Client-aware Cloud Storage

Feng Chen
Louisiana State University
fchen@csc.lsu.edu

Michael P. Mesnier
Intel Labs
michael.mesnier@intel.com

Scott Hahn
Intel Labs
scott.hahn@intel.com

***Abstract*—Cloud storage is receiving high interest in both academia and industry. As a new storage model, it provides many attractive features, such as high availability, resilience, and cost efficiency. Yet, cloud storage also brings many new challenges. In particular, it widens the already-significant semantic gap between applications, which generate data, and storage systems, which manage data. This widening semantic gap makes end-to-end differentiated services extremely difficult. In this paper, we present a client-aware cloud storage framework, which allows semantic information to flow from clients, across multiple intermediate layers, to the cloud storage system. In turn, the storage system can differentiate various data classes and enforce predefined policies. We showcase the effectiveness of enabling such client awareness by using Intel's Differentiated Storage Services (DSS) to enhance persistent disk caching and to control I/O traffic to different storage devices. We find that we can significantly outperform LRU-style caching, improving upload bandwidth by 5x and download bandwidth by 1.6x. Further, we can achieve 85% of the performance of a full-SSD solution at only a fraction (14%) of the cost.**

***Index Terms*—Storage, Operating systems, Cloud computing, Cloud storage**

I. Introduction

Cloud storage is becoming increasingly popular among enterprise and consumer users. According to an IHS report, personal cloud storage subscriptions have reached 500 million in 2012 for major providers such as DropBox and iCloud [10]. The combined public and private cloud storage market is predicted to be $22.6 billion by 2015 worldwide [24].

Unlike conventional storage, such as local disk drives or NFS servers, public cloud storage users store, access, and manage files (objects) stored in the data centers of service providers and pay for the service based on a monthly rate, or actual usage. For its unique technical merits, such as high availability, resilience, and cost efficiency, cloud storage is quickly changing the way people store and manage data.

A. The Challenge of Widening Semantic Gap

Benefits aside, cloud storage introduces many new challenges (e.g., [16], [17], [20], [26]). In particular, it further widens the already-significant *semantic gap* between applications and storage systems, making it especially difficult to realize end-to-end differentiated services, often referred to as Class of Service (CoS).

Cloud storage is not a file system that users can directly interact through conventional `read`/`write` commands. Instead, an HTTP-based REST interface (e.g., GET/PUT) is used for transmitting data from client to server. During this process,

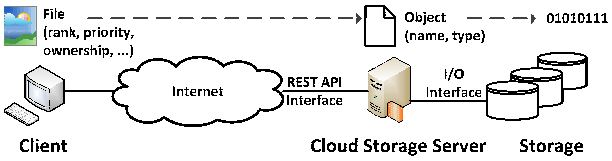


Fig. 1. Data flow in the existing cloud storage framework

valuable semantic information, which is only available on the client, is lost. Figure 1 illustrates such a process – (1) Before uploading a file, the user has rich semantic knowledge about the data, such as ownership, priority, data importance, interest-based ranking, etc. (2) When the data is transmitted to the cloud storage server, the server sees an object with limited information (e.g., object name), and most semantic information is stripped off. (3) When data is written to the storage system, the storage system only sees a stream of bytes. As so, realizing end-to-end (i.e., client-to-server-storage) differentiated services becomes extremely difficult. For example, how can cloud storage tell the difference between 1-star songs and 5-star songs and store them differently?

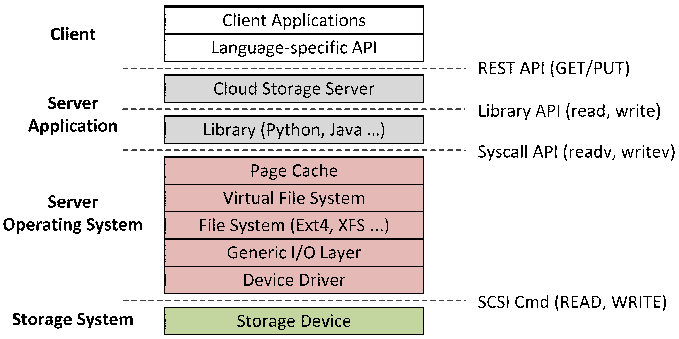


Fig. 2. Logical/physical interfaces in cloud storage

To enable client awareness in cloud storage, semantic information must flow along with the data across multiple logical and physical layers (See Figure 2). Unfortunately, with existing cloud storage infrastructures, we lack the appropriate mechanisms to enable such semantic information flow, from collecting the semantic information on the client side to utilizing such information on the server side. We need to systematically reconsider the entire cloud storage stack and build a semantic information channel from the client to the storage, which involves multiple intermediate layers that need to be tailored to make cloud storage truly *client-aware*.

978-1-4799-5671-5/14/$31.00 © 2014 IEEE Published by the IEEE Computer Society

B. Potential Use Cases

Client-aware cloud storage can be used in many scenarios.

- **Semantic importance-based caching** - Emerging storage technologies, such as flash SSDs, can be used for persistent storage caching in cloud storage systems. Lacking semantic hints from clients, traditional caching schemes, such as LRU, cannot reflect the semantic importance of data from users' perspective. With client-awareness, we can selectively choose semantically critical data for caching.
- **Selective on-line compression** - Cloud storage can apply on-line compression to reduce capacity and cost, however, the efficiency of compression is highly dependent on object types. For example, multimedia objects, such as video and audio files, are already heavily compressed. Compressing such objects cannot provide any further benefits, but only unnecessary overhead. With our client-aware cloud storage framework, these difficult-to-compress objects can be labeled in advance to bypass the compression.
- **Differentiated encryption** - Privacy and security are always important in cloud services. Encryption provides protection but incurs a potentially high performance penalty, which degrades user experience. Giving users the capability to selectively encrypt certain data can help achieve a good balance between performance, cost, and security. For example, videos can be uploaded without being encrypted, while emails can be encrypted.

To enable each of these possibilities, the cloud storage must have the capability of differentiating requests and applying different service policies to user generated data.

C. Client-Aware Cloud Storage

We present a design for building client-aware cloud storage. In principle, our design is based on *data classification*. Specifically, the client classifies data and requests different classes of data to be handled with different storage policies (e.g., low latency versus high throughput). The classification information, as well as the data, is transmitted over the network to the cloud storage server through an augmented REST interface (PUT/GET). The cloud storage is responsible for extracting the classifiers from the HTTP requests and translating them into storage system policies. The storage system, in turn, enforces the associated policies. In other words, this framework enables clients to label data and pass the label along with data to the storage, where data is managed based on labels.

We have prototyped a complete stack of the proposed client-aware cloud storage based on OpenStack Object Storage (Swift) [1]. This prototype system includes (1) a cloud storage client emulator that simulates hundreds of clients that are concurrently performing cloud storage operations based on specified object type/size distributions, (2) a classification-enabled cloud storage server, which handles labeled requests and interacts with a policy-based storage system, and (3) a tiered storage system based on DSS [23], which enforces certain service policies. We demonstrate the strength of this framework with two practical applications, persistent disk caching and fine-grained I/O traffic control, to enhance cloud storage performance and manageability. As mentioned, the proposed mechanism can also enable a variety of other optimization opportunities, such as differentiated security and reliability.

D. Our Contributions

In this paper, we make the following contributions: (1) We present a client-aware cloud storage framework designed to improve end-to-end class of service (CoS). (2) We propose a backward compatible and easy-to-standardize REST API interface for transmitting semantic hints to the server. (3) We present a complete prototype of the proposed framework. (4) We demonstrate the effectiveness of the proposed framework by using persistent disk caching and an I/O traffic control.

The paper is organized as follows. Section II presents background on cloud storage. Sections III and IV describe our design and implementation. Section V presents the experimental evaluation. Section VI presents the related work. Section VII discusses our future work.

II. Background

In this section, we introduce the storage model, the architecture, and the API of cloud storage. Our description is based on OpenStack Swift [1], which is similar to Amazon S3. More details can be found in the on-line documentation [1].

A. The Swift Cloud Storage Model

An *object* is the basic entity of user data in cloud storage. Conceptually, an object is akin to a file in file system. Each object can be associated with optional metadata in the form of a key/value pair. An object can be specified as a URL consisting of a service address, container, and object name (e.g., `http://localhost:8080/v1/AUTH_test/c1/foo`). The maximum object size is 5 GB, which is limited by the HTTP protocol. Objects larger than 5GB must be segmented into smaller chunks, and a manifest is used to locate the related segments of a big object. Objects are organized into logical groups, called *containers* (akin to *buckets* in Amazon S3). A container is analogous to a directory in a file system. Unlike directories, containers cannot be nested, but users can emulate a hierarchical naming structure by arbitrarily inserting "/" in object names.

B. Cloud Storage API

Almost all cloud storage services provide a simple REST (Representational State Transfer) web service interface to allow users to store and retrieve objects. Normally at least five standard HTTP primitives (verbs) are supported – `PUT` (uploading), `GET` (downloading), `POST` (updating object metadata), `HEAD` (retrieving object metadata), and `DELETE` (removing an object). For each operation, a *URL* is presented to specify the target object stored in the cloud storage. Besides the operation and the URL, a number of HTTP *headers* are used to carry extra information to the server. For example, in Swift, the `X-Auth-Token` header carries the authentication

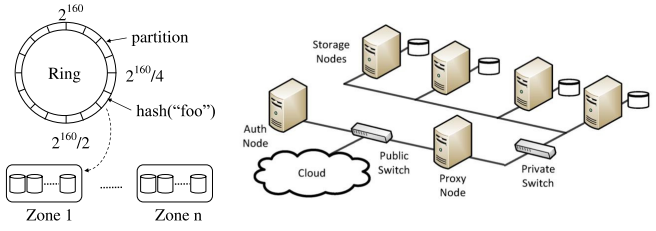
token for access control. We leverage HTTP headers to carry extra hints from the clients.

With the REST API interface, a request can be constructed easily with tools, like `curl` [2]. The below is an example of uploading the file 'foo' to cloud storage in the container 'c1' with an authentication token of 'abc'. Some cloud storage services also provide command-line, GUI (e.g., CyberDuck), or web interfaces. In essence, such tools simply translate actions to REST operations.

```
curl -X PUT -H "X-Auth-Token: abc" -T "foo" \
    http://localhost:8080/v1/AUTH_test/c1/foo
```

C. Swift Object Storage Architecture

Cloud storage is a large-scale distributed storage system carefully designed for availability, resilience, and cost efficiency. A *ring* describes the mapping from the name of storage entities (e.g., an object or a container) to their physical locations. Accounts, containers, and objects have separate rings. Storage entities in a ring are divided into *partitions*. Each partition is replicated (3 times by default) in the cluster, and the mapping (locations for a partition) is maintained by the ring. The ring also determines the fail-over devices. Partitions are guaranteed to be evenly distributed among all devices in the cluster. For reliability, storage devices are logically organized into *zones*, based on their physical location, network connectivity, machines/cabinets, etc. For example, a zone could be a disk drive, a server, or a cluster of servers connected to the same switch. Zones should be isolated from each other as much as possible. When replicating a partition, each replica is guaranteed to reside in a different zone.



(a) Ring Structure **(b) Server Architecture**

Fig. 3. An illustration of cloud storage system

A cloud storage cluster consists of many physical machines (*nodes*), each runs one or more *services* (Figure 3). The *proxy server* is a gateway that exposes APIs to clients and handles incoming requests. The *object server* performs PUT, GET, and DELETE operations on local storage devices. Each object is stored in the host file system as a file and associated metadata, if any, is stored in the file's extended attributes block (XATTR), which requires support from file systems, such as Ext4 or XFS. The *container server* maintains listings of objects in a container. It only tracks whether an object is in a container and disregards its physical location. The *account server* maintains listings of containers for an account, similar to container servers. Swift nodes also run several supporting services, such as the *replicator*, which replicates data in the cluster, the *updater*, which handles delayed updates to storage entities, and the *auditor*, which periodically scans the file system to check object integrity. We normally call a machine running the proxy service a *proxy node*, and a machine running the other services as *storage node*. Since the services and nodes are largely isolated from each other, the storage services can be scaled out.

III. Design

In order to make cloud storage client-aware, we need to consider the following three questions:

- How to describe and express the most valuable semantic information in a compact and general way to satisfy the CoS requirement for applications?
- How to transmit the semantic information across multiple logical and physical interfaces between client and storage, without introducing significant disruptive changes to the existing architecture?
- How to appropriately handle and leverage the client-supplied semantic hints for improving services?

In this section, we answer these questions by following the information flow from the client to the storage – (1) generating semantic information at the client, (2) transmitting the semantic information to the server, (3) handling the semantic information appropriately at the server, and (4) enforcing service policies in the storage system. Figure 4 illustrates this process.

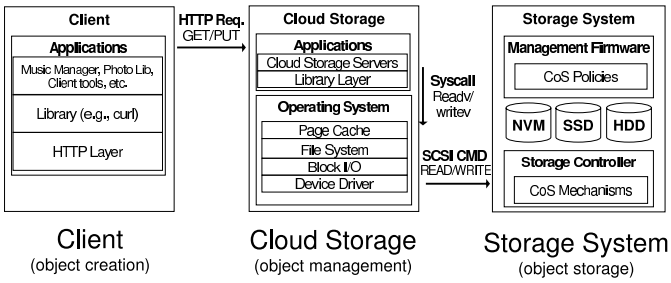


Fig. 4. Architecture of client-aware cloud storage

A. Collecting Semantic Information on the Client

Users possess rich semantic knowledge about data. However, the semantic information that matters the most differs across various applications. For example, ranking information is important for managing a music library (e.g., 5-star songs vs. 1-star songs), while distinguishing I-frames in a video file is important for video analytics. By being able to differentiate distinct classes of data, cloud storage can employ proper storage management policies for various purposes, such as selectively caching important objects in SSDs (e.g., 5-star songs) or encrypting certain objects (e.g., emails).

One principle of our design is to separate *data classification* and *policy enforcement*. Specifically, the client classifies data and the storage system enforces policies. Such a separation enables a dynamic mapping between data classes and storage policies, which can be flexibly defined and configured by administrators.

1) Data classification: A *classifier* is a numerical value differentiating one group of data from another. Classes are used only for the purpose of differentiating data, and the associated numerical values do not necessarily imply any relative priority for storage services. The size of a classifier is defined by the capability of storage system. For example, a 5-bit classifier is used in DSS [23].

In our design, data classification can be performed at three different granularities:

- **Object-based** – The client can associate a classifier with an entire object.
- **Range-based** – The client can associate a classifier with a range of 512-byte sectors in an object.
- **Block-based** – The client can associate a classifier with a given block (e.g., 4096 bytes) in an object.

Applications select the appropriate classification granularity based on their needs. For example, a music library manager can use object-based classification to differentiate 1-star songs and 5-star songs; a video analytics tool can use range-based classification to separate I-frames from the other frames in a video file; a virtual machine manager can use block-based classification to identify metadata and data blocks inside a virtual disk image file. If no class is provided, the object is regarded as a regular object with no classification information and the default management policies are applied. This makes classification an enhancement rather than a requirement, which provides backward compatibility.

2) Classifying data at the client: The client is responsible for creating and transmitting the class information to the server. In our vision, at least three classification mechanisms can be implemented on the client.

- **Manual labeling** – A user can manually assign a classifier to a file either through a command line interface or a GUI interface (e.g., mark a file with colors in a right-click menu). Users often upload data through a dedicated client, which also provides an opportunity for cloud service providers to integrate such a capability.
- **General-purpose classification** – Client systems can provide a default classification. For example, DropBox clients segment objects larger than 4MB into multiple 4MB chunks [14], which makes a cloud storage server lose track of the original file sizes, which is important for caching. A general-purpose classification scheme can directly extract the file size information from the client and send to the server.
- **In-app classification** – Applications with integrated cloud storage support can extract deep semantic information. For example, a music library manager (e.g., iTunes) can use the ranking information (e.g., 1-star songs vs. 5-star songs) to classify music files based on the semantic importance.

B. Transmitting Classification to the Server

We provide both in-band and out-of-band modes to transmit classification information from client to server. In both cases, classifiers can be transmitted either in headers, or in objects.

1) In-band mode: In the in-band mode, the classifiers are transmitted along with the object content to the cloud storage server in one single HTTP request. There are two ways to perform in-band classification.

• **Transmitting classifier in HTTP headers** – When a client sends an HTTP request to the cloud storage server, we create HTTP headers to transmit classifiers for object- and range-based classification. For object-based classification, we create a new header `X-DSS-Object-Class` to contain a classifier, which represents the class of all blocks in the object, in an HTTP request. The following is an example of classifying the object "foo" as class 25.

```
curl -X PUT -H "X-Auth-Token: abc" -T "foo" \
    -H "X-DSS-Object-Class: 25" \
    http://localhost:8080/v1/AUTH_test/c1/foo
```

For range-based classification, we create a new header `X-DSS-Range-Class` to contain a string, which specifies the classes for a sequence of data ranges of the object. Each range is represented as `<offset>-<length>-<class>`, where `offset` and `length` are in units of sectors (512 bytes) and `class` is the classifier for the corresponding data range. The following is an example of classifying two ranges of blocks to class 25 and 26.

```
curl -X PUT -H "X-Auth-Token: abc" -T "foo" \
  -H "X-DSS-Range-Class: 0-64-25,1024-32-26"\
  http://localhost:8080/v1/AUTH_test/c1/foo
```

A constraint of using headers to transmit classifiers is that the header size is limited. For example, the Apache server can accept a header of at most 8190 bytes by default. Thus, it is unsuitable for transmitting a large number of classifiers, especially for block-based classification in a large object.

• **Transmitting classifiers in objects** – To address the header-size limitation, classifiers can be transmitted by crafting a specially formatted object file, containing both classifiers and data. We create a header `X-DSS-Object-File` to use with PUT operations. If the header contains a string 'True', the uploaded file is an object with classifiers embedded. Upon receiving such a request, the cloud storage server knows to extract the embedded classification information.

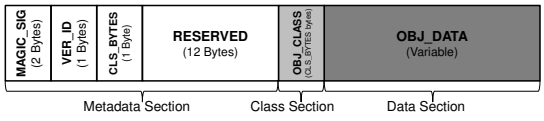


Fig. 5. Object-based Classification Format

An object with embedded classification information contains at least two sections, a *Metadata* and a *Class* section, plus an optional *Data* section. Specifically, (1) the metadata section describes the format of the class section, (2) the class section contains one or multiple fixed-size entries, each of which associates a class with object data, and (3) the data section contains the object data, if the in-band mode is used.

All three classification methods are supported by in-object transmission of classification information. Depending on the classification method used, metadata and class sections are defined differently. Figure 5 shows an example of object-based

classification. Complete descriptions of all three types of data classification can be found in the Appendix (A.1).

2) Out-of-band mode: In the out-of-band mode, the classifiers can be transmitted to the cloud storage after an object is uploaded. This gives users an opportunity to re-classify an object without re-uploading the data. There are two ways to perform out-of-band classification:

• **Transmitting classifiers in headers** – The header can be used with `GET` operations for downloading, or with `POST` operations to update the object's metadata. When the server receives the request, it re-classifies the object. The following example reclassifies object 'foo' to class 26.

```
curl -X POST -H "X-Auth-Token: abc" \
 -H "X-DSS-Object-Class: 26" \
 http://localhost:8080/v1/AUTH_test/c1/foo
```

• **Transmitting classifiers in objects** – We create a header `X-DSS-Class-File` with PUT operations to upload a file only with the metadata and class sections. When using this request, the target URL specifies the object for re-classification, and the header contains a string 'True.' Upon receiving such a request, the cloud storage server knows that the uploaded file contains classification information only. The following is an example of reclassifying object 'foo.'

```
curl -X PUT -H "X-Auth-Token: abc" \
 -T "foo.cls" \
 -H "X-DSS-Class-File: True" \
 http://localhost:8080/v1/AUTH_test/c1/foo
```

C. Handling Requests with Classifiers on the Server

Upon receiving an HTTP request with classifiers, the cloud storage server needs to (1) extract the classification information, and (2) access the local storage system (as per the DSS protocol).

1) Handling requests with classifiers: When a request arrives, the cloud storage server checks to see if it is a request with classification information. If the request carries classifiers in headers (using `X-DSS-Object-Class` or `X-DSS-Range-Class`), the embedded values are extracted. Then the server begins to receive the object data from the socket buffer and writes data into storage using the DSS protocol [23]. To improve storage performance, the object data is written into storage in chunks, each of which is a sequence of contiguous blocks with the same class.

If the incoming request carries classification information in the object (i.e., the `X-DSS-Object-File` header is 'True'), the server first reads the metadata section from the socket buffer to determine the classification format (object-, range-, or block-based), then reads the class section and forms the classifiers, and finally the object data is read out of the socket buffer and written into the storage system with I/Os and associated classifiers.

2) DSS protocol overview: Running at the application level, the cloud storage server needs to deliver the classification information across the application/OS interface. The server code normally interacts with storage using language-specific APIs, which interface to I/O syscalls. The standard I/O syscalls, such as `read()` and `write()`, only pass limited information, such as the file descriptor, length, memory pointer, etc. We use the POSIX scatter/gather I/O interface to transmit extra classification information to local storage [23].

```
unsigned class = 23; /* a class ID */
int fd = open("foo", O_RDWR|O_CLASSIFY);

iov[0].iov_base = &class; /* class ID */
iov[0].iov_len  = 1; /* 1 byte */
iov[1].iov_base = "Hello, world!"; /* data */
iov[1].iov_len  = strlen("Hello, world!");
rc = writev(fd, iov, 2);
close(fd);
```

The POSIX standard provides a scatter/gather I/O interface, namely `readv()` and `writev()`, to perform vectored I/Os to input/output a data stream from/to multiple memory locations in one syscall. We pack the classifier with the data into a multi-element vector and transmit it through the `readv/writev` interface to the OS kernel. As shown in the above example code [23], the file is first opened with a flag `O_CLASSIFY`. When preparing the array of memory buffers, one additional 1-byte scatter/gather element containing a classifier for the I/O is added as the first element. When the OS sees a scatter/gather I/O to a file with the `O_CLASSIFY` flag set, it assumes the first element of the received scatter/gather list points to a classifier (1 byte) and the remaining elements point to data buffers. Upon receiving such an I/O, the OS extracts the classifier and associates it with a kernel-level I/O request, passes it across the VFS layer, the generic I/O layer, and eventually to the device driver. When the request reaches the SCSI device driver, the classifier associated with the I/O is copied into a 5-bit, vendor-specific `Group Number` field in byte 6 of the SCSI CDB. At this point, the I/O with its classifier is given to the storage system.

D. Enforcing Policies in the Storage System

When an I/O with a classifier is received, the storage system enforces the associated policy, which is assigned to predefined classes when the storage system is initialized. A variety of storage system policies can be developed, and here we demonstrate with disk caching and traffic control.

1) Classification-based persistent disk caching: In the tiered storage system, flash SSDs are used as a cache for hard drives. With semantic hints, the limited SSD space can be efficiently used for caching the most "important" data, which improves system throughput, reduces latency, and also improves cost efficiency.

Here we briefly introduce our caching scheme. More details on the caching mechanisms are available in our prior work [23]. We first segment the SSD cache into 4KB entries. Initially all free entries are linked in a *free list*. For each I/O, cache entries are allocated from the free list and added to a class-specific *dirty list*. A hash table tracks the mapping of logical block number (LBN) to the allocated cache entries. A *syncer daemon* tracks the number of free cache entries. If it reaches a low watermark, the syncer initiates a cleaning

process by scanning the dirty list. Whenever an entry is accessed, it is moved to the end, so the syncer always cleans the least recently used (LRU) entry. The cleaned entries are added back to the free list, which is also an LRU list.

As any caching solution, the most important decision is cache admission and eviction. With semantic hints, the tiered storage system can make caching decisions based on the priorities of data classes. Two algorithms are used:

- **Selective allocation** – When the storage cache is under pressure, (i.e., when the syncer daemon is actively cleaning the dirty entries), incoming I/Os that carry a classifier below a specified priority level would bypass the cache, because caching such "less important" data would only increases cache pressure.
- **Selective eviction** – Knowing data classes and their relative priorities, eviction can start with the lowest priority class. When the low watermark is reached, the syncer daemon scans the lowest priority list first, and then the second lowest one, and so on. For each list, the LRU entry is evicted first. This process repeats until reaching a high watermark.

2) Fine-grained I/O traffic control: A client-aware cloud storage framework also enables many other potential optimizations, such as reliability, security, and management. Fine-grained I/O traffic control is one – Cloud storage vendors often desire to have the capability of controlling I/O traffic, e.g., directing emails and videos to different storage pools. Current solutions are very coarse-grained. and inflexible. Our framework provides a fine-grained classification capability to precisely control the location of each uploaded object by labeling objects with different classes. More details will be discussed in Section V.

IV. Implementation

We have implemented a complete stack of the proposed client-aware cloud storage system, from client, server, to storage. Our prototype system includes five major components.

(1) **Cloud Storage Client** – As existing cloud storage benchmarks (e.g., COSbench [4]) do not generate data classification information, we developed a cloud storage client emulator to generate synthesized cloud storage workloads based on specified distributions. With an actual distribution provided by a cloud storage service provider, we can faithfully generate realistic cloud storage traffic. This tool is implemented in Python and consists of about 2,300 lines of code. We use the `pycurl` library for the HTTP communication. Users can specify object type distributions (e.g., 60% for videos), and the object size distribution for each type (e.g., 80% less than 128KB), and assign data classes. When initialized, a pool of files with different types and sizes is created. Then, a pool of connections is created to emulate a specified number of client connections to the cloud storage. This emulator performs experiments in several phases. Each phase of operations follows the user-configured distribution. Details about the workloads will be introduced in the next section.

(2) **Modified the Object Storage Server** – Our client-aware cloud storage server is based on OpenStack Swift 1.4.6 [1]. We added about 600 lines of code in the object server controller, which is a fairly small patch. The main work is to add support to handle requests with classifiers. When we receive a request with classification-related headers (i.e., `X-DSS-<foo>`), the request and the related data are as described in the prior section. For requests with classifiers, we use Python-specific API functions that interface to `writev()` and `readv()` system calls to communicate with the OS kernel. In our current prototype, we have implemented the object-based and range-based classification for both in-band and out-of-band modes. The in-object classification is partially supported in our current prototype.

(3) **Python APIs for Scatter/Gather Syscalls** – OpenStack Swift reads and writes objects through the standard Python APIs for I/Os. Swift 1.4.6 relies on Python 2.x, which does not support the scatter/gather I/O. We wrote a standalone Python library module, called *dssio*, which provides two API functions `dread()` and `dwrite()`, which converts to `readv()` and `writev()` syscalls, respectively. The module is written in C and consists of about 110 lines of code. The library enables the classification information to flow from the cloud storage server to the OS.

(4) **Modified the OS Kernel for Passing I/O Classification** – We modified Linux kernel 3.2.1 to add classification support in the Ext4 file system to allow passing classifiers received from applications via `readv()` and `writev()` to the storage system. We added a classification field (`b_class`) in the `buffer_head` data structure to carry the file system related classifier (e.g., inode). When an I/O request reads or writes the buffers, the classifier is copied to a new classification field (`bi_class`) in a block I/O request (`bio`). When the `bio` reaches the SCSI device driver, the classifier is copied into the 5-bit *Group Number* field in byte 6 of the SCSI CDB. An additional 3 reserved bits can be used, which can further extend 32 classifiers to 256 classifiers.

(5) **Hybrid Storage System** – Our hybrid storage system is implemented as a standalone RAID module in the Linux kernel [23]. We implemented a new RAID level, RAID-9, for users to create and manage a hybrid storage with a heterogeneous set of storage devices (e.g., SSDs and HDDs). Unlike conventional RAID levels, our RAID-9 module dynamically decides the logical-to-physical block mapping across multiple devices based on the data classification information. Also, unlike some RAID levels, such as RAID-5 and RAID-6, we do not need to perform parity calculation. We also modified Linux `mdadm` utility to load the module and create a hybrid storage device (`/dev/md`) with a specified caching device (SSD) and a backend storage device (HDD). The classification scheme and the priority policy are specified during the module loading time. Since the caching device and storage device can both be an another RAID volume, multi-tier (recursive) caching is possible. After the device is initiated, we can create partitions, make file systems, and use it like any block device.

V. Evaluation

We evaluate our prototype with our emulated cloud storage workloads based on real object distributions, including user files, pictures, videos, music, and virtual disk images. The generated workload mimics the cloud storage traffic of a well-known public cloud storage service provider [3].

A. Experimental Setup

All experiments are run on a six-node Linux-based cluster. Two nodes are equipped with two 8-core Xeon Sandy Bridge (E5-2690) 2.9GHz processors (16 cores) and 128GB memory. One node is used as the client, which is responsible for emulating concurrent client connections. The other is used as the cloud storage proxy node, which runs the proxy server to accept incoming requests from the client node. The other four nodes are standard storage servers with two 6-core Xeon Westmere (X5680) 3.3 GHz processors (12 cores) and 24GB memory. The four nodes work as storage nodes providing object, container, account and other supporting services. According to the recommended setup [1], we connect the client and proxy nodes to a 10GbE Switch through two 10GbE links, and the four storage nodes are connected to the switch through 1GbE links.

All nodes are installed with Fedora Core 14 with a DSS-patched Linux 3.2.1 kernel and the Ext4 file system. Each node is equipped with one Intel 710 SSD and two Seagate Constellation ST1000NM0011 1TB hard drives, one of which is used as the system disk and the other is used for experiments. In order to complete the experiments in a reasonable time frame, we keep the working-set around 100GB. We use an SSD and an HDD to organize a hybrid device for each storage node. The actual SSD cache size is configured to a specified percentage (5% and 10%) of the working-set size to reflect the true performance in a real-world setup [8].

Swift cloud storage servers are mostly configured with the recommended default settings. In particular, we set each object to be replicated 3 times. The connection timeout is set to 0.5 second. All services are enabled. We use the built-in authentication method for the proxy server. For each experiment, we create 100 containers and objects are uploaded to randomly selected containers. We configure 32 workers for the proxy server and 8 workers for each object, container, and account server. Each device is configured as an individual zone to evenly distribute the load.

B. Workloads

Our workloads emulate five object categories. For each category, the object size distribution is derived from real files. We use their combination to mimic a real-life public cloud storage service based on their input [3]. Figure 6 shows the Cumulative Distribution Function (CDF). The figure only shows file sizes to 16MB; file sizes larger than 16MB are collapsed. More details are shown in Table I.

- **Files** emulates regular user files (e.g., documents) based on the distribution of files generated by the SPECsfs2008 benchmark, which is also used in prior work [23].

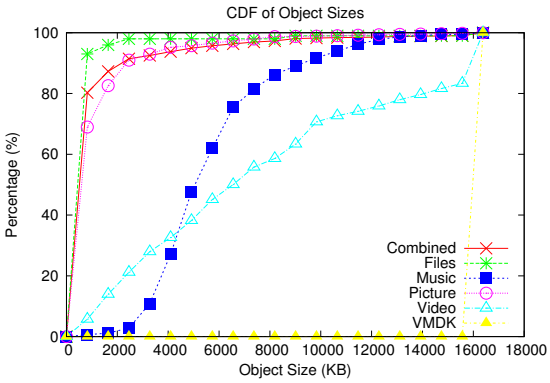


Fig. 6. Distributions of object sizes.

- **Picture** emulates a picture file distribution based on a photo library of 10,711 pictures from flickr.com. The pictures are retrieved with top-ranked search keywords, such as 'wedding,' 'arts,' 'animals,' etc.
- **Music** emulates a music file distribution based on a music library of 2,346 audio files, which consists of different genres, e.g., pop, jazz, and rock, etc.
- **Video** emulates a video file distribution based on a large video library, which contains 319,073 video files crawled from YouTube.com [12].
- **VMDK** emulates a file distribution of large virtual machine disk images in cloud storage. Since the HTTP protocol has an object size limit of 5GB, files larger than that split into smaller ones. So we randomly select the object size between 1GB and 5GB.

File Size	Files	Picture	Music	Video	VMDK
≤64KB	79%	17.1%	0%	0.3%	0%
≤512KB	14%	43.6%	0%	2.8%	0%
≤1MB	3%	14.5%	0.8%	4.9%	0%
≤5MB	2%	20%	52.3%	32.4%	0%
≤10MB	1%	3.5%	39.6%	31.7%	0%
≤50MB	0%	1.3%	7.3%	27.3%	0%
≤100MB	0%	0%	0%	0.4%	0%
>100MB	0%	0%	0%	0.2%	100%
Percentage	60%	35%	4%	0.9%	0.1%

TABLE I
Object Type and Size Distributions

The synthesized workloads perform cloud storage operations in five phases. We first create 100 containers, then perform 26,000 PUT requests to upload 100GB data to the cloud storage, which is replicated 3 times in the cluster. Then we perform 100,000 GET (downloading) requests, and finally DELETE all objects.

C. Case study: disk caching

In cloud storage, SSDs can be used as a cache to speed up I/O accesses. User-specified classification information can be used to guide cloud storage to cache the most important data in the SSD, which significantly improves caching performance.

1) Classification and Storage System Policies: As an example policy, we use object types to classify the objects. We assign the five object types, *Files*, *Pictures*, *Music*, *Video*, and

Description	Class ID	Priority
FS Metadata	1-10	0
Files $\leq$ 256KB	11-14	1
USER0	24	2
...	...	...
USER7	31	9
Files $>$ 256KB	15-22	10
Unclassified	0	11

TABLE II
REFERENCE CLASSES AND CACHING PRIORITY

VMDK with different medium-priority classes (class `USER1-USER5` in DSS [23]), whose caching priorities are high to low in that order. Recognizing that large objects can easily pollute the SSD cache, we assign large objects with the lowest priority (class `USER7` in DSS). For selective allocation, we fence off the lowest priority data when there is cache pressure and let them directly bypass the cache and be self-evicted. As so, we can evict large objects to the backend storage (HDDs) to avoid cache thrash. For selective eviction, we give file system metadata and small files the highest priority, since Swift involves many metadata and small file operations (e.g., SQLite DB files updates). The objects with user-defined classifiers (`USER0-USER7`) are given the second highest priority. The lowest priority is given to large files and unclassified data. Table II gives more details.

2) Performance of Semantic Hint-based Caching: We compare the cloud storage performance on storage with hard drives only (*HDD*), SSDs only (*SSD*), an LRU-based cache (*LRU*), and an enhanced LRU cache that uses semantic hints from the clients (*DSS*). We show the caching effects by setting the cache space proportional (5% and 10%) to the working-set size, which are considered cost-effective in practice [8].

Figure 7 shows the bandwidths, latencies, and failure rates. We can see in the figure that, as we increase the cache space from 5% to 10%, the uploading performance (PUT) is improved for DSS, from 53MB/sec to 76MB/sec. With a 10% cache size, DSS can achieve 87% of the bandwidth of using an SSD-only solution (86MB/sec) and outperform LRU by 5 times. In contrast, LRU remains at 14MB/sec to 15MB/sec, which is even 3.6 times lower than HDD. This is because, knowing data classes, DSS can selectively allocate SSD cache space when the cache is under pressure. In contrast, LRU cannot distinguish and blindly caches everything, which causes data to first flush into cache and soon be evicted out to the hard drives. This doubles the I/O operations and leads to cache thrash.

The downloading (GET) bandwidth difference is also significant. DSS can achieve a bandwidth of 295MB/sec, which is 85% of the performance of the full-SSD solution (347MB/sec) and 1.6 times higher than LRU. LRU, in contrast, shows degraded performance (88MB/sec) with a small 5% cache size, which is 2.1 times lower than HDD. With a 10% cache size, the bandwidth of LRU (178MB/sec) becomes close to HDD.

For latencies, as shown in Figure 7(e), the average downloading latency (i.e., until the first byte is received by the client) for DSS with 10% cache is 89 ms, which is 5.5 times less than LRU (497 ms) and 3 times lower latencies than SSD (275 ms). This is because with DSS, incoming requests can be served from two devices (SSD and HDD), while the SSD-only solution cannot benefit from the device-level parallelism. Also, since the SSD holds mostly small objects, the small requests wait less behind large requests.

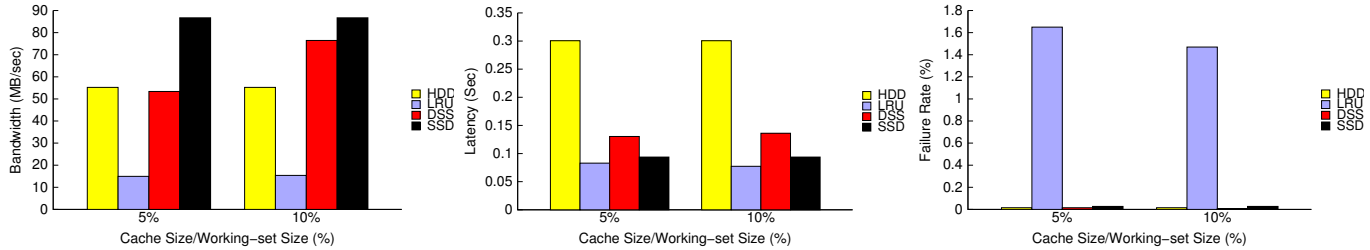
In Figure 7(b), we can find that LRU appears to show a lower latency than DSS for uploading. Figure 7(c) explains this. Due to the HTTP connection timeout, the failure rate of LRU is much higher than DSS, and the failed connections complete earlier than the successful connections by only sending back a failure response (e.g., HTTP 503 code). This makes the average latency of LRU appear lower.

In order to understand why DSS performs better than LRU, we show the content they choose to cache. As shown in Figure 8, LRU uses most space to cache VMDK and other large objects (more than 10MB), and the portion of cache space occupied by these large objects is also roughly constant across different cache sizes (from 5% to 10%). In contrast, DSS caches data based on the user-specified importance. When the cache size is limited, most space is used to cache metadata and the small objects from the "Files" distribution. As cache space increases, more space is used to cache the other classes, such as pictures and music, while VMDK and other large objects remain a small percentage. Since the user-defined classification offloads large objects and VMDK files to the HDD, DSS uses the SSD space more effectively.

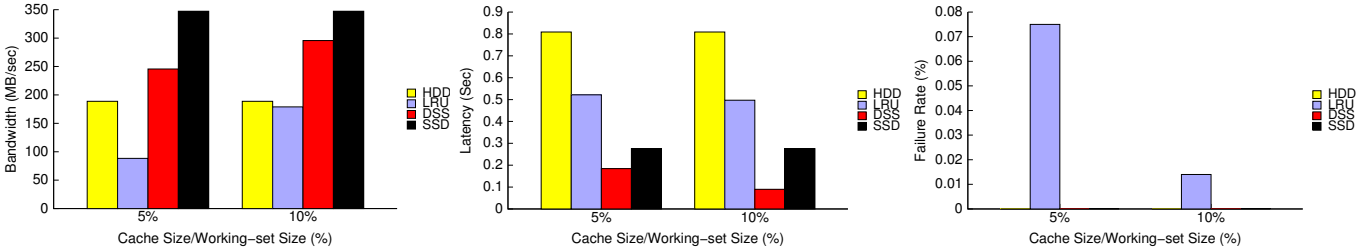
3) Cost Efficiency: We also use "U.S. $/GB/IOPS" as the metric to calculate the cost efficiency. According to Amazon.com, the cost of an Intel 710 series SSD is about $3.95 per GB, which is 26 times more expensive than a Seagate Constellation ST1000NM0011 hard disk ($0.15 per GB). Figure 9 shows the cost efficiency for the four configurations with a 10% cache size. For presentation, we normalize the numbers by using HDD as the baseline. We can see that although the SSD-only solution provides high performance, its cost is 9.9 and 14.3 times higher than HDD for uploading and downloading. DSS is much more cost effective. DSS can achieve a performance comparable to SSD for downloading, but its cost is only 14% of that. LRU is 1.4 times more costly than DSS, due to its less efficient use of SSD space.

D. Case Study: I/O traffic control

Cloud storage service providers often desire to have a flexible control capability to direct I/O traffic to different storage pools for a variety of reasons, such as performance and reliability. In client-aware cloud storage, I/O traffic carries classifiers, which helps achieve this goal easily at a fine level of granularity. In this section, we demonstrate such a case by redirecting I/O traffic to either an SSD or an HDD based on object type. In this experiment, we reuse workloads in the prior section. We set up a non-caching hybrid storage system. As so, when uploading an object, the I/O traffic can be directed either to the SSD or the HDD. This allows us to only speed up accesses for selected data classes.



(a) Uploading B/W (b) Uploading Latency (c) Uploading Failure Rate



(d) Downloading B/W (e) Downloading Latency (f) Downloading Failure Rate

Fig. 7. Performance of caching with semantic hints

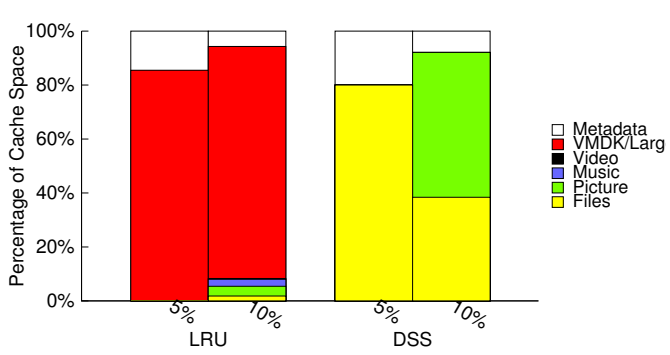


Fig. 8. Cache content breakdown by object types

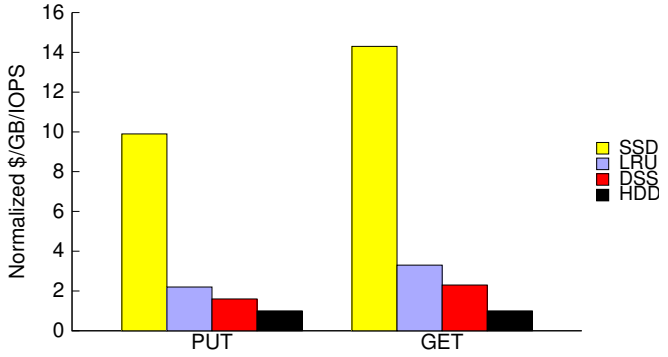


Fig. 9. Normalized cost efficiency (10% cache)

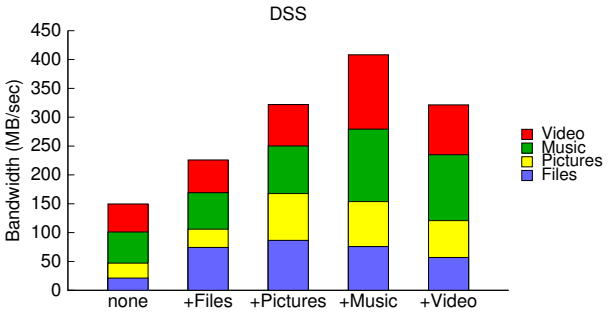
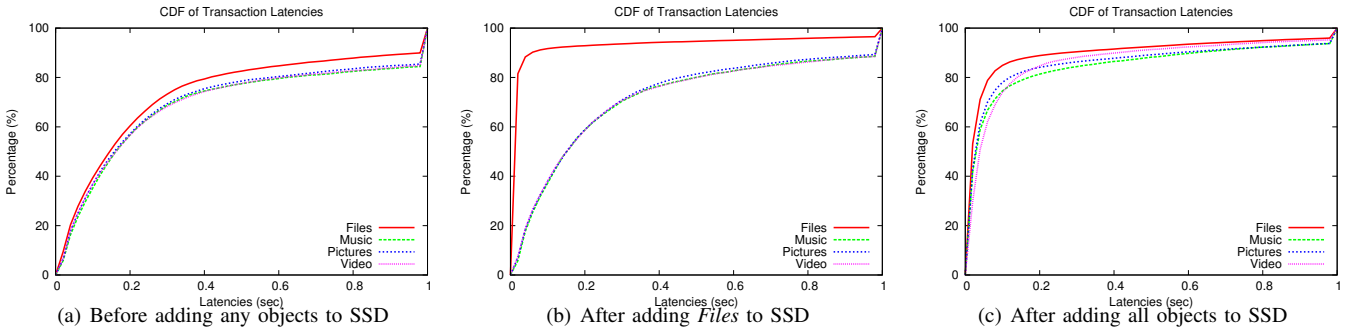


Fig. 10. Aggregate bandwidths with I/O traffic control

For the experiments, we set four concurrent streams, each of which has 25 clients. Each stream uploads one class of objects, namely files, pictures, music, and video. We calculate the bandwidth for each stream individually. For experiments, we performed five test runs with different classification schemes. We first classify all objects as unclassified and send all four streams to the HDD (*none*). In the second run, we send Files and direct its traffic to the SSD (*+Files*). In the third run, we send both Files and Pictures to the SSD (*+Pictures*). In the fourth run, we send Files, Pictures, Music to SSD. In the fifth run, we send all the objects to SSD.

Figure 10 shows the aggregate bandwidths of the five test runs and the bandwidth breakdown of each stream. As we include one additional stream to the SSD, the added stream receives an increase of bandwidth due to the faster device speed. The aggregate bandwidth keeps increasing until it saturates the SSD and network bandwidth. When all four streams are directed to the SSD, we see a decrease in aggregate bandwidth. This is due to two reasons. Firstly, the SSD is over-congested and used to serve all the objects, disregarding the fact that videos can be streamed from the HDD with a good performance. Secondly, the HDD is left unused and the potential I/O parallelism of the two devices is lost.

We also show the Cumulative Distribution Function (CDF) of the transaction latencies for each stream in Figure 11. We can clearly see that after redirecting Files to the SSD (Figure 11(b)), its average access latency can be immediately reduced and its curve differs from other curves. Over 90% of the requests to Files can be finished in less than 100ms. In contrast,



(a) Before adding any objects to SSD (b) After adding *Files* to SSD (c) After adding all objects to SSD

Fig. 11. The effect of traffic control by directing different object types to the SSD

only 40% of the requests can achieve the same latency before this optimization. After we direct all the I/Os to the SSD, their distributions become similar again (Figure 11(c)).

Finally, we would again like to point out that persistent disk caching and the I/O traffic control capability are just two of many possible applications of this client-aware cloud storage framework. Although our evaluation is mostly focused on performance, other optimization goals can be realized. For example, certain objects can be selectively made more reliable through a high-degree replication. Also, selected objects (such as personal emails) can be made more secure by using encryption.

VI. Related Work

Prior cloud storage research has worked on addressing a variety of issues, such as performance, reliability, availability, confidentiality, and lock-in concerns (e.g., [5], [6], [9], [16], [18], [20], [29]). A large body of research has focused on studying the performance of commercial cloud storage services, such as Dropbox and Amazon S3, by passively intercepting and statistically analyzing network traffic on Internet (e.g., [7], [14], [15], [19], [22], [27]). Some prior research attempts to unify the strength of cloud storage and file systems. For example, Vrable et al. have presented a cloud-backed network file system for the enterprise use, called BlueSky [25], to store data in cloud storage and access storage through an on-site proxy, which caches data and provides an NFS and CIFS interface to the clients. Dong et al. presented a similar network file system design, called RFS [13], for mobile devices. Our work is largely orthogonal to these prior efforts. If semantic hints can be provided by the proxy servers when communicating with the cloud storage, potential optimization can be easily achieved with our framework.

Our work is also related to hybrid storage technologies. Caching is important in large-scale storage systems [21]. Mesnier et al. have presented a storage CoS framework, called Differentiated Storage Services, to associate semantic hints with each I/O for optimizing performance in a local storage system [23]. Chen et al. presented a hybrid storage system, called Hystor, by integrating SSDs and HDDs and leveraging SSDs as a cache to hold the small and frequently accessed data [11]. Karma uses hints on database block access patterns to improve multi-level caching [28]. This work aims to use semantic hints in cloud storage scenario, and we find that it can significantly improve performance.

VII. Discussions and Future Work

Exploiting client awareness in cloud storage requires collaboration among clients, servers, and storage. This paper presents a first step in this direction – building a system framework to enable such end-to-end semantic information flow. We demonstrate that this is feasible in practice and can be achieved with relatively small changes to the existing systems. As future research, we will further investigate how to leverage such information to optimize storage systems. In this paper, we have shown two such cases: a priority-based persistent disk caching, and a fine-grained I/O traffic control. Both cases focus on storage management, in which DSS plays an important role for hybrid storage management. In fact, leveraging semantic hints from clients can realize numerous optimization opportunities at various levels, even with non-DSS storage. For example, proxy servers can differentiate uploading traffics to enable a coarse-grained data placement (e.g., different sets of servers). Another potential future work is on the object structure definition and protocol standardization. In this paper, we propose a set of predefined object formats to embed the classification in objects. These definitions, by no means, will be the only possible ones. Other definitions could be developed in the future. However, as any protocol, we need to seek a common agreement between clients and cloud storage service providers. This demands an industry-wide effort to eventually reach a standard to define the ways we describe, transmit, and handle the data and the associated semantic hints in a proper and consistent way. This will be a long-term but important effort in the future.

VIII. Conclusion

Cloud storage is deeply changing the way people store, access, and manage data. In this paper, we present a client-aware cloud storage framework to close the widening semantic gap between client and storage and realize end-to-end differentiated services in cloud storage. With only minor changes to the existing system, we can make semantic information travel together with data from the client end, where data is generated,

to the storage end, where data is stored. We have showcased the effectiveness of enabling client awareness in cloud storage by using semantic hints for persistent disk caching and I/O traffic control. Our experimental results show that we can effectively leverage semantic hints from users to enhance LRU caching, and the cost efficiency ($/GB/IOPS) is 7 times higher than the full-SSD solution for 85% of the performance.

ACKNOWLEDGMENT

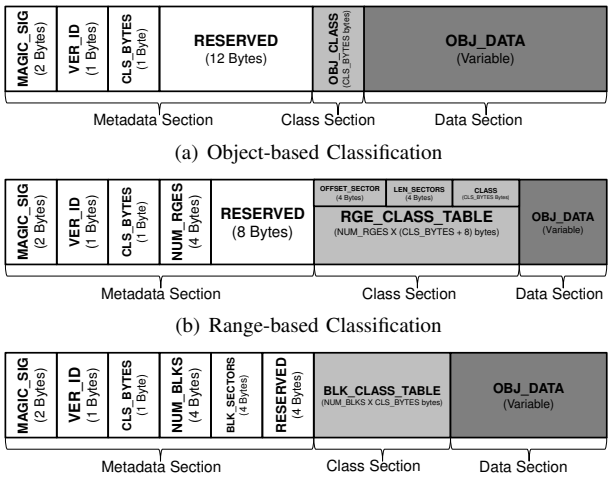
The authors would like to thank reviewers for their constructive comments and the advice. We also thank Paul Brett, Ren Wang, and Pat Stolt for discussions and support during this work. We also thank our industrial partners for providing data and feedback for this research.

REFERENCES

[1] http://www.openstack.org/.
[2] http://curl.haxx.se/.
[3] Anonymized for commercial reasons.
[4] Intel Cloud Object Storage Benchmark. https://github.com/intel-cloud/cosbench.
[5] H. Abu-Libdeh, L. Princehouse, and H. Weatherspoon. RACS: A Case for Cloud Storage Diversity. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, Indianapolis, IN, June 10-11 2010.
[6] S. Bazarbayev, M. Hiltunen, K. Joshi, R. Schlichting, and W. Sanders. PSCloud: A Durable Context-Aware Personal Storage Cloud. In *Proceedings of the 9th Workshop on Hot Topics in Dependable Systems (HotDep 2013)*, Farmington, PA, Nov. 3 2013.
[7] I. Bermudez, S. Traverso, M. Mellia, and M. Munafo. Exploring the Cloud from Passive Measurement: the Amazon AWS Case. In *Proceedings of The 32nd IEEE International Conference on Computer Communications (INFOCOM 2013)*, Turin, Italy, April 14-19 2013.
[8] C. Black, M. Mesnier, and T. Yoshii. Solid-State Drive Caching with Differentiated Storage Services. In *IT@Intel White Paper*. Intel Co., July 2012.
[9] N. Bonvin, T. G. Papaioannou, and K. Aberer. A Self-Organized, Fault-Tolerant and Scalable Replication Scheme for Cloud Storage. In *Proceedings of the 1st ACM symposium on Cloud computing (SoCC 2010)*, Indianapolis, Indiana, June 10-11 2010.
[10] B. Butler. Personal Cloud Subscriptions Expected to Reach Half a Billion This Year. In *Network World*, September 7 2012.
[11] F. Chen, D. Koufaty, and X. Zhang. Hystor: Making the Best Use of Solid State Drives in High Performance Storage Systems. In *Proceedings of the 25th ACM International Conference on Supercomputing (ICS 2011)*, Tucson, AZ, May 31 - June 4 2011.
[12] X. Cheng, C. Dale, and J. Liu. Statistics and Social Network of YouTube Videos. In *Proceedings of the 16th International Workshop on Quality of Service*, Enschede, Netherlands, June 2-4 2008.
[13] Y. Dong, J. Peng, D. Wang, H. Zhu, F. Wang, S. C. Chan, and M. P. Mesnier. RFS - A Network File System for Mobile Devices and the Cloud. In *SIGOPS Operating System Review*, volume 45, pages 101–111, February 2011.
[14] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras. Benchmarking Personal Cloud Storage. In *Proceedings of the 2013 ACM conference on Internet measurement conference (IMC 2013)*, Barcelona, Spain, October 23-25 2013.
[15] I. Drago, M. Mellia, M. M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside Dropbox: Understanding Personal Cloud Storage Services. In *Proceedings of the 2012 ACM conference on Internet measurement conference (IMC 2012)*, New York, NY, November 14-16 2012.
[16] D. Ford, F. Labelle, F. I. Popovici, M. Stokely, V.-A. Truong, L. Barroso, C. Grimes, and S. Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of 9th USENIX Symposium on Operating Systems Design and Implementation*, Vancouver, Canada, Oct 4-6 2010.
[17] S. L. Garfinkel. An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. In *Tech Report TR-08-07*, Harvard University, 2008.
[18] A. Gulati, G. Shanmuganathan, I. Ahmad, C. Waldspurger, and M. Uysal. Pesto: Online Storage Performance Management in Virtualized Datacenters. In *Proceedings of the 2nd ACM symposium on Cloud computing (SoCC 2011)*, Cascais, Portugal, October 27.28 2011.
[19] W. Hu, T. Yang, and J. N. Matthews. The Good, the Bad and the Ugly of Consumer Cloud Storage. In *ACM SIGOPS Operating Systems Review*, volume 44:3, July 2010.
[20] Y. Hu, H. C. H. Chen, P. P. Lee, and Y. Tang. 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 2012)*, San Jose, CA, February 14-17 2012.
[21] Q. Huang, K. Birman, R. van Renesse, W. Lloyd, S. Kumar, and H. C. Li. An Analysis of Facebook Photo Caching. In *Proceedings of the 24th Symposium on Operating Systems Principles (SOSP 2013)*, Farmington, PA, November 2013.
[22] T. Mager, E. Biersack, and P. Michiardi. A Measurement Study of the Wuala On-line Storage Service. In *Proceedings of the IEEE 12th International Conference on Peer-to-Peer Computing (P2P 2012)*, Sophia Antipolis, France, Sept 3-5 2012.
[23] M. P. Mesnier, J. Akers, F. Chen, and T. Luo. Differentiated Storage Services. In *Proceedings of the 23rd ACM Symposium on Operating System Principles (SOSP 2011)*, Cascais, Portugal, October 23-26 2011.
[24] I. P. Release. Demand from Public Cloud Service Providers and Private Cloud Adopters will Drive Strong Growth for Full Range of Storage Solutions, According to IDC. October 11 2011.
[25] M. Vrable, S. Savage, and G. M. Voelker. BlueSky: A Cloud-Backed File System for the Enterprise. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST 2012)*, San Jose, CA, February 14-17 2012.
[26] E. Walker. Benchmarking Amazon EC2 for High-Performance Scientific Computing. In *;Login*, volume 33, October 2008.
[27] H. Wang, R. Shea, F. Wang, and J. Liu. On the Impact of Virtualization on Dropbox-like Cloud File Storage/Synchronization Services. In *Proceedings of International Workshop on Quality of Service (IWQoS 2012)*, Coimbra, Portugal, June 4-5 2012.
[28] G. Yadgar, M. Factor, and A. Schuster. Karma: Know-it-All Replacement for a Multilevel cAche. In *Proceedings of the 5th USENIX Conference on File and Storage Technologies (FAST 07)*, San Jose, CA, February 2007. The USENIX Association.
[29] R. Zhang, R. Routray, D. Eyers, D. Chambliss, P. Sarkar, D. Willcocks, and P. Pietzuch. IO Tetris: Deep Storage Consolidation for the Cloud via Fine-grained Workload Analysis. In *Proceedings of the 4th International IEEE Conference on Cloud Computing (IEEE CLOUD 2011)*, Washington D.C., July 2011.

APPENDIX

A.1 Data Formats for Embedded Data Classification



(a) Object-based Classification

(b) Range-based Classification

(c) Block-based Classification

Fig. 12. Format of objects with embedded classification

- **Object-based Classification** – The metadata section contains four components, a 2-byte `MAGIC_SIG`, which is a randomly selected magic signature indicating that the object contains self-describing classification information, a

1-byte VER_ID, which is a version ID specifying which classification method is used (0 for object-based, 1 for range-based, and 2 for block-based), a 1-byte CLS_BYTES, which specifies the size (in bytes) of the class value, and a 12-byte RESERVED, which is a reserved space for future extension. The class section contains only 1 classifier, whose size is CLS_BYTES bytes.

- **Range-based Classification** – The metadata section contains five components, a 2-byte MAGIC_SIG, a 1-byte VER_ID, a 1-byte CLS_BYTES, a 4-byte NUM_RGES, which specifies the number of entries in the class section, and an 8-byte RESERVED. The class section contains NUM_RGES entries, each of which contains three components, a 4-byte OFFSET_SECTOR, which is the start offset of the range in sectors, a 4-byte LEN_SECTORS, which is the length of the range in sectors, and a CLS_BYTES-byte classifier, which is the class value of the associated range.
- **Block-based Classification** – The metadata section contains six components, a 2-byte MAGIC_SIG, a 1-byte VER_ID, a 1-byte CLS_BYTES, a 4-byte NUM_BLKS, which specifies the number of entries in the class section, a 4-byte BLK_SECTORS, which specifies the size of each block in sectors, and a 4-byte RESERVED. The class section contains NUM_BLKS entries, each of which contains a CLS_BYTES-byte classifier, which is the class value of the corresponding block of the object.