-source clients and in academic studies, such as RFS [21], S3FS [63], S3Backer [62], BlueSky [71], and SCFS [9]. Almost all of these systems use an LRU-based replacement for managing the cloud storage cache, which unfortunately is suboptimal for latency and cost considerations.

In this article, we are particularly interested in integrating cloud storage as a primary storage layer to serve I/O-intensive workloads with the caching approach. Figure 1 illustrates the client-side caching for cloud storage. In practice, a client cache can be not only a local storage device but also a client-side gateway or proxy.

2.3 GreedyDual-Size

GDS is a classic cost-aware caching replacement scheme and has been proven to be online optimal [14]. It is an extension of a simple but efficient algorithm called *GreedyDual* [75]. GDS makes the replacement decision by leveraging the recency, size, and fetching cost of cached documents.

```
0   Initialize L = 0.
1   For processing each requested document p:
2
3     if p is already in memory,
4        H(p) = L + Cost(p)/Size(p).
5
6     if p is not in memory,
7        while there is not enough room in memory for p,
8           Let  L = min H
9           Evict q such that H(q) = L.
10       Bring p into memory and set H(p) = L + Cost(p)/Size(p)
```

Fig. 2. GDS algorithm [14].

Figure 2 illustrates the algorithm of GDS [14]. In this algorithm, each document is associated with a value, H (lines 4 and 10), to determine the caching priority of an object: when a cache replacement happens, the document with the minimum H value will be evicted (lines 8 and 9). The H value is a sum of two components (lines 4 and 10). The first component is a global value L (lines 0 and 8), which is used to incorporate the recency into the cost function by tracking the minimum H value (line 8). Since the document with the minimum H value is always selected for eviction (lines 8 and 9), the L value keeps inflating. Consequently, a more recently accessed document has a larger L value and thus potentially has a larger H value, tending to have a higher chance of staying in cache. The second component is the value of $\frac{cost}{size}$, which represents the considerations of cost and size by calculating the average cost per size unit. As such, GDS is able to integrate the three factors (including recency, cost, and size in making a caching decision) to minimize the overall fetching cost.

The GDS algorithm was originally designed for Web caching. Variants of GDS have also been proven to be efficient in other working scenarios, such as page cache [40] and key-value stores [44]. In this article, we present a caching scheme based on GDS to optimize user experience in terms of both access latency and monetary cost for cloud storage caching.

3 CACHING ISSUES

3.1 Challenges

To make a caching scheme effectively achieve two optimization targets is nontrivial. In a cloud storage scenario, we have to consider at least three critical factors for optimization: locality, latency, and price. Locality represents on the time axis how likely an object will be reaccessed in the future. The better locality is, the longer the object should stay in cache. Latency specifies how much time is needed to complete one cloud I/O request, such as `PUT` or `GET`. The longer the latency is, the more performance impact would be perceived by the user, as the user has to wait for the object to be retrieved from the cloud, which directly influences the user experience. Price is the monetary cost that a user has to pay for completing one cloud I/O. It is determined by the pricing model of the cloud storage service provider.

What makes the caching decision complicated is that, although related, the factors mentioned previously are orthogonal to each other. For example, a high-latency object may not be an object that will soon be accessed (weak locality), and a high-cost object may not raise a high latency for accessing (e.g., an object in a more distant data center is cheaper to retrieve). How to address these situations is challenging. A well-designed caching policy must consider and balance all of the factors to identify the object that will incur the lowest penalty if being chosen to be evicted as the victim. A cost-aware caching can *indirectly* reshape future cloud I/Os, and the ideal situation is that we only see a small number of low-latency and low-price cloud I/Os.

3.2 Revisiting GDS in Cloud Storage

As a typical cost-aware caching scheme, GDS has considered recency and other factors, including file size and the fetching cost (see Section 2.3). However, it is difficult for GDS to be directly used in the cloud environment for several reasons.

First, the original GDS can only optimize for one cost target. Cloud storage users are highly sensitive to both performance and monetary cost, especially for running I/O-intensive workloads. Unfortunately, these two optimization goals are orthogonal, thus directly combining these two optimization dimensions together as a single numeric value lacks a concrete semantic basis. A straightforward method, for example, is to set the weighted average value of the two optimization targets as a combined cost. However, this method is based on the assumption that access latency and monetary cost are exchangeable and can be directly compared (i.e., 1 second = 1 dollar), which is not semantically meaningful. In addition, the diversity of working scenarios and pricing models further complicates the selection of the weights. Therefore, the method of using a single value as the combined cost is suboptimal. In this work, to achieve two optimization goals, we adopt a two-region design, in which each region focuses on minimizing one cost target (see Section 4.1).

Second, unlike storage I/Os in traditional Web systems, in which the Web pages are frequently read and rarely modified, storage I/Os in the cloud environment are bidirectional, meaning that the data can be not only frequently read (downloaded) but also frequently written (uploaded). The problem of directly using the cost function of the original GDS algorithm designed for Web caching is that it simply defines the *cost* as the penalties of fetching objects and ignores the cost differences of clean objects and dirty objects. Using cloud storage as a primary storage, an object can be both read (downloaded) or written (uploaded), and consequently the cost of evicting clean objects and dirty objects can be different in cache management—evicting a dirty object incurs a high on-demand uploading latency, in addition to the downloading latency of fetching the object upon a related cache miss. Considering this, we define the *cost* as the penalties of evicting an object, including the cost of downloading the object and the cost of uploading for dirty objects (see Section 4.2).

Third, assessing the access cost of cloud storage should also consider several cloud environment issues. For example, the access time could fluctuate due to many real-time factors (e.g., network conditions). The variance of access latency may degrade the efficiency of cost-aware caching and thus has to be well considered. To address this issue, we adopt an adaptive normalization approach (see Section 4.2).

4 DESIGN OF GDS-LC

In this section, we present the design of GDS-LC, which exploits locality, size, latency, and price to improve caching efficiency, aiming to minimize both access latency and monetary cost. We first describe the basic cache design of GDS-LC and then present GDS-LCF, which is an enhanced version of GDS-LC by introducing frequency into caching decisions.

4.1 Cache Space Management

To achieve both optimization goals in terms of access latency and monetary cost, in the design of GDS-LC we adopt a two-region design: each region is managed with a dedicated cost-aware caching scheme to achieve a specific optimization target (either low access latency or low monetary cost), respectively; objects are migrated between the two regions, and the low-locality, low-latency, and low-cost ones will be finally evicted from the local cache.

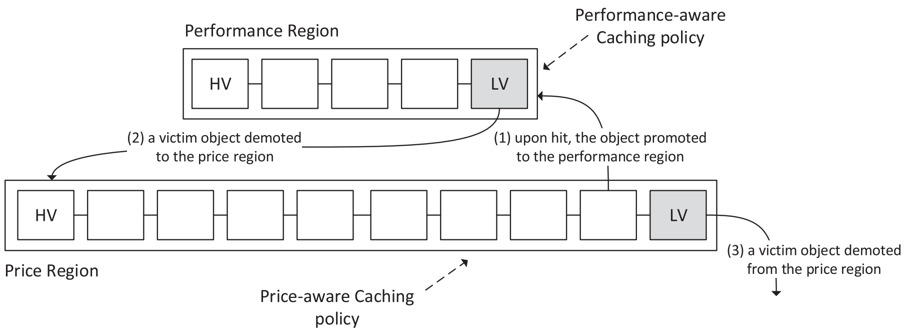

Performance Region
Performance-aware Caching policy
HV
LV
(2) a victim object demoted to the price region
(1) upon hit, the object promoted to the performance region
HV
LV
(3) a victim object demoted from the price region
Price Region
Price-aware Caching policy

Fig. 3. Two-region structure for caching.

Cache partitioning. As shown in Figure 3, GDS-LC logically splits the cache space into two regions: a performance region and a price region. The performance region is a high-priority region, which is reserved to contain performance-critical objects (i.e., hot objects that are to be reaccessed shortly and have long access latencies). The cost region is a low-priority region, which contains relatively cold objects with weaker locality but a higher monetary cost. The two regions adopt two different replacement algorithms: GDS-Latency and GDS-Cost, which focus on the latency and price goals, respectively. Particularly, when considering the cost, we focus on cost per size unit by using latency/size and price/size, which is based on GDS (see Section 4.2).

The main reason for such a two-region design is twofold. First, separating objects into two regions allows us to apply different caching replacement algorithms for the management rather than blending all factors in a meaningless numeric value. Second, we can flexibly give different priorities to different optimization goals. Considering the excessively long access latency to the cloud is a critical concern for most users, in our design we regard performance as more important than monetary cost and thus give a higher priority to performance by setting the performance region as a top region. In practice, high priority can also be given to the monetary concern by setting the price region as the top region.

It is worth noting that GDS-LC adopts "logical" partitioning, which means that it is unnecessary to physically partition the cache space, and we simply keep track of the actual space occupied by the objects in each region. For the partition sizes, we adopt a scheme similar to the memory management in the Linux kernel by reserving one third of the total client cache space for the performance region, and the rest is reserved for the price region. Theoretically, such a cache partitioning can be dynamically tuned. In our experiments, we find that the ratio 1:2 works very well across all workloads in our test. We will further discuss the impact of different caching partitioning ratios in Section 5.3.

Object migration. Figure 4 shows the algorithm of object migration between the two regions. Initially, an object is admitted into the performance region (lines 23 through 31). If the performance region has available space, the object will be added into the performance region (line 31). If the performance region has insufficient space, we need to run the GDS-Latency algorithm (see Section 4.2) to move one or multiple *low-latency* objects to the price region to accommodate the new object (lines 25 through 30). In this process, the object with the weakest locality and the smallest latency will be demoted (line 26). If there is insufficient space in the price region, by running GDS-Cost (see Section 4.2) we further evict *low-cost* objects from the price region and reclaim enough partition space (lines 27 and 28).

A second access to an object in the price region will promote it into the high-priority performance region (lines 6 through 21), as the object has proven itself to have high temporal locality. If

```
0   /* Procedure is invoked upon a reference to object b */
1   reference_object (b)
2   {
3     if b is in cache { /* hit in cache */
4       if b is in performance region
5         hit_object_in_region(b, performance)
6       else {
7         remove_object_from_region(b, price)
8
9        /* if no space in performance region,
10           demote some objects to price region*/
11
12        while (b.size > performance.free_space)i{
13          a = evict_object_from_region(performance)
14          /* if no enough space in price region,
15             evict some objects*/
16          while (a.size > price.free_space)
17             evict_object_from_region(price)
18          add_object_to_region(a, price)
19        }
20        add_object_to_region(b, performance)
21      }
22    } else { /* miss in cache*/
23        download b from the cloud storage
24
25        while (b.size > performance.free_space){
26           a = evict_object_from_region(performance)
27           while (a.size > price.free_space)
28             evict_object_from_region(price)
29           add_object_to_region(a, price)
30         }
31         add_object_to_region(b, performance)
32    }
33  }
```

Fig. 4. Algorithm of migrating objects between regions.

the performance region has available space, the object is added into the region (line 20); otherwise, we have to evict one or multiple objects from the performance region and demote them into the lower-priority price region (lines 12 through 19).

As illustrated earlier, in the two-region design, each cost-aware caching scheme works like a filter: GDS-Latency filters out the low-latency objects, and GDS-Cost filters out the low-cost objects, and the migration gives a high caching priority to high-locality objects. Therefore, the low-locality, low-latency, and low-cost objects will be first evicted from the local cache.

4.2 Cost-Aware Caching Replacement

As described earlier, we split the client cache space into two regions, each of which adopts a cost-aware replacement algorithm, i.e., GDS-Latency and GDS-Cost, to identify the victim objects for eviction. Both GDS-Latency and GDS-Cost are based on GDS but use a carefully designed cost function. Namely, we calculate the value of an object in each region by applying corresponding cost functions to the equation of GDS: $H(obj) = L_{region} + Cost(obj)/Size(obj)$ (see Section 2.3 for details). In this section, we discuss the *latency* function used in GDS-Latency and the *price* function used in GDS-Cost.

4.2.1 Latency Function. Compared to the latency function used in the original GDS [14], our latency function has two particular considerations: clean-dirty differentiation and adaptive normalization.

Clean-dirty differentiation. For evicting a clean object, we set the cost to be the latency of downloading the object from the cloud, which is similar to the original GDS [14]. The difference is that for evicting a dirty object, we set the cost to be the sum of the latency of downloading the object and the latency of uploading it to the cloud. This is motivated by the fact that dirty objects have to be synchronized to the cloud before being discarded. Therefore, in addition to the miss penalty (i.e., the downloading latency caused by a cache miss), the cost of evicting a dirty object should also include the uploading latency. Such a clean-dirty differentiation gives relatively higher values to dirty objects and is thus helpful to reduce the on-demand uploading latencies experienced by the users.

Adaptive normalization. To evaluate the cost in terms of latency, we can measure the access latency online and correspondingly calculate the cost associated with each object. However, due to the possible variance of network performance and the speed of cloud servers, the latency of uploading/downloading an object from the cloud may not be constant. The latency variance may lead to inaccurate cost evaluations and thus deteriorate the efficiency of latency-aware caching.

The rationale behind the latency-aware caching scheme is that the miss penalty of an object is the time used to download the object (i.e., downloading latency) when the evicted object is retrieved again. When the downloading latency fluctuates, the latency-aware caching schemes will make inaccurate cost estimations. For example, if downloading an object A needs 0.4 seconds, the latency-aware caching scheme will take 0.4 seconds as the miss penalty of evicting object A. However, downloading object A from the cloud upon a cache miss may take a shorter (e.g., 0.36 seconds) or longer time (e.g., 0.42 seconds). In other words, the real cache miss penalty may be lower or higher than evaluated, leading to a mistaken selection of victim objects.

To alleviate this problem, we normalize the latency (including both the download latency and upload latency) by dividing it by *a normalization factor* and rounding the result up to an integer. Specifically, the latencies that are no larger than the normalization factor will be normalized to 1; otherwise, they will be normalized to the nearest integers. For example, if we set the normalization factor as 0.2 seconds, the absolute values of the latencies (e.g., 0.36, 0.4, and 0.42) are normalized to the same value (i.e., 2). Although the interference of latency variance cannot be completely avoided, with the normalization approach we can allow the algorithm to tolerate certain variance of access latencies in a small range and in the meantime still retain the capability of differentiating high-cost and low-cost objects with a reasonable resolution.

It is also worth noting that a normalization factor with a fixed absolute value cannot fit all scenarios in the real world. With an excessively small normalization factor, the negative effects of the variance of access latencies cannot be effectively reduced. However, an excessively large normalization factor may weaken the capability of differentiating distinct costs of objects. In our experiments, we set an adaptive normalization factor based on the round-trip time (RTT) between the client and the cloud. Specifically, we set the normalization factor to be multiple times of RTT. If the client simultaneously connects with multiple clouds located in different geographic locations, we set the normalization factor to be multiple times of RTT_{min} (i.e., the minimum RTT). The normalization factor can be tuned under different working scenarios. We will further study the impact of normalization in Section 5.4.

4.2.2 Price Function. Due to its service nature, each cloud I/O takes a certain amount of monetary cost. Based on the service provider's current pricing model, it includes three components:

storage cost, request cost, and data transfer cost. Since the storage cost is based on the total size of all objects stored on the cloud, it is not related to the real-time accesses. The price of a cloud I/O includes two latter components: the request cost and data transfer cost.

Specifically, for downloading an object, the price is the sum of the cost of a GET request and the cost of transferring data out from the cloud; for uploading an object, since most cloud storage service providers do not charge data transfer cost for uploading, the price of uploading an object equals the cost of a PUT request. Similar to the latency function, we also differentiate the monetary cost of evicting a clean object and a dirty object—that is, we set the monetary cost of evicting a clean object to be the price of downloading the object and set the monetary cost of evicting a dirty object to be the sum of the uploading price and the downloading price.

It is worth noting that the pricing model of a cloud storage service provider is not always constant, and different cloud storage service providers or even different data centers of the same cloud storage service provider can price differently. When the service provider's pricing model changes, this price function should be updated as well.

4.3 Further Enhancement

The framework of GDS-LC is highly flexible. It can be easily extended to include additional factors to make caching decisions. In this section, we introduce a further enhancement to GDS-LC by including the consideration of access frequency into the cache replacement.

In GDS-LC, the two regions adopt two caching schemes: GDS-Latency and GDS-Cost, where the H value (used to determine the caching priority) is updated when the object is admitted into the cache (see Section 4.2). Both approaches do not effectively reflect how frequently an object is referenced while it is resident in cache. As a further enhancement, we propose the second method to incorporate the frequency information into the calculation of the object values. We call *GDS-LCF* the frequency version of GDS-LC. Correspondingly, we call *GDS-LF* the frequency-version of GDS-Latency and *GDS-CF* the frequency-version of GDS-Cost.

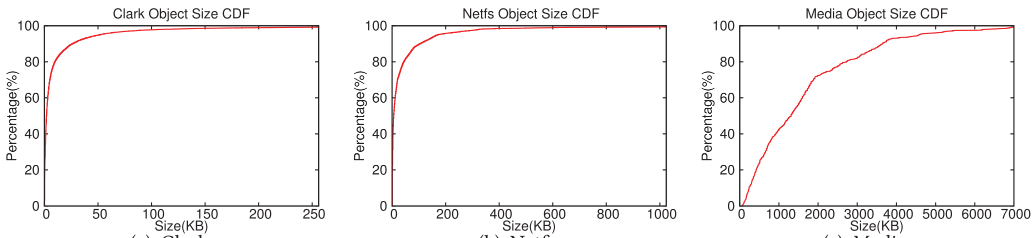
By incorporating frequency into the equation used in each region (see Section 4.2), we get a new equation for GDS-LF and GDS-CF to calculate the value of an object (e.g., obj): $H(obj) = L_{region} + Cost(obj) \times Freq(obj)/Size(obj)$, in which $Freq(obj)$ refers to the approximation function of frequency. Determining a proper approximation function of frequency (i.e., $Freq(obj)$) is an important issue, as it may affect the caching efficiency substantially [49]. Frequently accessed objects are important even if not recently accessed. Some caching algorithms count at most two most recent references to each cache page (e.g., ARC [49] and LRU-2 [56]). Similarly, we set $Freq(obj)$ to not become greater than two and four for the top and bottom regions, respectively.

Specifically, we associate each object with a *counter*, which is incremented by one upon an access to the object and stays unchanged when the object is demoted. $Freq(obj)$ is updated as follows. First, for an object in the top region, $Freq(obj)$ is set to two if the *counter* is larger than two; otherwise, $Freq(obj)$ is set to the value of the *counter*. Second, for an object in the bottom region, $Freq(obj)$ is set to four if the *counter* is larger than four; otherwise, $Freq(obj)$ is set to the value of the *counter*.

With such an approximation, we can effectively avoid the possible situation that frequency outweighs other factors in the extreme cases. Our experiments show that our approximation function works satisfactorily. It is also worth noting that GDS-LCF incurs trivial overhead. For cost-aware caching, such as GDS and GDS-LC, the most important metadata of the cached objects is maintained locally (e.g., the retrieval path, the latest modified time, the state indicating whether it is clean or dirty). Comparatively, GDS-LCF adds only one additional *counter*, which increases negligible time and space overhead.

Table 1. Trace Characteristics

Trace	Service Type	Total Unique Object Size	PUT Requests	GET Requests
Clark	Web	164MB	645	229,233
Netfs	Files system	707MB	594,433	135,949
Media	Multimedia	1,294MB	0	166,366



(a) Clark (b) Netfs (c) Media

Fig. 5. CDF of object sizes.

5 PERFORMANCE EVALUATION

5.1 Experimental Methodology and Environment

5.1.1 Trace-Driven Emulation. To evaluate our proposed caching schemes, we have developed a prototype to emulate a cloud storage cache manager. Our prototype simulates a typical cloud storage client cache, which leverages a specified amount of local storage space as cache for cloud storage. I/O accesses that cannot be satisfied in the local cache will be converted into PUT or GET requests to the cloud storage. For each request, we recorded the execution information, including the request type, the end-to-end completion time, and whether it is a cache hit. This information can be used to calculate the hit ratio, average latency, and monetary cost of each run of the experiments (see Section 5.1.4 for the methodology of result reporting in our experiments).

Considering that Web services, file system services (e.g., S3FS [63], BlueSky [71], Tombolo [74], and SCFS [9]), and multimedia services (e.g., Netflix is deployed on Amazon S3 [4], and Spotify has moved to Google Cloud [37]) are popular and typical services using cloud storage as a *primary* storage, we selected three representative workloads in the experiments: *Clark*, *Netfs*, and *Media* (details of the traces are shown in Table 1, and Figure 5 shows the distributions of object sizes):

- *Clark* is a Web trace [19], which accesses 164MB of unique objects (i.e., Web pages) and consists of 229,233 GET requests and only 645 PUT requests. The object size distribution is shown in Figure 5(a). This workload is highly read intensive. Thus, we use a write-through policy to synchronize the data to the cloud for this workload.
- *Netfs* is a file system workload converted from the *networkfs* workload in FileBench 1.4.9 [48] running on S3FS [63]. We collect the PUT and GET requests in a trace file. This workload is more write intensive. It accesses 707MB of unique objects (i.e., files) in total, including 135,949 GET requests and 594,433 PUT requests. The object size distribution is shown in Figure 5(b). A write-back policy is adopted to sync back the dirty data that reside in the cache for more than 30 seconds periodically (every 5 seconds), similar to the Linux write-back policy. All updated objects are filled in with randomly generated content.
- *Media* is a multimedia workload synthesized using the open-source generator MediSyn, in which the access pattern of multimedia objects (e.g., small video and audio files) follows

Table 2. The Pricing Model of Amazon S3 Services

	Client	Data Center	Request Cost PUT	Request Cost GET	Transfer Cost PUT	Transfer Cost GET
Local cloud	Oregon	Oregon	\$0.005/1,000	\$0.004/10,000	0	0
Internet cloud	Louisiana	Oregon	\$0.005/1,000	\$0.004/10,000	0	\$0.090/GB
Hetero cloud	Singapore	Oregon	\$0.005/1,000	\$0.004/10,000	0	\$0.020/GB
	Singapore	Tokyo	\$0.0047/1,000	\$0.0037/10,000	0	\$0.090/GB

Note: This table shows a simplified version of the Amazon S3 pricing policy. The referenced price data was collected on December 6, 2016. The actual price fluctuates. Interested readers may refer to the Amazon Web site for more details [3].

Zipfian distribution [68]. We synthesized this workload by collecting the size of each unique object and traced the access sequence of the object ID. By replaying this trace, we aim to simulate the object-based client caching for multimedia objects, which has attracted attention from academia (e.g., Acharya and Smith [2], Amazon [32], and Shafiq et al. [64]) and is widely adopted in industrial products (e.g., VideoCache [70] and Blue Coat ProxySG Appliances [11]). This workload accesses 1,294MB of unique objects. As a typical multimedia workload, it is read-only and contains 166,366 GET requests. All objects are filled in with randomly generated content. The object size distribution is shown in Figure 5(c).

5.1.2 Experimental Platform. Our experiments were conducted with Amazon Simple Storage Services (S3). As a representative cloud storage service, Amazon S3 is widely adopted as a storage layer in various consumer and commercial services such as Netflix [4]. Some consumer cloud storage services, such as Dropbox, directly use S3 as the low-level storage system for data hosting [65]. In our experiments, we used the S3 storage hosted in Amazon's data centers in Oregon (s3-us-west-2.amazonaws.com) and Tokyo (s3-ap-northeast-1.amazonaws.com) as the cloud storage service providers. We also used three clients: two Amazon EC2 instances and a workstation on our campus. All three clients use the Linux 3.2.1 kernel and Ext4 file system.

To comprehensively test our GDS-LC algorithm, we designed three different system setups. Each system setup simulates a typical working scenario of cloud storage in the real world:

- *Local cloud* simulates a typical cloud system where the client and the storage servers are in the same data center. In our experiments, the client is an Amazon EC2 instance and located in the Oregon data center with the Amazon S3 cloud storage.
- *Internet cloud* simulates a public cloud system in a consumer environment where the client connects to the storage service through the Internet. The client is a workstation on our campus in Louisiana, and the S3 cloud storage is in the Oregon data center.
- *Hetero cloud* simulates a special scenario where a client connects simultaneously to two different clouds. The client is an EC2 instance in Singapore and connects to two S3 cloud storage systems, one in Tokyo and the other in Oregon.

Table 2 shows the details of the pricing model corresponding to each system setup used in our experiments. It is also worth noting that we do not intend to cover all possible use cases in the experiments; instead, using these system setups with different features in terms of access latencies and pricing policies, we attempt to evaluate various latency/cost implications in our solution.

5.1.3 Algorithms for Comparison. In our experiments, we compared our proposed caching schemes to typical traditional caching algorithms and the original GDS-based algorithms in different working scenarios. We also conducted a series of experiments to test the impact of critical parameter settings and further compare our proposed caching schemes to the customized GDS-based algorithms.

Basic experiments. We compare our caching schemes GDS-LC and GDS-LCF to two traditional caching algorithms that focus on improving hit ratios (i.e., *LRU* and *ARC*) and two different settings of the original GDS algorithm (i.e., GDS(latency) and GDS(price)). We present the basic experimental results in Section 5.2. The configurations of these algorithms are as follows:

- *LRU*: The traditional LRU policy, which applies the LRU replacement algorithm. As far as we can see in practical systems, LRU is currently the most widely adopted caching algorithm in cloud-based storage services in academia (e.g., BlueSky [71] and SCFS [9]) and industry (e.g., Nasuni [54] and SteelStore [55]).
- *ARC*: ARC is an advanced caching algorithm, which improves LRU by making use of history access references with ghost buffers to efficiently filter one-time access [49]. ARC splits the cache space into two LRU lists (i.e., T1 and T2) to manage the cache entries that are recently referenced and the cache entries that are frequently referenced (at least twice), respectively. The cache entries in T1 are promoted to T2 when they are referenced again. ARC also maintains two *ghost* LRU lists (i.e., B1 and B2) to track the cache entries evicted from T1 and T2, respectively. The sizes of the four LRU lists can be tuned adaptive to the access pattern of workloads (see the literature [49] for details). Since the original ARC replacement algorithm is designed for page cache and each caching unit is a fixed-size page or block (generally 4KB), the basic adaptation granularity of ARC is the page size. In our experiments, we replace the original adaptation granularity (page size) with the object size. With such a customization, ARC can work for variable-size objects but does not rely on object sizes and the associated costs to make caching decisions, as its methodology and working principles are not changed. Comparing our caching schemes with ARC, we aim to reveal that it is not enough to only consider recency and frequency for cloud storage caching.
- *GDS(latency)*: The original GDS caching scheme that directly uses the downloading latency of each object as the cost function. With this configuration, GDS aims at minimizing the overall latency.
- *GDS(price)*: The original GDS caching scheme that uses the monetary cost of downloading each object as the cost function. With this configuration, GDS aims at minimizing the overall monetary cost.
- *GDS-LC*: The cache is divided into two regions (a performance region and a cost region). We use a size ratio of 1:2, similar to page cache management in Linux. The performance region is managed with the GDS-Latency scheme, and the cost region uses the GDS-Cost scheme. The difference between GDS-Latency and GDS(latency) is that the former scheme differentiates the cost of clean and dirty object and uses the normalized latency as the cost function. Similarly, compared to GDS(price), GDS-Cost has a different monetary cost function for dirty objects. For the normalization factor, we set it to 10 times the RTT between the client and the cloud. In particular, in the scenario of the heterogeneous cloud, we set it to 10 times the minimum RTT from the client to the clouds.
- *GDS-LCF*: The cache partitioning is the same as GDS-LC, but we further introduce the frequency factor into the cost function. Thus, a more frequently read or written object will have a larger weight to be protected in the local cache. For the frequency approximation, we count the access frequency at most two when it is in the performance region and at most four when it is in the price region (see Section 4.3). GDS-LCF sets the same normalization factor as GDS-LC.

Extensive experiments. In addition to the basic experiments, we investigate the effect of partition sizes in Section 5.3 and study the impact of latency normalization in Section 5.4. We also compare our proposed caching schemes to the frequency version of GDS and the enhanced GDS-based caching schemes that can recognize clean and dirty objects in Section 5.5 and Section 5.6, respectively.

5.1.4 Methodology of Result Reporting. Since access latency and monetary cost are two optimization goals of our caching schemes, we take the average latency and monetary cost as our major metrics. We also report the hit ratio, which is one of the most critical metrics to evaluating caching efficiency. Each experiment is repeated five times. After each run of the experiments, we calculate the average latency of all requests (including both the requests served by local cache and the requests served by cloud), the total monetary cost charged by accessing the cloud, and the hit ratio.

After all experiments, we have five sets of average latency, monetary cost, and hit ratio. For each metric, we finally report the average value $\overline{x} = \frac{1}{N}\sum_{i=1}^{N} x_i$, in which x_i denotes the value of the metric obtained from the i_{th} run of the experiments and N denotes the number of runs. We also calculate the standard error $SE = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$, which describes the variance of the experimental results.

5.2 Basic Experimental Results

5.2.1 Local Cloud. Large enterprises often require high-performance cloud storage services to efficiently store/retrieve the data. To satisfy this requirement, managing the data from a client located in the same data center as the storage servers is a desirable choice in terms of performance and monetary cost. In such an environment, both clients and the cloud are close to each other and the network connection is good. Typically, the client-cloud RTT is low (0.28ms in our system setup).

Figure 6 shows the experimental results with all three workloads in the local cloud scenario. From the results, we not only see the advantages of our caching design but also observe some interesting behaviors of different caching schemes. In this section, we first present the observations on the experimental results of the read-intensive workloads (*Clark* and *Media*) and then present some different observations on the experimental results of the write-intensive workload (*Netfs*).

Observations on the read-intensive workloads. In our experiments, both the *Clark* workload and the *Media* workload are read intensive: *Clark* is dominated by read requests, and a write-through policy is adopted; *Media* is read only (see Section 5.1.1). Consequently, all victim objects are clean when working with these two workloads. From the experimental results obtained with *Clark* and *Media*, we have the following observations.

The GDS-based policies are observed to be better than LRU and ARC. This is because the GDS-based policies take recency, object size, and cost (in terms of both latency and price) into account, whereas LRU and ARC are cost unaware. Since the GDS-based policies prefer to evict the objects of larger size and smaller cost, these replacement policies have higher caching efficiency. Particularly, in this scenario, the price for evicting each object is equal, as all objects to be evicted are clean and the price only includes the cost of `GET` requests for internal data transfer in the data center (see Table 2). In this case, the GDS-based caching schemes cause less monetary cost than LRU and ARC, as they generally have higher hit ratios (see Figure 6(a) and (g)). As shown in Figure 7, we also take the experimental results with *Clark*, of which the cache size is set to 10% of the total size of unique objects, to investigate the caching behaviors of different caching schemes. Figure 7(a) shows the distributions of the end-to-end completion time of all requests (including both the requests served by the local cache and those served by the cloud). Figure 7(b) shows the differences among the size

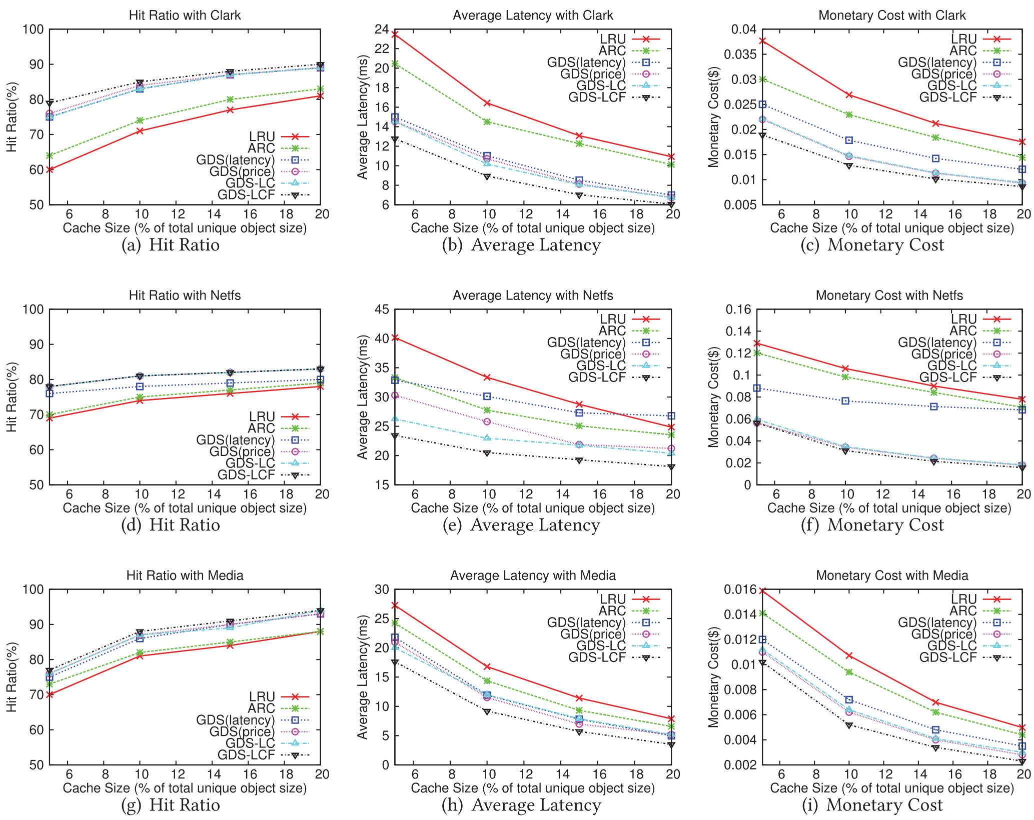
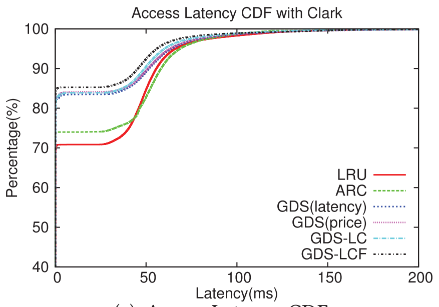


Fig. 6. Local cloud: Hit ratios, average latencies, and monetary cost of *Clark*, *Netfs*, and *Media*.

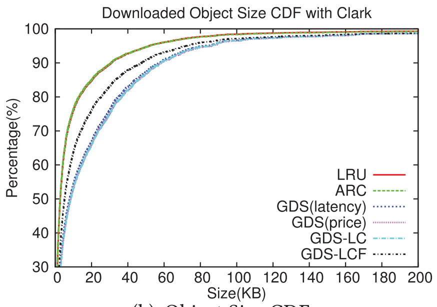
distributions with different caching schemes. For example, the object size larger than 20KB is 37% of all downloaded objects upon related cache misses with GDS-LC, but the corresponding number with LRU is 18%. The reason is that the GDS-based caching schemes prefer evicting larger objects. In contrast, the differences of the latency distributions are not so significant (see Figure 7(c)). This is because the access latency does not increase as the request size increases for small requests (e.g., smaller than 64KB), which has been reported in a prior study [33].

Compared to GDS(latency) and GDS(price), GDS-LC can minimize both average latency and monetary cost. Specifically, the average latency of GDS-LC is close to that of GDS(latency) (see Figure 6(b) and (h)), and the monetary cost of GDS-LC is close to that of GDS(price) (see Figure 6(c) and (i)). This demonstrates the effect of the two-region design of GDS-LC: via adopting GDS-Latency in the performance region and GDS-Cost in the price region, GDS-LC keeps the most "expensive" objects in terms of both latency and monetary cost in the cache so that it can optimize both metrics at the same time.

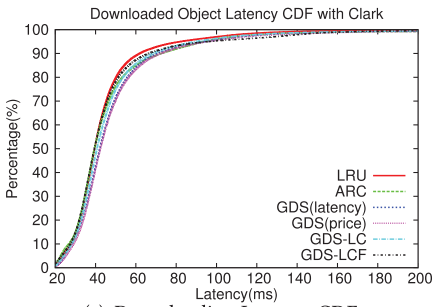
GDS-LCF performs the best in this scenario. The difference between GDS-LC and GDS-LCF is that GDS-LCF further includes the frequency into the caching consideration, which helps identify the hottest object from the perspective of popularity. Consider this case: object A has value 1, being accessed four times, and object B has value 2, being accessed once. With GDS-LC, object A will be evicted because GDS-LC is unaware of the access frequency; whereas based on GDS-LCF, object B will be evicted ($2 \times 1 < 1 \times 4$). Thus, GDS-LCF focuses more on the frequently accessed objects, and the experimental results demonstrate the strength of such a consideration.



(a) Access Latency CDF



(b) Object Size CDF

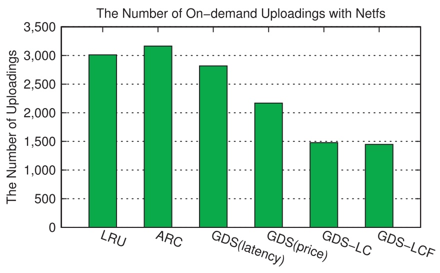


(c) Downloading Latency CDF

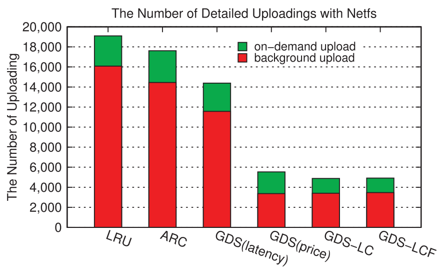
Fig. 7. Local cloud: An example with *Clark* to investigate the caching behaviors of different caching schemes, in which the cache size is set to 10% of the working set (i.e., the total size of unique objects). (a) CDFs of the access latencies. (b, c): CDFs of sizes and downloading latencies of the objects fetched from the cloud upon related cache misses, respectively.

Observations on the write-intensive workload. Compared to *Clark* and *Media*, *Netfs* has more intensive writes, and dirty data are asynchronously written back to the cloud periodically (see Section 5.1.1). With this workload, we have similar observations, which show the advantages of our caching schemes: GDS-LCF performs the best in this experiment, and GDS-LC can optimize both average latency and monetary cost. Meanwhile, for such a write-intensive workload, we also have some different observations, which are presented next.

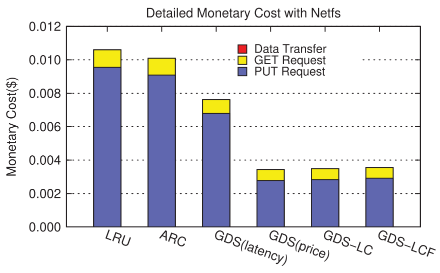
GDS-LC can achieve lower average latency than both GDS(latency) and GDS(price), especially when the cache size is relatively small. As shown in Figure 6(e), for example, when the cache size is 5% of the working set, GDS-LC reduces the average latency by 21% (from 33 to 26 ms). That is because GDS-LC particularly considers the cost of data synchronization for evicting dirty objects so that less dirty objects are discarded when the cache space is not enough; consequently, GDS-LC makes the requests suffer less from waiting for on-demand synchronization (i.e., uploading). This is consistent with our observation on the number of uploadings with different caching schemes: as shown in Figure 8(a), compared to GDS(latency), GDS-LC decreases the number of on-demand uploadings by 46% (from 2,800 to 1,500). As for the price, we find that GDS-LC and GDS-LCF do not have obvious advantages over GDS(price). This is because the total uploadings of these three caching schemes (i.e., GDS-LC, GDS-LCF, and GDS(price)) are comparable (see Figure 8(b)). At the same time, since data transfer is not charged in this scenario, and the fee of PUT request is 12.5 times as that of GET request (see Table 2), the charge of the PUT requests dominates the overall monetary cost; thus, the monetary costs of these three caching schemes (i.e., GDS-LC, GDS-LCF, and GDS(price)) are comparable (see Figure 8(c)).



(a) On-demand Uploadings



(b) Detailed Uploadings



(c) Detailed Monetary Cost

Fig. 8. Local cloud: The number of uploadings and detailed monetary cost of *Netfs* achieved by different caching schemes with the cache size set to 10% of the working set (i.e., the total size of unique objects). The number of uploadings refers to the number of uploading requests caused by synchronizing dirty objects to the cloud. Specifically, on-demand uploading refers to synchronizing dirty objects to the target cloud when being evicted from the local cache, and background uploading refers to synchronizing dirty objects to the target cloud with the background write-back daemon. The monetary cost for accessing cloud objects includes a data transfer fee and a request fee (see Table 2 for the pricing model used in our experiments).

ARC has more on-demand uploadings than other caching schemes (see Figure 8(a)). This is because LRU always evicts the least recently accessed objects, which means that the most recently written objects (i.e., dirty objects) will be protected in the cache. In contrast, ARC also attempts to recognize one-time accesses and select such objects as victim objects, resulting in more on-demand uploadings than LRU. However, ARC has a higher cache hit ratio than LRU, which means that ARC can absorb more write traffic than LRU, and therefore ARC has less background uploadings than LRU and finally creates less total uploadings and monetary cost than LRU. However, compared to GDS-based policies, both LRU and ARC have more uploadings and monetary cost (see Figure 8), as both LRU and ARC have lower hit ratios than the GDS-based policies and do not actively differentiate clean and dirty objects (compared to GDS-LC and GDS-LCF).

GDS(latency) does not work as well as expected. From Figure 6(e), we find that the performance of GDS(latency) is worse than that of GDS(price), and even worse than that of LRU when the cache size is 20% of the working set. It is understandable. Working with *Netfs*, the objects may be frequently updated with an object size change; in this case, the access latencies could be different from the values observed previously. Without a reasonable estimation, the cost used in the caching replacement scheme may be different from the real value. Comparatively, the performance of our solution GDS-LC is more stable. This is because we only adopt the latency-aware caching scheme in the first region, and optimization including clean-dirty cost differentiation and latency normalization contribute to improving the caching efficiency.

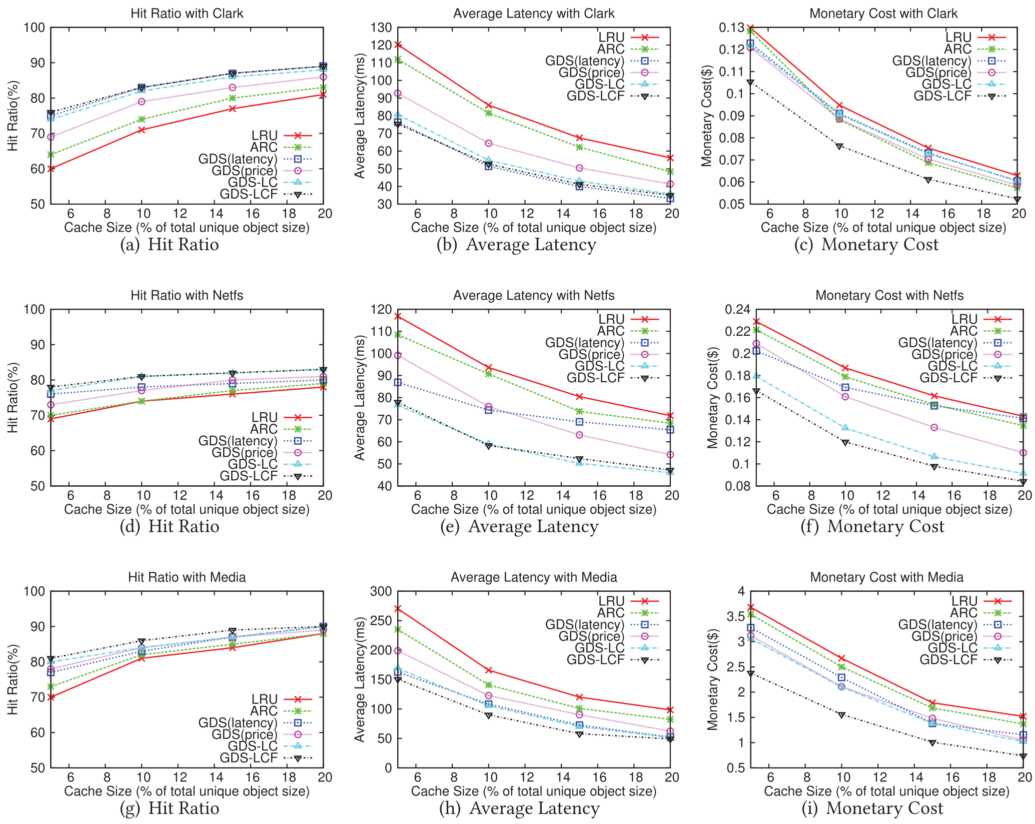
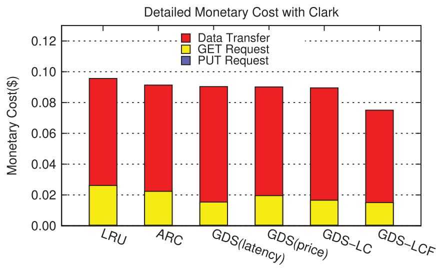


Fig. 9. Internet cloud: Hit ratios, average latencies, and monetary cost of *Clark*, *Netfs*, and *Media*.

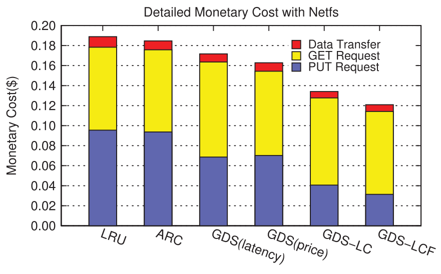
5.2.2 Internet Cloud. The Internet cloud system setup simulates a typical consumer cloud storage environment. In this case, the client locates on our campus in Louisiana and accesses cloud storage data stored in Amazon's Oregon data center. Different from the local cloud scenario, the RTT between the client and the cloud is high (113ms), and the price of data transfer is also more expensive. Figure 9 shows the results of different caching schemes in the Internet cloud scenario. Particularly, for *Clark* and *Media*, in which the requests are read intensive (dominated by `GET` requests), the data transfer fee is much higher than the request fee (see Figure 10(a) and (c)).

Similar to the results achieved in the local cloud scenario, GDS-LC and GDS-LCF perform the best, and LRU performs the worst. However, we can also find some differences caused by the distinct characteristics of the system setup in terms of latency and pricing policies in this scenario. The most obvious difference is about the results of the monetary cost. For the *Clark* workload, for example, the monetary cost of LRU is close to that of other caching schemes except GDS-LCF (see Figure 9(c)). Comparatively, in the local cloud scenario, significant gaps can be observed between the result of LRU and other caching schemes (see Figure 6(c)). This difference is caused by the charging of data transfer out from the cloud. As shown in Figure 10(a), compared to LRU, GDS-LC has a lower request fee but a much higher data transfer fee, so the gap between the overall monetary cost is narrowed down. The reason GDS-LC has a higher data transfer fee is that GDS-LC prefers to evict larger objects, leading to a larger data transfer traffic on cache misses.

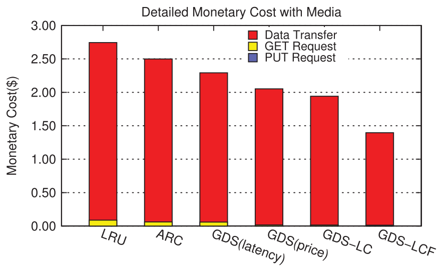
In addition, for the monetary cost charged with the *Netfs* workload, GDS-LC and GDS-LCF significantly outperform GDS(price) in the Internet cloud scenario (see Figure 9(f)); comparatively,



(a) Clark



(b) Netfs



(c) Media

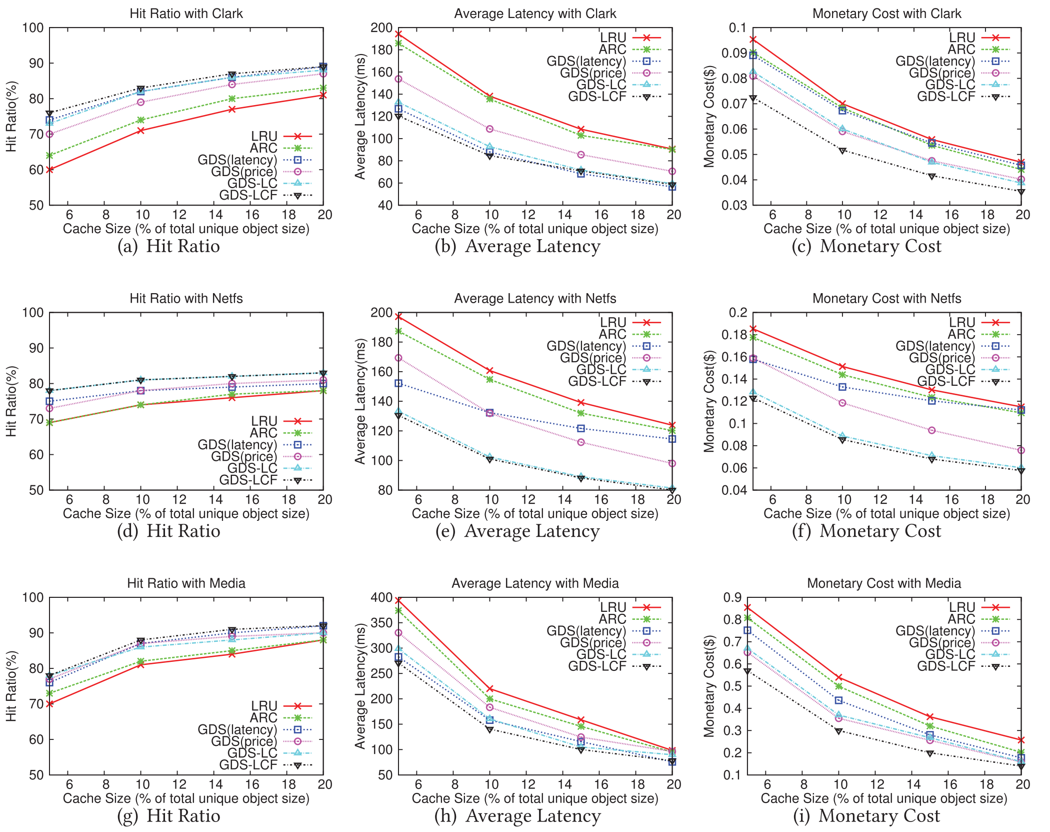
Fig. 10. Internet cloud: Detailed monetary cost of *Clark*, *Netfs*, and *Media* with the cache size set to 10% of the working set (i.e., the total size of unique objects). The monetary cost for accessing cloud objects includes data transfer fee and request fee (see Table 2 for details).

GDS-LC and GDS-LCF do not have obvious advantages over GDS(price) in the local cloud scenario (see Figure 6(f)). From Figure 10(b), we can see that the data transfer fee and GET request fee of GDS-LC and GDS(price) are close; the main difference comes from the PUT request fee. This is understandable. Since the RTT in this scenario is quite high, the dirty objects that are not synchronized by cache replacement cannot be quickly synchronized to the cloud; consequently, GDS-LC, which adds weight to dirty objects, has better caching efficiency for dirty objects. This explains why GDS-LC leads to a lower PUT request fee.

Again, the results achieved in this scenario demonstrate the merits of GDS-LC and GDS-LCF. For monetary cost, GDS-LCF performs better than GDS-LC; for average latency, the performance of GDS-LC and GDS-LCF are comparable, and both outperform other algorithms.

5.2.3 Heterogeneous Cloud. Heterogeneous cloud storage systems are generally adopted to exploit the advantages of multiple clouds. For example, RACS [1] adopts an RAID-like structure, which provides high-level data availability and reliability and prevents a vendor lock-in problem. Several other cloud-based storage systems, such as NCCloud [36] and DepSky [10], are also based on distributing data to multiple clouds. Another use case is to integrate different cloud storage services to uniformly access the storage space, especially for the purpose of utilizing the free tiers (e.g., the AWS Free Tier [5] and the Google Cloud Platform Free Tier [30]). In this case, the data may also be distributed to heterogeneous clouds.

In our experiments, to emulate the heterogeneous cloud storage system environment, we set up an EC2 instance in Amazon's Singapore data center as the client, which simultaneously connects to two cloud storage locating in Amazon's Oregon and Tokyo data centers. For each dataset, we evenly distribute the objects to these two data centers, organizing the data similar to RAID-0. An



(a) Hit Ratio (b) Average Latency (c) Monetary Cost

(d) Hit Ratio (e) Average Latency (f) Monetary Cost

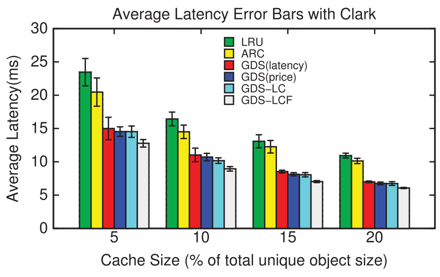
(g) Hit Ratio (h) Average Latency (i) Monetary Cost

Fig. 11. Heterogeneous cloud: Hit ratios, average latencies, and monetary cost of *Clark*, *Netfs*, and *Media*.

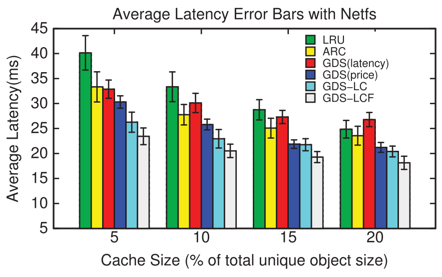
interesting fact is the difference of the pricing and network delay: due to a shorter geographic distance to the client, the Tokyo data center can provide a shorter latency (a 74ms RTT) for cloud storage I/Os than the Oregon data center (a 161ms RTT). However, its pricing on data transfers is significantly higher than the Oregon data center (see Table 2). The client caching scheme has to intelligently trade off and balance the two cloud storage sources for data accesses—for each eviction decision, it needs to choose either the closer but more expensive Tokyo data center or the farther but cheaper Oregon data center. This is particularly difficult for caching schemes.

As shown in Figure 11, our proposed GDS-LC and GDS-LCF caching schemes perform very well in this complicated scenario. LRU performs the worst, and GDS-LC can optimize both latency and monetary cost when being compared to the original GDS algorithms (i.e., GDS(latency) and GDS(price)); particularly in some cases, GDS-LC and GDS-LCF can perform much better than the GDS algorithms. These results well demonstrate the effect of our approach—our caching solution can well optimize both the latency and monetary cost in complicated environment.

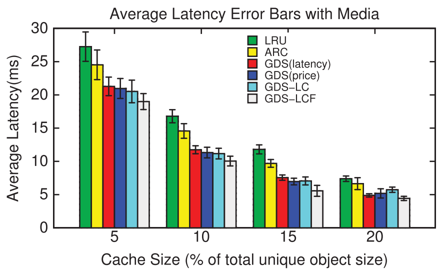
5.2.4 *Variance of Experimental Results.* In addition to system performance, we also examined the variance, which can be caused by the unexpected dynamics of network performance and cloud services. As presented in Section 5.1.4, we use the standard error of the values of each metric (i.e., hit ratio, average latency, and monetary cost) measured from five runs of the experiments to describe the variance. Since the observed variances of hit ratios and monetary cost are



(a) Clark



(b) Netfs



(c) Media

Fig. 12. Local cloud: Observed variances of average latencies.

insignificant, for brevity we only show the observed variances of the average latencies in Figures 12, 13, and 14.

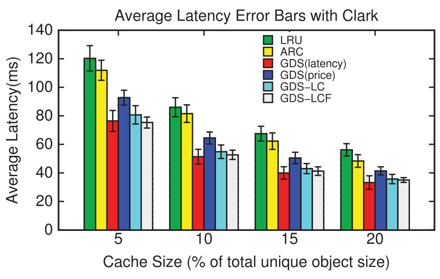
From the figures, we can see that the absolute values of the variances observed in the local cloud scenario are lower than those observed in the other two scenarios; however, we do not observe obvious differences with respect to the *relative* variances (i.e., the ratio of the variance and the average latency), which are about 5% to 10% in all working scenarios. As for the latency variances observed on the experimental results of different caching schemes, we find that when the hit ratios are relatively low, the variances of average latencies are relatively higher. That is because a lower hit ratio means that the client has to more frequently access the cloud and thus is more likely to subject to a larger variation of average access latencies. In particular, for LRU and ARC, the miss ratios of these caching schemes are higher than others. When the cache sizes are small, we can observe relatively larger variances on the average access latencies achieved by these two caching schemes.

It is worth noting that the discussions on the observed variances should be confined in the context of our experimental platform and the runs of our experiments; in other words, the comparisons are based on our observations and should be not be regarded as general conclusions.

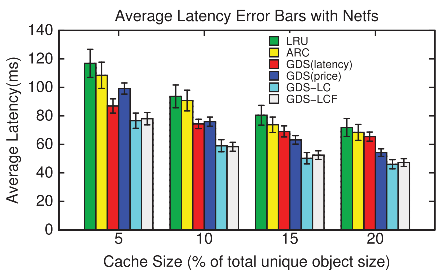
5.3 Sensitivity Study on Partition Size

Cache partitioning may influence the caching decision and its effectiveness. To evaluate the sensitivity of the GDS-LCF caching scheme to the cache partition size, in this experiment we run three workloads with the three system setups by using four different ratios of performance-to-price regions, specifically 1:2, 1:1, 2:1, and 1:3. For brevity, we use the Internet cloud scenario to illustrate the effect of cache partitioning.

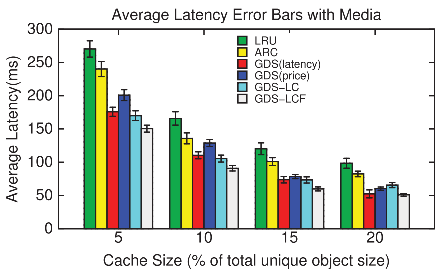
As shown in Figure 15, we can see that the effect of cache partitioning is workload dependent. In *Clark* and *Media*, the three partition ratios have a relatively weak impact on the observed latencies,



(a) Clark

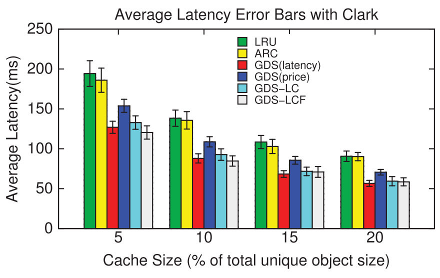


(b) Netfs

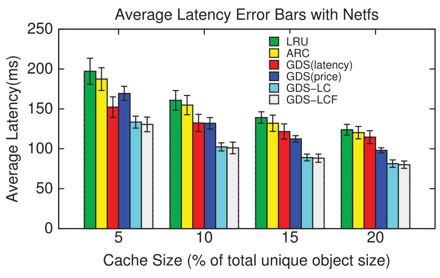


(c) Media

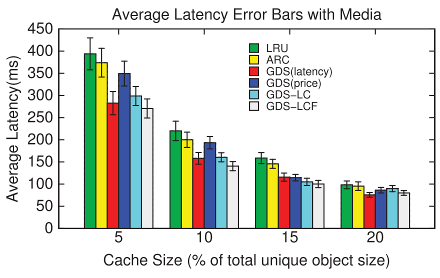
Fig. 13. Internet cloud: Observed variances of average latencies.



(a) Clark



(b) Netfs



(c) Media

Fig. 14. Heterogeneous cloud: Observed variances of average latencies.

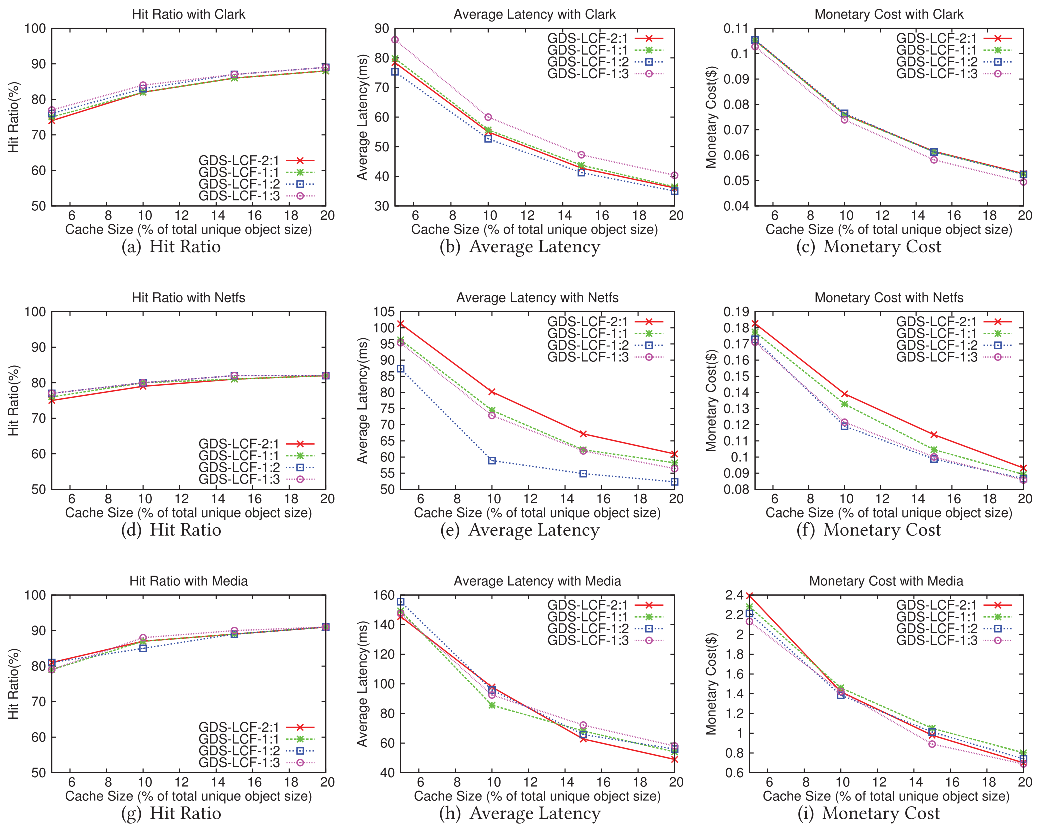
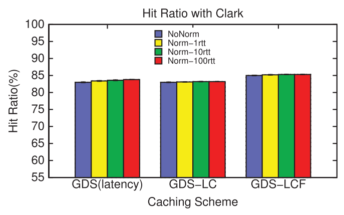


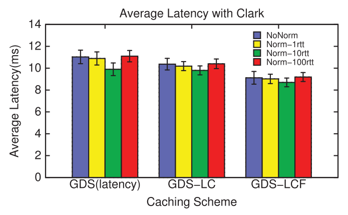
Fig. 15. Internet cloud: Evaluation of the effects of different size ratios of the performance region and the cost region. Shown in the figures are hit ratios, average latencies, and monetary cost of *Clark*, *Netfs*, and *Media* achieved by our caching scheme *GDS-LCF* with different size ratios.

hit ratio, and cost. In contrast, the *Netfs* workload exhibits certain distinctions. Generally, the ratio 1:2 is a reasonably sound choice to effectively reduce both access latency and monetary cost (see Figure 15(e) and (f)). Particularly, compared to the ratio 1:3, the ratio 1:2 can achieve comparable monetary cost but significantly lower average latency. Thus, the 1:2 ratio is a proper choice.

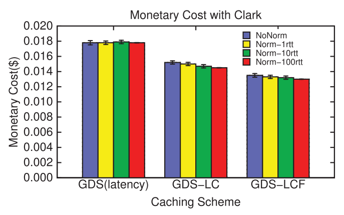
Interestingly, we also note that a larger performance region does not necessarily result in a lower average latency. As shown in Figure 15(e), for example, when the cache size is 10% of the working set, increasing the performance region from one fourth of the cache size (with the ratio 1:3) to one third of the cache size (with the ratio 1:2), the average latency decreases from 74 to 59 ms; however, further increasing the performance partition to two thirds of the cache size (with the ratio 2:1), the average latency increases from 59 to 80 ms. The effect is caused by the object migration between the two regions (see Section 4.2). On the one hand, a larger performance region means that the objects that have high values in terms of latency are more likely to be kept in the local cache. On the other hand, a larger performance region leads to a smaller price region, which means that the objects demoted to the price region may be quickly evicted from the local cache and thus have less opportunities to be promoted to the performance region again upon a second access. Consequently, the relationship between the latency and the size of the performance region is not a simple linear function.



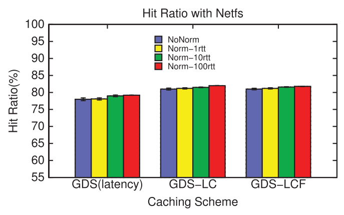
(a) Hit Ratio



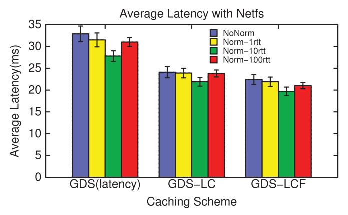
(b) Average Latency



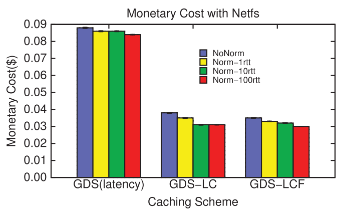
(c) Monetary Cost



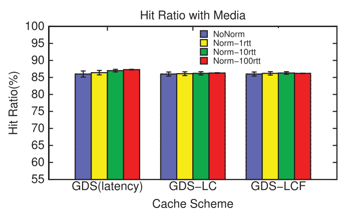
(d) Hit Ratio



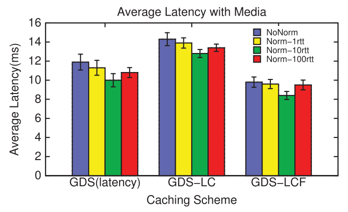
(e) Average Latency



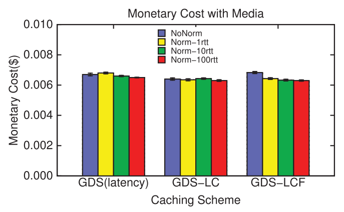
(f) Monetary Cost



(g) Hit Ratio



(h) Average Latency



(i) Monetary Cost

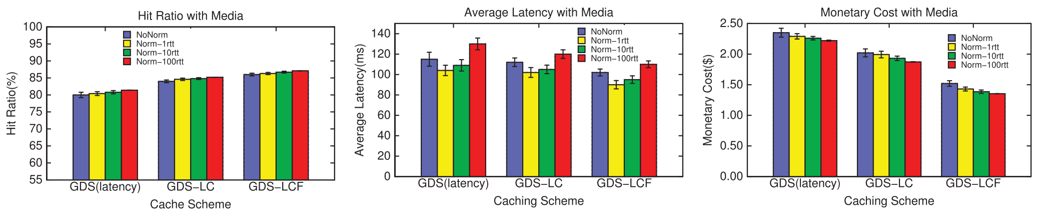
Fig. 16. Local cloud: The performance of GDS(latency), GDS-LC, and GDS-LCF with different normalization factors for *Clark*, *Netfs*, and *Media*. The cache size is set to 10% of the working set.

5.4 Impact of Latency Normalization

As analyzed in Section 4.2.1, the latency variance for accessing cloud storage may affect the efficiency of latency-aware caching schemes, and we adopt an adaptive normalization approach based on the observed RTT between the client and the cloud to alleviate this problem. In this section, we further discuss the impact of normalization.

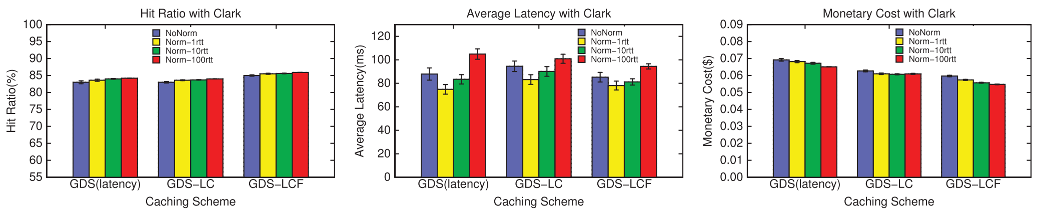
To evaluate the effects of the normalization approach, we conduct a set of experiments with our proposed GDS-LC and GDS-LCF and the GDS(latency). We set four different normalization levels: (1) *NoNorm*, directly using the absolutely value of latency; (2) *Norm-1rtt*, using 1x RTT between the client and the cloud as the normalization factor; (3) *Norm-10rtt*, normalizing with 10x RTT; and (4) *Norm-100rtt*, normalizing with 100x RTT. We run the experiments 10 times, report the average value of each metric, and calculate the standard error as the variance (see Section 5.1.4).

Figure 16 shows the experiments in the local cloud scenario, and the cache size is set to be 10% of the working set. From the figure, we can see that the hit ratios increase as the values of the normalization factors increase; however, the increase of hit ratios does not always lead to lower average latencies. As shown in the figures, we find that setting the normalization factor to be 10x RTT achieves the best performance among the settings; meanwhile, setting the normalization factor as 1x RTT brings trivial benefits and 100x RTT may reduce the benefits. That is because the local cloud has low RTT (i.e., 0.28ms). With setting the normalization factor to be 1x RTT, the interference of the latency variance cannot be effectively reduced, whereas setting the normalization



(a) Hit Ratio (b) Average Latency (c) Monetary Cost

Fig. 17. Internet cloud: The performance of GDS(latency), GDS-LC, and GDS-LCF with different normalization factors for *Media*. The cache size is set to be 10% of the working set.



(a) Hit Ratio (b) Average Latency (c) Monetary Cost

Fig. 18. Heterogeneous cloud: The performance of GDS(latency), GDS-LC, and GDS-LCF achieved by different normalization factors for *Clark*. The cache size is set to be 10% of the working set.

factor to 100x RTT (i.e., 28ms) normalizes the latencies of many objects to 1, which may decrease the overall system performance. As for the impact of normalization on different caching schemes, we note that the impact of normalization on GDS(latency) is more significant than that on GDS-LC and GDS-LCF. This is because only the top region in the design of GDS-LC and GDS-LCF adopts the latency-aware caching scheme and the object evicted from the top region will be migrated to the second region and still has the opportunity to be fetched back instead of being immediately evicted from the local cache, which makes them less sensitive to normalization than GDS(latency).

We also note that in the scenarios of the Internet cloud and heterogeneous cloud, the normalization factor 10x RTT can still achieve better performance than no normalization, but 1x RTT performs better. The performance achieved by different caching schemes with the *Media* trace is shown in Figure 17: when the normalization factor is larger than one RTT, the benefit brought by normalization is diminishing; when the normalization factor is 100x RTT, the aggressive normalization approach leads to performance loss. In the Internet cloud scenario, the RTT is 113ms, thus setting the normalization factor to be 100x RTT (i.e., 11.3 seconds) means that almost all latencies of the objects are normalized to 1. In this case, although the hit ratio is improved, the overall system performance decreases. Similarly, in the heterogeneous cloud scenario, the minimum RTT between the two (74 and 161 ms) is 74ms, and setting the normalization factor to be higher than 10x RTT may cause negative effects (i.e., setting the normalization factor to be 100x RTT). Shown in Figure 18 is the performance achieved by different caching schemes with the *Clark* trace, which indicates that setting the normalization factor to be 1x RTT performs the best among the settings.

Therefore, based on our observations, a proper normalization factor varies with different working scenarios. In our platform, we find 10x RTT is a good choice for the scenario in which the client and the cloud are in the same data center. A smaller normalization factor (e.g., 1x RTT) is good for the scenarios in which the clients access the cloud across data centers, where the RTT between the client and the cloud is a relatively larger value. Setting the normalization factor to an excessively large one (e.g., 100x RTT) is generally undesirable, as it removes the capability of differentiating access costs. It is also worth noting that the negative effects of latency variance to cost-aware caching schemes cannot be completely eliminated due to the difficulty of accurately predicting latency variance. In our proposed caching schemes, we attempt to reduce the interference of latency variance by using an adaptive normalization approach. In practice, we can further improve the accuracy of cost evaluations with the knowledge of the performance behaviors of cloud storage services and the variance of network services, which can be gained by long-term observations. For example, if we know that an object will be reloaded during the "busy hours" of the target cloud storage (at the time when the cloud is busy with handling intensive requests, leading to longer response time), the cost of evicting the object should be estimated higher than the download latency measured beyond the busy hours.

5.5 Further Evaluation on GDS-LCF

By introducing frequency into the cost functions of GDS-LC, GDS-LCF gives higher caching priority to frequently accessed objects. As shown in Section 5.2, GDS-LCF outperforms traditional caching schemes (i.e., LRU and ARC) and GDS with different settings (i.e., GDS(latency) and GDS(price)) and can successfully improve the caching efficiency of GDS-LC in most cases. In this section, we further compare GDS-LCF to the frequency-enhanced version of GDS called *GDSF* and discuss the enhancement.

Both GDS-LCF and GDSF are enhanced by introducing frequency. GDS-LCF is an enhanced version of GDS-LC, and GDSF is an enhanced version of GDS. We expect that the advantage of GDS-LCF over GDSF is similar to the advantage of GDS-LC over GDS: for read-intensive workloads, GDS-LCF can optimize both performance and monetary cost instead of only one optimization goal; for write-intensive workloads, GDS-LCF can significantly outperform GDSF, as the former has a two-region design and can also differentiate the cost of evicting clean objects and dirty objects.

To verify our speculation, we implement GDSF, a frequency-enhanced version of GDS. As for the frequency approximation, we count frequency to at most 4. We also test other approximation methods, for example, counting frequency to at most 8, 16, or higher. We find that counting frequency to at most 4 achieves comparable performance as other methods. Setting the optimization goals as latency and monetary cost, respectively, we get two versions of GDSF: GDSF(latency) and GDSF(price).

Figure 19 shows the performance comparison of GDS-LCF, GDSF(latency), and GDSF(price) with different traces in the Internet cloud scenario. The experimental results have confirmed our speculation: for the read-intensive traces *Clark* and *Media*, GDS-LCF can achieve comparable average latency to GDSF(latency) and comparable monetary cost to GDSF(price), successfully optimizing both goals; for the write-intensive trace *Netfs*, GDS-LCF have much better performance than GDSF(latency) and GDSF(price).

5.6 Comparisons to GDS Enhanced With Clean-Dirty Differentiation

As stated in Section 4.2, a significant difference between the latency functions used in our caching schemes (i.e., GDS-LC and GDS-LCF) and the original GDS-based policies (i.e., GDS(latency) and GDSF) is that our latency functions have the capability of distinguishing clean and dirty objects. Since in prior sections we compared our caching schemes to the original GDS-based policies, in this section we further compare our caching schemes to the improved GDS-based policies, which have the same latency functions as our caching schemes, called *GDS-L* and *GDS-LF*. Compared to GDS-L and GDS-LC, GDS-L and GDS-LF do not have a price region and take all cache space as the performance region. Particularly, GDS-LF counts the access frequency at most four in its cost functions, which is the same as that of GDSF (see Section 5.5).

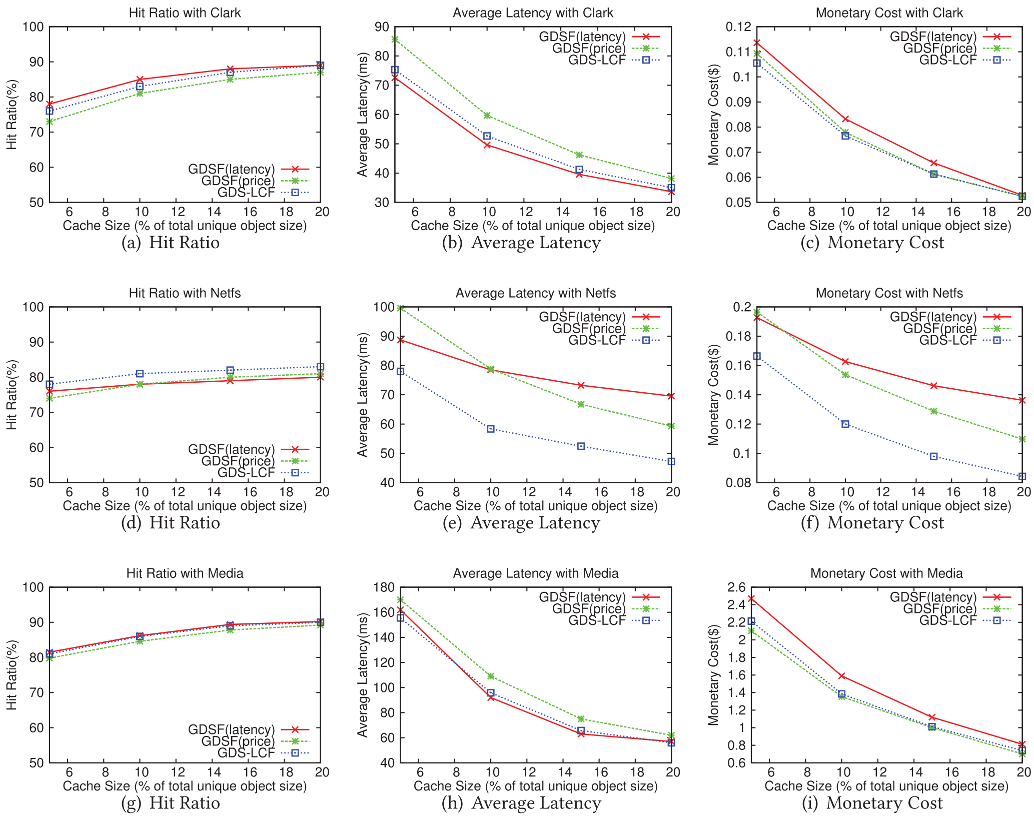


Fig. 19. Internet cloud: Comparisons of *GDS-LCF*, *GDSF(latency)*, and *GDSF(price)*.

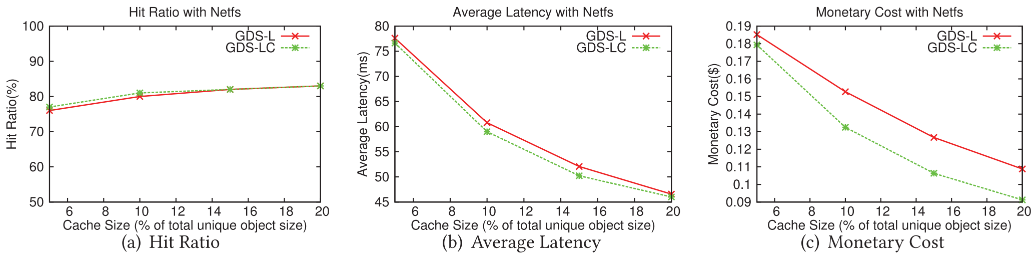


Fig. 20. Internet cloud: Comparisons of GDS-LC and GDS-L.

Since no victim objects are dirty when working with the *Clark* trace and the *Media* trace (note that *Clark* is highly read intensive and a write-through policy is adopted and *Media* is read only), we use the *Netfs* trace in the experiments. Figure 20 shows the experimental results of GDS-L and GDS-LC in the Internet cloud scenario. Since GDS-LC has a price region to keep high-price objects, it can significantly reduce the monetary cost. With respect to access latency, compared to GDS-L, GDS-LC achieves comparable (even slightly better) performance. Although GDS-LC reserves two-thirds of the cache space as the price region, the objects that have the highest cost in terms of access latency are kept in the performance region, and the objects demoted to the price region still have opportunities to be fetched back to the performance region; thus, GDS-LC can

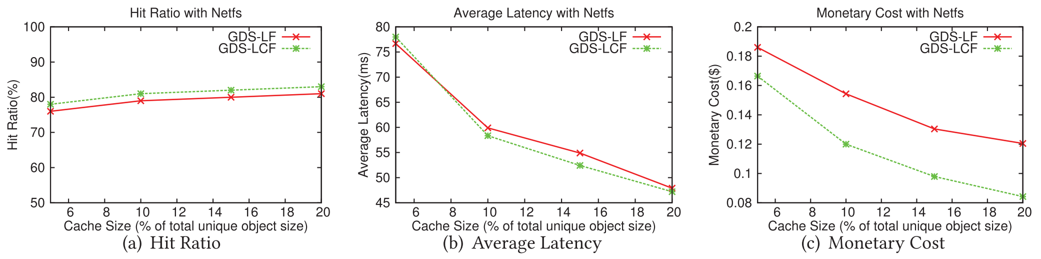


Fig. 21. Internet cloud: Comparisons of GDS-LCF and GDS-LF.

achieve comparable average latency even with a smaller performance region. For similar reasons, compared to GDS-LF, GDS-LCF achieves lower monetary cost and comparable average latency (Figure 21). The experimental results further demonstrate the advantages of our two-region design.

6 OTHER RELATED WORK

Both cloud storage and cache replacement algorithms have received extensive studies. In this section, we present other prior work most related to this article.

Cloud storage systems. Cloud storage recently has attracted a lot of research attention. A variety of issues of cloud storage systems have been studied, such as performance, reliability, availability, confidentiality, and service lock-in concerns [1, 7, 12, 26, 31, 36, 76]. Much research has been conducted to characterize the performance and I/O behaviors of cloud storage [8, 22, 23, 33–35, 47, 57, 72]. Our work is orthogonal to these studies.

Cloud-based file systems. For easy use of cloud storage, prior research has also attempted to unify the I/O interfaces of cloud storage and file systems. For example, a cloud-backed network file system for the enterprise use, called *BlueSky* [71], stores data in cloud storage and accesses storage through an on-site proxy, which caches data and supports multiple protocols including NFS and CIFS. Another similar network file system design, called *RFS* [21], is proposed for mobile devices. SCFS [9] provides a POSIX-like interface on top of cloud storage. Similarly, S3FS [63] also provides simple file system–like interfaces for Amazon Simple Storage Services (S3). These solutions typically adopt an LRU-based caching scheme on local clients or proxies. Our work focuses particularly on caching schemes and can potentially enhance these systems.

Commercial cloud-based storage products. Our work is also related to the caching algorithms adopted by the commercial cloud storage products, including cloud proxies and gateways (e.g., Nasuni [53], Twinstrata [69], CTERA [20], Panzura [58], and StorSimple [66]). These products mainly provide storage accelerating services, acting as a cloud-based cache between user applications and remote clouds. Although the implementation details of these products are not publicly available, according to open documents, LRU is the most popular caching algorithm adopted by the majority of these products [50, 54, 55, 59]. Our work aims at optimizing the cloud-based storage systems with cost-aware caching by considering various factors, including access latency, price, object size, and access recency and frequency, and can be flexibly applied in these working scenarios to improve user experience in terms of not only performance but also monetary cost.

Cost-aware caching. Recent studies have studied cost-aware caching in different working scenarios for different proposes. Jiang et al. [40] presented an OS kernel buffer cache management scheme, called *DULO*. DULO leverages the speed distinction of random and sequential I/Os on hard disk drives and gives higher caching priority to the blocks that are randomly accessed, since random accesses are slower than sequential accesses on hard disk drives. Similarly, Li and Cox [44] customized and also proposed a caching scheme based on GDS [14], called *GD-Wheel*, in the scenario of key-value stores by considering recomputing latency as cost. Kim and Anh [42] presented a caching scheme, called *BPLRU*, for improving random writes in flash storage. PS-BC [16] leverages the filtering effect of OS buffer cache to create bursty disk I/Os for disk power saving. Forney et al. [27] introduced a set of storage-aware caching algorithms that partition the file buffer for heterogeneous storage and dynamically tune the partition sizes to balance the workloads across the storage devices. Liang et al. [46] studied caching replacement policies for distributed storage systems and proposed two offline heuristics and an online algorithm by considering access latencies as the major cost when deciding the victim data. Araldo et al. [6] proposed two optimization models that either minimize the overall costs or maximize the hit ratio, jointly considering cache sizing, object placement, and path selection, and taking the retrieval latency as the cost in the scenario of information-centric networks (ICNs). Jeong and Dubois [38] made several extensions of LRU, taking into account nonuniform miss costs (e.g., the latency, penalty, power consumption, bandwidth consumption, or any other ad hoc numerical property attached to a miss) in different practical cases, such as multiprocessor memory systems and single superscalar processor systems.

Although sharing a similar design principle with these solutions by leveraging cost awareness in caching decisions, our solution particularly aims to enhance caching for cloud storage, which shows distinct properties compared to other systems. In particular, its special performance behaviors and pricing models demand us to focus on improving the user experience with regard to both access latency and monetary cost. Additionally, our caching scheme is designed for using cloud storage as primary storage. In this scenario, I/O accesses are both read and write intensive, requiring us to fully consider the access time of handling both clean and dirty data rather than one-direction cloud I/Os. As well, different from prior schemes that only consider the cost from only one aspect (e.g., latency, bandwidth, or energy), we aim at minimizing the cost from two orthogonal dimensions (latency and price) at the same time.

Other advanced caching options. In addition to cost-aware caching algorithms, some other advanced caching optimizations have been introduced to improve caching efficiency. These caching schemes can be roughly classified into four categories. The first category is leveraging application-level hints. For example, application-controlled file caching [13] and informed prefetching and caching [60] are motivated by making use of hints from applications. Recently, for the purpose of improving cloud storage performance, Chen et al. [15] presented a solution called *client-aware cloud storage*, which further leverages client-provided semantic hints to enhance server-side caching. (2) The second category is tracing and utilizing history information. Some advanced caching algorithms (e.g., CLOCK-Pro [39], ARC [49], LIRS [41], and Multi-Queue [77]) utilize access history to assist cache replacement and thus outperform LRU, which only takes recency in its caching consideration. Most of these schemes leverage a deep caching history to identify weak-locality and low-frequency data for improving cache performance. The third category is exploiting access patterns and data correlations. Prior studies have also exploited the access patterns from history information to direct caching for virtual memory management and buffer management (e.g., Choi et al. [17, 18], Glass and Cao [28], and Kim et al. [43]). In addition to these online methods, data semantic relationships have also been exploited to improve caching efficiency. For example, Li et al. [45] proposed a data mining scheme working in storage systems to obtain block-level correlations that can be used to direct caching and prefetching. Some Web mining (e.g., Nanopoulos et al. [52] and Yang et al. [73]) and file correlations algorithms (e.g., Eaton et al. [25]) have also been leveraged to optimize caching in Web services and file systems. The fourth category is partitioning the shared cache. Cache partitioning is a typical method used to deal with the cache sharing issues, working in the scenarios of sharing cache between multiple applications or heterogeneous storage devices. For example, in prior work [27], the cache is partitioned for heterogeneous storage systems, and each storage device is assigned a partition. Qureshi and Patt [61] also proposed a method that partitions a shared cache between multiple applications to reduce cache misses for a given amount of cache resources. Dynamic partitioning of shared cache memory is also used to assign cache resources for simultaneously executing processes or threads that can be applied to set-associative caches [67].

Our solution is largely orthogonal to these classic caching approaches. The key idea of our solution is to address the unique requirements in the cloud storage scenario and take both performance and monetary cost into consideration with a cost-aware caching algorithm. We do not rely on application-level hints, history information, or extra knowledge gained through data mining or machine learning methods; however, our solution can be flexibly integrated with other optimization methods. For example, each partition of a shared cache can be managed with our caching scheme. In fact, as a special case, we have demonstrated the effectiveness and efficiency of GDS-LCF, which is an integration of our basic scheme GDS-LC with another caching factor, *frequency*. It would be an interesting and practically valuable research topic to investigate how to properly integrate these advanced caching schemes within our solution, which we leave as our future work.

7 CONCLUSIONS

Client caching is crucial to truly integrating cloud storage as a primary storage layer in computer systems. By keeping the most valuable objects in the local cache and evicting the least important ones, client caching policies can influence future accesses. Leveraging such a filtering effect, we design two unique caching schemes, called *GDS-LC* and *GDS-LCF*, with an attempt to minimize future access latency and monetary cost. Compared to traditional caching schemes, our experimental results show that our solution can effectively improve system performance and reduce system cost.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback and insightful comments.

REFERENCES

[1] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. 2010. RACS: A case for cloud storage diversity. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*.

[2] Soam Acharya and Brian Smith. 2000. Middleman: A video caching proxy server. In *Proceedings of the 10th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV'00)*.

[3] Amazon. 2016. Amazon S3 Pricing. Retrieved October 17, 2017, from https://aws.amazon.com/s3/pricing/.

[4] Amazon. 2016. Netflix Case Study. Retrieved October 17, 2017, from https://aws.amazon.com/solutions/case-studies/netflix/.

[5] Amazon. 2017. AWS Free Tier. Retrieved October 17, 2017, from https://aws.amazon.com/free/.

[6] Andrea Araldo, Michele Mangili, Fabio Martignon, and Dario Rossi. 2014. Cost-aware caching: Optimizing cache provisioning and object placement in ICN. In *Proceedings of the 2014 IEEE Global Communications Conference (GLOBECOM'14)*.

[7] Sobir Bazarbayev, Matti Hiltunen, Kaustubh Joshi, Richard Schlichting, and William Sanders. 2013. PSCloud: A durable context-aware personal storage cloud. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep'13)*.

[8] Ignacio Bermudez, Stefano Traverso, Marco Mellia, and Maurizio Munafo. 2013. Exploring the cloud from passive measurement: The Amazon AWS case. In *Proceedings of the 32nd Annual IEEE International Conference on Computer Communications (INFOCOM'13)*.

[9] Bessani, Alysson, Ricardo Mendes, Tiago Oliveira, Nuno Neves, Miguel Correia, Marcelo Pasin, and Paulo Verissimo. 2014. SCFS: A shared cloud-backed file system. In *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'14)*.

[10] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. 2013. DepSky: Dependable and secure storage in a cloud-of-clouds. *ACM Transactions on Storage* 9, 4, 1–33.

[11] BlueCoat. 2017. Executive Summary. Retrieved October 17, 2017, from https://www.bluecoat.com/sites/default/files/documents/files/Object_Caching.1.pdf.

[12] Nicolas Bonvin, Thanasis G. Papaioannou, and Karl Aberer. 2010. A self-organized, fault-tolerant and scalable replication scheme for cloud storage. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*.

[13] Pei Cao, Edward W. Felten, and Kai Li. 1994. Application-controlled file caching policies. In *Proceedings of the 1994 USENIX Summer Technical Conference (USTC'94)*.

[14] Pei Cao and Sandy Irani. 1997. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems (USITS'97)*.

[15] Feng Chen, Michael P. Mesnier, and Scott Hahn. 2014. Client-aware cloud storage. In *Proceedings of the 30th International Conference on Massive Storage Systems and Technology (MSST'14)*.

[16] Feng Chen and Xiaodong Zhang. 2010. PS-BC: Power-saving considerations in design of buffer caches serving heterogeneous storage devices. In *Proceedings of the 2010 International Symposium on Low Power Electronics and Design (ISLPED'10)*.

[17] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 1999. An implementation study of a detection-based adaptive block replacement scheme. In *Proceedings of the 1999 Annual USENIX Technical Conference (ATC'99)*.

[18] Jongmoo Choi, Sam H. Noh, Sang Lyul Min, and Yookun Cho. 2000. Towards application/file-level characterization of block references. In *Proceedings of the 2000 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems (SIGMETRICS'00)*.

[19] ClarkNet. 2016. ClarkNet-HTTP. Retrieved October 17, 2017, from http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html.

[20] CTERA. 2017. Home Page. Retrieved October 17, 2017, from http://www.ctera.com/.

[21] Yuan Dong, Jinzhan Peng, Dawei Wang, Haiyang Zhu, Fang Wang, Sun C. Chan, and Michael P. Mesnier. 2011. RFS—a network file system for mobile devices and the cloud. *ACM SIGOPS Operating Systems Review* 45, 1, 101–111.

[22] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. 2013. Benchmarking personal cloud storage. In *Proceedings of the 2013 ACM Internet Measurement Conference (IMC'13)*.

[23] Idilio Drago, Marco Mellia, Maurizio M. Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. 2012. Inside Dropbox: Understanding personal cloud storage services. In *Proceedings of the 2012 ACM Internet Measurement Conference (IMC'12)*.

[24] Dropbox. 2016. Home Page. Retrieved October 17, 2017, from https://www.dropbox.com/.

[25] Patrick R. Eaton, Dennis Geels, and Greg Mori. 1999. Clump: Improving File System Performance Through Adaptive Optimizations. Available at http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.23.227.

[26] D. Ford, F. Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. 2010. Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*.

[27] Brian C. Forney, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2002. Storage-aware caching: Revisiting caching for heterogeneous storage systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'02)*.

[28] Gideon Glass and Pei Cao. 1997. Adaptive page replacement based on memory reference behavior. In *Proceedings of the 1997 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems (SIGMETRICS'97)*.

[29] Google. 2016. Home Page. Retrieved October 17, 2017, from https://www.google.com/drive/.

[30] Google. 2017. Google Cloud Platform Free Tier: Always Free Usage Limits. Retrieved October 17, 2017, from https://cloud.google.com/free/docs/always-free-usage-limits.

[31] Ajay Gulati, Ganesha Shanmuganathan, Irfan Ahmad, Carl Waldspurger, and Mustafa Uysal. 2011. Pesto: Online storage performance management in virtualized datacenters. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SoCC'11)*.

[32] Flex Hartanto, Jussi Kangasharju, Martin Reisslein, and Keith W. Ross. 2002. Caching video objects: Layers vs versions? In *Proceedings of the 2002 IEEE International Conference on Multimedia and Expo (ICME'02)*.

[33] Binbing Hou, Feng Chen, Zhonghong Ou, Ren Wang, and Michael Mesnier. 2016. 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)*.

[34] Binbing Hou, Feng Chen, Zhonghong Ou, Ren Wang, and Michael Mesnier. 2017. Understanding I/O performance behaviors of cloud storage from a client's perspective. *ACM Transactions on Storage* 13, 2, 16:1–16:36.

[35] Wenjin Hu, Tao Yang, and Jeanna N. Matthews. 2010. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review* 44, 3, 110–115.

[36] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. 2012. NCCloud: Applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12).*

[37] InTheCloud. 2016. Spotify Moving Onto Google Cloud Is A Big Win for Google Over Amazon and Microsoft. Retrieved October 17, 2017, from https://www.forbes.com/sites/alexkonrad/2016/02/23/spotify-is-a-big-win-for-google-cloud/#49cc582374b9.

[38] Jaeheon Jeong and Michel Dubois. 2003. Cost-sensitive cache replacement algorithms. In *Proceedings of the 9th International Symposium on High Performance Computer Architecture (HPCA'03).*

[39] Song Jiang, Feng Chen, and Xiaodong Zhang. 2005. CLOCK-pro: An effective improvement of the CLOCK replacement. In *Proceedings of the 2005 USENIX Annual Technical Conference (ATC'05).*

[40] Song Jiang, Xiaoning Ding, Feng Chen, Enhua Tan, and Xiaodong Zhang. 2005. 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).*

[41] Song Jiang and Xiaodong Zhang. 2002. LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance. In *Proceedings of the 2002 International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'02).*

[42] Hyojun Kim and Seongjun Ahn. 2008. BPLRU: A buffer management scheme for improving random writes in flash storage. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST'08).*

[43] Jong Min Kim, Jongmoo Choi, Jesung Kim, Sam H. Noh, Sang Lyul Min, Yookun Cho, and Chong Sang Kim. 2000. A low-overhead high-performance unified buffer management scheme that exploits sequential and looping references. In *Proceedings of the 4th Conference Symposium on Operating System Design and Implementation (OSDI'04).*

[44] Conglong Li and Alan L. Cox. 2015. GD-wheel: A cost-aware replacement policy for key-value stores. In *Proceedings of the 10th European Conference on Computer Systems (EuroSys'15).* ACM, New York, NY.

[45] Zhenmin Li, Zhifeng Chen, Sudarshan M. Srinivasan, and Yuanyuan Zhou. 2004. C-miner: Mining block correlations in storage systems. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies (FAST'04).*

[46] Shuang Liang, Ke Chen, Song Jiang, and Xiaodong Zhang. 2007. Cost-aware caching algorithms for distributed storage servers. In *Proceedings of 21st International Symposium on Distributed Computing (DISC'07).*

[47] Thomas Mager, Ernst Biersack, and Pietro Michiardi. 2012. A measurement study of the Wuala on-line storage service. In *Proceedings of the 12th IEEE International Conference on Peer-to-Peer Computing (P2P'12).*

[48] Richard McDougall, Joshua Crase, and Shawn Debnath. 2005. FileBench. Retrieved October 17, 2017, from http://sourceforge.net/projects/filebench.

[49] N. Megiddo and D. Modha. 2003. ARC: A self-tuning, low overhead replacement cache. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03).*

[50] Microsoft. 2016. *Cache Configuration.* ESG Microsoft Azure StorSimple White Paper. Microsoft. Available at http://download.microsoft.com/download/8/B/3/8B3F84CD-DBEE-483D-943A-936752AD4516/ESG_Microsoft_Azure_StorSimple_White_paper.pdf.

[51] Microsoft. 2016. OneDrive. Retrieved October 17, 2017, from https://onedrive.live.com/.

[52] Alexandros Nanopoulos, Dimitrios Katsaros, and Yannis Manolopoulos. 2003. A data mining algorithm for generalized Web prefetching. *IEEE Transactions on Knowledge and Data Engineering* 15, 5, 1155–1169.

[53] Nasuni. 2016. Home Page. Retrieved October 17, 2017, from https://www.nasuni.com/.

[54] Nasuni. 2016. Nasuni Cache Configuration. Retrieved October 17, 2017, from http://www6.nasuni.com/rs/445-ZDB-645/images/CacheConfig.pdf.

[55] NetApp. 2016. NetApp SteelStore Cloud Integrated Storage 3.2: Deployment Guide. Retrieved October 17, 2017, from https://library.netapp.com/ecm/ecm_download_file/ECMP12031272.

[56] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K page replacement algorithm for database disk buffering. In *Proceedings of the 1993 ACM International Conference on Management of Data (SIGMOD'93).*

[57] Zhonghong Ou, Zhen-Huan Hwang, Antti Ylä-Jääski, Feng Chen, and Ren Wang. 2015. Is cloud storage ready? A comprehensive study of IP-based storage systems. In *Proceedings of the 8th IEEE/ACM International Conference on Utility and Cloud Computing (UCC'15).*

[58] Panzura. 2016. Home Page. Retrieved October 17, 2017, from http://panzura.com/.

[59] Panzura. 2016. Panzura Debuts Version 3.0 of Its Global Cloud Storage System. Retrieved October 17, 2017, from http://panzura.com/press-releases/panzura-debuts-version-3-0-of-its-global-cloud-storage-system/.

[60] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, and Jim Zelenka. 1995. Informed prefetching and caching. In *Proceedings of the 15th Symposium on Operating System Principles (SOSP'95).*

[61] Moinuddin K. Qureshi and Yale N. Patt. 2006. Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches. In *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06).*

[62] S3Backer. 2016. Google Code Archive: S3Backer. Retrieved October 17, 2017, from https://code.google.com/p/s3backer/.

[63] S3FS. 2016. Google Code Archive: S3FS. Retrieved October 17, 2017, from https://code.google.com/p/s3fs/.

[64] Muhammad Zubair Shafiq, Alex X. Liu, and Amir R. Khakpour. 2014. Revisiting caching in content delivery networks. *ACM SIGMETRICS Performance Evaluation Review* 42, 1, 567–568.

[65] StorageServers. 2013. Dropbox Uses Amazon S3 Services for Storage! Retrieved October 17, 2017, from https://storageservers.wordpress.com/2013/10/25/dropbox-uses-amazon-s3-services-forstorage/.

[66] StorSimple. 2016. Microsoft Azure: StorSimple. Retrieved October 17, 2017, from https://www.microsoft.com/en-us/cloud-platform/azure-storsimple.

[67] G. Edward Suh, Larry Rudolph, and Srinivas Devadas. 2004. Dynamic partitioning of shared cache memory. *Journal of Supercomputing* 28, 1, 7–26.

[68] Wenting Tang, Yun Fu, Ludmila Cherkasova, and Amin Vahdat. 2003. Medisyn: A synthetic streaming media service workload generator. In *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV'03).*

[69] TwinStrata. 2016. EMC CloudArray. Retrieved October 17, 2017, from http://www.emc.com/domains/cloudarray/.

[70] VideoCache. 2017. Home Page. Retrieved October 17, 2017, from https://cachevideos.com/.

[71] Michael Vrable, Stefan Savage, and Geoffrey M. Voelker. 2012. BlueSky: A cloud-backed file system for the enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST'12).*

[72] Haiyang Wang, Ryan Shea, Feng Wang, and Jiangchuan Liu. 2012. On the impact of virtualization on Dropbox-like cloud file storage/synchronization services. In *Proceedings of the 20th International Workshop on Quality of Service (IWQoS'12).*

[73] Qiang Yang, Haining Henry Zhang, and Tianyi Li. 2001. Mining Web logs for prediction models in WWW caching and prefetching. In *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (SIGKDD'01).*

[74] Suli Yang, Kiran Srinivasan, Kishore Udayashankar, Swetha Krishnan, Jingxin Feng, Yupu Zhang, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Tombolo: Performance enhancements for cloud storage gateways. In *Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST'16).*

[75] Neal Young. 1994. The K-server dual and loose competitiveness for paging. *Algorithmica* 11, 6, 525–541.

[76] Rui Zhang, Ramani Routray, David Eyers, David Chambliss, Prasenjit Sarkar, Douglas Willcocks, and Peter Pietzuch. 2011. IO Tetris: Deep storage consolidation for the cloud via fine-grained workload analysis. In *Proceedings of the 4th IEEE International Conference on Cloud Computing (CLOUD'11).*

[77] Yuanyuan Zhou, James F. Philbin, and Kai Li. 2001. The multi-queue replacement algorithm for second level buffer caches. In *Proceedings of the 2001 USENIX Annual Technical Conference (ATC'01).*

Received August 2016; revised September 2017; accepted September 2017