Understanding I/O Performance Behaviors of Cloud Storage from a Client's Perspective

BINBING HOU and FENG CHEN, Louisiana State University
ZHONGHONG OU, Beijing University of Posts and Telecommunications, China
REN WANG and MICHAEL MESNIER, Intel Labs

Cloud storage has gained increasing popularity in the past few years. In cloud storage, data is stored in the service provider's data centers, and users access data via the network. For such a new storage model, our prior wisdom about conventional storage may not remain valid nor applicable to the emerging cloud storage. In this article, we present a comprehensive study to gain insight into the unique characteristics of cloud storage and optimize user experiences with cloud storage from a client's perspective. Unlike prior measurement work that mostly aims to characterize cloud storage providers or specific client applications, we focus on analyzing the effects of various client-side factors on the user-experienced performance. Through extensive experiments and quantitative analysis, we have obtained several important findings. For example, we find that (1) a proper combination of parallelism and request size can achieve optimized bandwidths, (2) a client's capabilities and geographical location play an important role in determining the end-to-end user-perceivable performance, and (3) the interference among mixed cloud storage requests may cause performance degradation. Based on our findings, we showcase a sampling- and inference-based method to determine a proper combination for different optimization goals. We further present a set of case studies on client-side chunking and parallelization for typical cloud-based applications. Our studies show that specific attention should be paid to fully exploiting the capabilities of clients and the great potential of cloud storage services.

Categories and Subject Descriptors: D.4.2 [**Operating Systems**]: Storage Management—*Secondary storage*; D.4.8 [**Operating Systems**]: Performance—*Measurements*

General Terms: Design, Experimentation, Measurement, Performance

Additional Key Words and Phrases: Cloud storage, storage systems, performance analysis, measurement, performance optimization

ACM Reference Format:
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 Trans. Storage 13, 2, Article 16 (May 2017), 36 pages.
DOI: http://dx.doi.org/10.1145/3078838

16

An earlier version of this article was published in *Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST'16)*, Santa Clara, CA.
This work was supported in part by the Louisiana Board of Regents under grants LEQSF(2014-17)-RD-A-01 and LEQSF-EPS(2015)-PFUND-391, the National Science Foundation under grants CCF-1453705 and CCF-1629291, and generous support from Intel Corporation.
Authors' addresses: B. Hou and F. Chen (corresponding author), Department of Computer Science and Engineering, Louisiana State University, 102G Electrical Engineering Building, Baton Rouge, LA, USA 70803; emails: {bhou, fchen}@csc.lsu.edu; Z. Ou (corresponding author), Department of Computer Science and Engineering, Beijing University of Posts and Telecommunications, Xitucheng Road 10, Haidian District, Beijing, China 100876; email: zhonghong.ou@bupt.edu.cn; R. Wang and M. Mesnier, Intel Labs, Intel Corporation, 2111 NE 25th Ave, Hillsboro, OR, USA 97124; emails: {ren.wang, michael.mesnier}@intel.com.

DOI: http://dx.doi.org/10.1145/3078838

1. INTRODUCTION

Cloud storage is a quickly growing market. According to a report from Information Handling Services (IHS), personal cloud storage subscriptions increased to 500 million in 2012 and will reach 1.3 billion by 2017 [IHS 2012]. The global market is expected to grow from $18.87 billion in 2015 to $65.41 billion by 2020 [MarketsandMarkets 2015]. To date, cloud storage not only is used for archiving personal data but also plays an indispensable role in various core commercial services, from serving videos on demand to storing unstructured scientific data.

To end-users, cloud storage is particularly interesting because it provides a compelling new storage model. In this model, data is stored in the service provider's data centers, and users access data through an HTTP-based REST protocol via the Internet. By physically and logically separating data storage from data consumers, this architecture enables enormous flexibility and elasticity, as well as the highly desirable cross-platform capability. On the other hand, such a model is drastically different from conventional direct-attached storage: the "storage medium" is replaced by a large-scale storage cluster, which may consist of thousands of massively parallelized machines; the "I/O bus" is generally the worldwide Internet, which allows connecting two geographically distant ends; the strictly defined "I/O protocol" is replaced by an HTTP-based protocol; and the "host" is not a single computing entity anymore but could be any kind of computing device (e.g., PCs or smartphones). All these properties, together, form a rather loosely coupled system, which is fundamentally different from its conventional counterpart. A direct impact of such change is that much of our prior wisdom about storage, the basis for our system optimizations, may not continue to be applicable to the emerging cloud-based storage.

This is because of several reasons. First, the massively parallelized storage cluster, where data is stored, potentially allows a large amount of independent parallel I/Os to be processed quickly and efficiently. In contrast, our conventional storage emphasizes how to organize sequential I/O patterns to address the limitation of rotating mediums [Ding et al. 2007; Jiang et al. 2005]. Second, compared to the stable and speedy I/O bus, such as Small Computer System Interface (SCSI), the lengthy Internet connection between the client and the cloud is slow, unstable, and sometimes unreliable. A cloud I/O could travel an excessively long distance (e.g., thousands of miles from coast to coast) to the service provider's data center, which may involve dozens of network components and finally result in an I/O latency of hundreds of milliseconds or even more. Finally, the clients, which consume the data and drive the I/O activities, are highly diverse in all aspects, from CPU to memory to storage to communication. Certain specifications (e.g., CPU) are directly related to the capability of a client for handling parallel network I/Os.

Unfortunately, our current understanding of storage behaviors is mostly confined to the conventional storage, which is well defined and heavily tuned to scale in a limited scope, such as direct attached storage or local Storage Area Network (SAN). Some recent studies have benchmarked the performance of cloud storage services [Li et al. 2010; Cooper et al. 2010; Bocchi et al. 2014] and cloud storage clients [Cooper et al. 2010; Meng et al. 2010; Drago et al. 2013, 2014, 2012; Gracia-Tinedo et al. 2013; Ou et al. 2015]. These studies mostly focus on either benchmarking cloud storage on the server side or testing specific cloud storage client applications. In this work, we consider the cloud storage service as a "black box" and aim to observe and analyze the end-to-end performance behaviors from a client's perspective. The user-perceivable cloud storage performance is a result of the interactions between the cloud and the client. A unique challenge in cloud storage is the highly diverse working scenario—cloud storage clients accessing the same cloud can be of very different capabilities (e.g., smartphones, tablets, desktops, servers); even the same client could have variable performance, such as different network speeds caused by the changes of geographical locations. This strongly motivates us to obtain key insight into the unique I/O behaviors of cloud storage, a storage solution for the cloud, especially from the perspective of data consumers.

Specifically in this article, we attempt to answer a set of important questions listed to follow. Successfully answering these questions can help us understand not only the effects of several conventional key factors (e.g., parallelization and request sizes) on cloud I/O behaviors but also several new issues (e.g., client capabilities and geo-distances), which are unique to the cloud-based storage model.

—Parallelization and request size are two key factors affecting the performance of storage. What are their effects on the performance of cloud storage? Can we make a proper tradeoff between parallelism degree and request size?
—CPU, memory, and storage are three major components defining the capability of a client. In the scenario of cloud storage, which is the most critical one affecting the performance of cloud storage? What are their effects on the performance under different workloads?
—The geographical distance between the client and the cloud determines the Round-Trip Time (RTT), which is assumed to be a critical factor affecting the cloud storage speed. What is the effect of such geographical distance to cloud I/O bandwidths and latencies? Should we always attempt to find a nearby data center of a cloud storage provider?
—Parallel reads and writes have been observed having an impact on each other in conventional storage [Chen et al. 2009]. How do cloud storage reads (GET) and writes (PUT) generated by a client interfere with each other? Which one is affected the most?
—Cloud storage provides an object-based storage service. The request size may not always be equal. If large requests and small requests are mingled, what is the effect, and which one is more sensitive to the interference?
—Based on our experimental studies on the performance of cloud storage, what are the associated system implications? How can we use them to optimize client applications by efficiently exploiting the advantages of cloud storage?

To answer these critical questions, we present a comprehensive experimental study on cloud storage from a client's perspective. In essence, our study regards cloud storage as a type of storage service rather than network service. As such, we are more interested in characterizing the end-to-end performance perceived by the client, rather than the intermediate communications. We believe this approach also echoes the demand for thoroughly understanding cloud storage for a full-system integration as a storage solution [Chen et al. 2014].

For our experiments, we develop and run a homemade test tool over Amazon Simple Storage Services (S3). By using latencies and bandwidths, which are the two key metrics used in storage studies, we perform a series of experiments with five different client settings (each client is an Amazon EC2 instance) to study the effect of clients' capabilities and locations on perceived performance. It is also worth noting that our main purpose is not to benchmark the speed of specific cloud storage services. We desire to investigate the end-to-end effects of the major factors that are related to the cloud and its client and gain insight on how to make proper optimizations on the client side. Specifically, we investigate each factor by controlled comparison. Namely, we change one configuration of the baseline client each time and observe its impact on performance.

Our contribution can be summarized as follows:

—Unlike prior measurement work that mostly aims to characterize cloud storage services or specific client applications, we focus on analyzing the effect of client-related factors (e.g., client's capability and location) on the user-experienced performance and the direct interactions between the client and the cloud. For this purpose, we build a homemade tool to access Amazon S3, one of the most popular cloud storage providers, running on Amazon EC2 instances as clients.
—Through extensive experiments and quantitative analysis, we have obtained several important and interesting findings: (1) Parallelizing I/Os and organizing large requests are key to improving client-perceivable performance. An optimized bandwidth could be achieved with a proper combination of the two parameters, parallelism degree and request size. (2) Client capabilities, including CPU, memory, and storage, play an important role in determining the end-to-end performance. (3) A long geographical distance affects client-perceived performance but does not always result in lower bandwidth. Appropriately parallelizing cloud I/Os can effectively hide the impact of long geo-distances. (4) The interference among mixed cloud storage requests may cause significant performance degradation and should be paid sufficient attention to.
—Based on our findings, we present a sampling- and inference-based method to determine a proper combination of two key factors: parallelism degree and request size. We also show the possible strategies that can be taken to achieve different optimization goals: reducing latency and increasing bandwidth. The evaluation results demonstrate the effectiveness of this method.
—We further present a set of case studies on client-side chunking and parallelization for typical cloud-based applications including informed prefetching, synchronization, and file systems. In these case studies, we showcase how to efficiently take advantage of the great performance potential of cloud storage with appropriate client-side optimization.

The rest of the article is organized as follows. Section 2 introduces background. Section 3 describes the methodology for our experimental studies. Section 4 presents the observations and analyzes the results. Section 5 describes the sampling- and inference-based method. Section 6 presents the case studies. Section 7 discusses the system implications of our findings. Related work is presented in Section 8, and the last section concludes this article.

2. BACKGROUND

2.1. Cloud Storage Model

In cloud storage, the basic entity of user data is an *object*. An object is conceptually similar to a file in file systems. An object is associated with certain metadata in the form of key/value pairs. Typically, an object can be specified by a URL consisting of a service address, a bucket, and an object name (e.g., https://1.1.1.1:8080/v1/AUTH_test/c1/foo). The maximum object size is typically 5GB, which is the limit of the HTTP protocol [Amazon 2010]. Objects are further organized into logical groups, called *buckets* or *containers*. A bucket/container is akin to a directory in a file system but cannot be nested.

Almost all cloud storage service providers offer an HTTP-based Representational State Transfer (REST) interface to users for accessing cloud storage objects. Some also provide language-specific APIs for programming. Two typical operations are `PUT` (uploading) and `GET` (downloading), which are akin to `write` and `read` in conventional storage. Other operations, such as `DELETE`, `HEAD`, and `POST`, are provided to remove objects and retrieve and change metadata. For each operation, a URL specifies the target object in the cloud storage. Additional HTTP headers may be attached as well.

2.2. Cloud Storage Services

Cloud storage is designed to offer convenient storage services with high elasticity, reliability, availability, and security guarantees. Amazon S3 [Amazon 2015c] is one of the most typical and popular cloud storage services. Other cloud storage services, such as OpenStack Swift [OpenStack 2011], share a similar structure. Typically, the cloud storage service is running on a large-scale storage cluster consisting of many servers for different purposes, from handling HTTP requests to accounting to storage to bucket listings, and so forth. These servers could be further logically organized into *partitions* or *zones* based on physical locations, machines/cabinets, network connectivity, and so on. For reliability, the zones/partitions are isolated from each other, and data replicas should reside separately. In short, the cloud storage services are built on a massively parallelized structure and are highly optimized for throughput.

2.3. Cloud Storage Applications

Applications can access cloud storage in different ways. Some applications use the vendor-provided APIs to directly program data accesses to the cloud in their software. Such APIs are provided by the service provider and are usually language specific (e.g., Java or Python). Since a cloud storage object can be located via a specified URL, users can also manually generate HTTP requests by using tools like `curl` to access the link.

A more popular category of cloud storage applications is personal file sharing and backup (e.g., Dropbox). Such applications often provide a file-system-like interface to allow end-users to access cloud storage. From the perspective of data exchange, these clients often use syncing or caching to enhance user experience. With the syncing approach, the client maintains a complete copy of the data stored on the cloud-side repository. A syncer daemon monitors the changes and periodically synchronizes the data between the client and the cloud. With the caching approach, the client only maintains the most frequently used data locally, and any cache miss leads to on-demand data fetching from the cloud. In practice, the syncing mode is adopted by almost all personal cloud storage applications, such as Dropbox [2015], Google Drive [Google 2015], and OneDrive [Microsoft 2015]. The caching mode is adopted by applications and storage systems that make use of the cloud as a part of the I/O stack, such as RFS [Dong et al. 2011; S3FS 2015; S3Backer 2015], BlueSky [Vrable et al. 2012], and SCFS [Bessani et al. 2014]. In general, all the aforementioned applications essentially convert the POSIX-like file operations into HTTP requests (e.g., a `read` function call is converted to a `GET` HTTP request). To accurately and directly observe data exchange between the client and the cloud, our study carefully avoids using any specific application techniques (e.g., caching and prefetching) but directly uses the HTTP protocol, which is the underlying communication protocol in cloud storage. This well serves the purpose of our experiments for providing design guidance to optimize cloud storage clients, such as making appropriate caching and prefetching decisions.

3. MEASUREMENT METHODOLOGY

As mentioned earlier, the main purpose of our experimental studies is to characterize the performance behaviors of cloud storage from the client's perspective. In our experiments, we treat the cloud as a "blackbox" storage. In order to avoid interference from client-side optimizations, we carefully generate raw cloud I/O traffic via the HTTP-based REST protocol to directly access the cloud storage and observe the performance on the client side.

Table I. Configurations of Amazon EC2-Based Clients

Client	Instance	Location	Zone	vCPU	Memory	Storage
Baseline	m1.large	Oregon	us-west-2a	2	7.5GB	Magnetic(410GB)
CPU-plus	c3.xlarge	Oregon	us-west-2a	4	7.5GB	Magnetic(410GB)
MEM-minus	m1.large	Oregon	us-west-2a	2	3.5GB	Magnetic(410GB)
STOR-ssd	m1.large	Oregon	us-west-2a	2	7.5GB	SSD(410GB)
GEO-Sydney	m1.large	Sydney	ap-southeast-2a	2	7.5GB	Magnetic(410GB)

Note: The SSD used in our experiments is the provisioned SSD with 3,000 IOPS, and all the clients are equipped with *Moderate* network (tested as 82MB/s) and Ubuntu 14.04 LTS (PV).

Table II. Magnetic Versus SSD on Amazon EC2

Speed / Size	Magnetic		SSD	
	Read	Write	Read	Write
1KB	2.13MB/s	0.77MB/s	2.7MB/s	1.24MB/s
4KB	6.70MB/s	3.13MB/s	10.57MB/s	5.67MB/s
16KB	6.80MB/s	4.60MB/s	34.87MB/s	10.65MB/s
64KB	7.36MB/s	10.67MB/s	62.00MB/s	28.48MB/s
256KB	17.36MB/s	17.46MB/s	58.24MB/s	86.63MB/s
1MB	38.33MB/s	22.38MB/s	58.24MB/s	82.71MB/s
4MB	61.59MB/s	23.20MB/s	58.06MB/s	82.72MB/s
16MB	58.12MB/s	22.66MB/s	58.12MB/s	82.92MB/s

3.1. Main Experimental Platform

Cloud storage services: Our experiments are conducted on Amazon Simple Storage Services (S3). As a representative cloud storage service, Amazon S3 is widely adopted as the basic storage layer in consumer and commercial services (e.g., Netflix and EC2). Some third-party cloud storage services, such as Dropbox, are directly built on S3 [Dropbox 2015]. In our experiments, we use the S3 storage system hosted in Amazon's data center in Oregon (s3-us-west-2.amazonaws.com).

Cloud storage clients: In order to run the experiments in a stable and well-contained system, we choose Amazon EC2 as our client platform from which the cloud storage I/O traffic is generated to exercise the target S3 repository. An important reason for choosing Amazon EC2 rather than our own machines is to have a quantitatively standardized client that provides a publicly available baseline for repeatable and meaningful measurement. For analyzing the impact of client-related factors, we customize five configurations of Amazon EC2 instances that feature different capabilities in terms of CPU, memory, storage, and geographical location. Table I shows these configurations. The *Baseline* client is located in Oregon and equipped with two processors, 7.5GB memory, and 410GB disk storage (denoted as Magnetic). The speeds of the Magnetic and the SSD are tested and shown in Table II. The other four configurations vary in different aspects, specifically CPU, memory, storage, and geographical location (in Sydney). These instances with different configurations can properly satisfy our needs of observing cloud storage performance with "controlled comparison," which means that we observe the effect of an individual factor by comparing the performance of the *Baseline* client with the client that has exactly one different configuration each time. For example, we investigate the effect of the client CPU by comparing the performance observed on the *Baseline* client with that on the *CPU-plus* client, because these two clients only have different CPUs while other configurations remain the same. In other words, our main objective is not to benchmark specific cloud storage clients; instead, we are more interested in the performance difference between the *Baseline* client and the other comparison clients.

Table III. Object-Based Workloads

Object Size	Object Number	Workload Size
1KB	81920	80MB
4KB	40960	160MB
16KB	40960	640MB
64KB	40960	2,560MB
256KB	40960	10,240MB
1MB	16384	16,384MB
4MB	4096	16,384MB
16MB	2048	32,768MB

3.2. Additional Experimental Platform

The aforementioned cloud storage data centers and clients are the basic platforms for us to investigate the effects of different client-related factors. To verify some of our findings, we further deploy our tests on other data centers and clients for verification. In Amazon storage clusters, we also use another S3 data center in Ireland (s3-eu-west-1.amazonaws.com) and set up an EC2 client in Ireland (denoted as *GEO-Ireland*). *GEO-Ireland* has the same configurations as the *Baseline* client except the geographical location. In addition, a client located on the LSU campus (denoted as *Local-campus*) is also used as the client outside Amazon's storage clusters. This client is a workstation equipped with a four-core 3.2GHz Intel Xeon CPU, 8GB memory, a 910GB disk drive, 1,000Mbps network connection, and installed with Ubuntu 12.04.5 LTS and Ext4 file system. The read speed of the disk is tested as 167MB/s, and the write speed is 137MB/s for sequential access. With these additional test platforms, we can verify our findings in a more general way.

3.3. Testing Tool and Steps

For our experiments, we have developed a homemade tool that can flexibly generate different workloads and directly issue raw cloud storage I/O requests to the S3 storage. The tool uses the S3 API [Boto 2015b], which is HTTP based and provided by Amazon. As mentioned in the prior section, we purposely avoid using POSIX APIs (e.g., S3FS), because our goal is to gain the direct view of the cloud storage performance from the client side. Certain techniques (e.g., local cache, data deduplication, data compression) used in some client tools will prevent us from observing the cloud I/O behaviors completely or accurately.

Our testing tool generates workloads with four parameters: *request type*, *request size*, *parallelism degree*, and *object number*. Specifically, *request type* refers to `PUT` or `GET` (i.e., uploading or downloading); *request size* refers to the size of the requested object; *parallelism degree* refers to the number of concurrent requests to cloud storage; and *object number* refers to the number of the objects to be requested in test. Limited by the current implementation of Amazon S3 APIs [Boto 2015a], our testing tool generates upload and download requests for one individual object per connection.

Each run of the test is composed of three steps: (1) Generating workloads and a thread pool. For both uploading and downloading tests, we generate a pool of objects of the same size (determined by the parameter *request size*); the number of the objects is determined by the parameter *object number*. Table III gives more details of the workloads used in our experiments for different object sizes. For the uploading test, the objects will be stored on the client disk, meaning that each object to be uploaded has to be read from the client disk first; for the downloading test, the objects have to be first stored in the cloud as the workload. It is noted that the downloading tests consider the full sync cycle, in which objects are necessarily first saved to the client storage as most cloud clients do, leading to the effects of client storage that we will see later. In particular, each object is associated with a unique ID. That is because we hope to make each request unique, avoiding the possible interference caused by requesting the same objects. Besides, we also create a thread pool, in which the number of the threads is determined by the parameter *parallelism degree*. (2) Sending requests and collecting the test results. In this step, the threads are responsible for sending requests associated with the objects concurrently. The test results including the latency of each request are collected. (3) Processing the collected test data and reporting the statistics data (e.g., average latency and bandwidth).

Two main metrics used in our experiments are *latency* and *bandwidth*. Specifically, *latency* refers to the end-to-end completion time of each request (i.e., the time used to upload/download a single object); *bandwidth* refers to the aggregate bandwidth observed on the client (i.e., the total amount of the data uploaded or downloaded by the client in a time unit), which is calculated as $\frac{object\_size \times object\_num}{duration}$, in which *object_size* denotes the size of a single object, *object_num* denotes the number of objects, and *duration* denotes the time taken to upload/download all the objects. In this article, we also use *peak bandwidth* to refer to the maximum bandwidth observed on the client for a given workload.

3.4. Accuracy

Considering the possible variance of network services and multithread scheduling, we take the following measures to ensure the accuracy and repeatability of the experiments: (1) as stated earlier, we customize the instances of Amazon EC2, which can provide stable services as standard clients rather than picking up a random machine; (2) to avoid memory interference across experiments, the memory is flushed before each run of the experiments; (3) we make the size of the workloads large enough (see Table III) so that each run of an individual experiment lasts for a sufficiently long duration (at least 60 seconds) while still being able to complete the experiments within a reasonable time frame; and (4) each experiment is repeated five times, and we report the average value.

4. PERFORMANCE STUDIES

To comprehensively reveal the effects of different factors, our measurement work is composed of two parts. We first conduct a set of general experiments to evaluate the properties of cloud storage, including parallelism degree and request size. We then focus on studying the effects of client capabilities, including CPU, memory, storage, and geographical locations of the clients. We further conduct a set of extensive experiments to unveil the effects of interference among mixed parallel requests, including mixed upload/download requests and mixed small/large requests.

4.1. Basic Observations

Parallelism and request size are two critical factors that significantly affect the storage performance. Considering the parallelism potential of cloud storage, we set the parallelism degree up to 64. With regard to request size, prior work has found that most user requests are not excessively large [Bocchi et al. 2015], typically smaller than 10MB [Drago et al. 2012]. Also, for transfer over the Internet, most cloud storage clients split large requests into smaller ones. Wuala and Dropbox, for example, adopt 4MB chunks, and Google Drive uses 8MB chunks, while OneDrive uses 4MB for upload and 1MB for download [Bocchi et al. 2015]. Therefore, we set the request size up to 16MB to study the size effect.

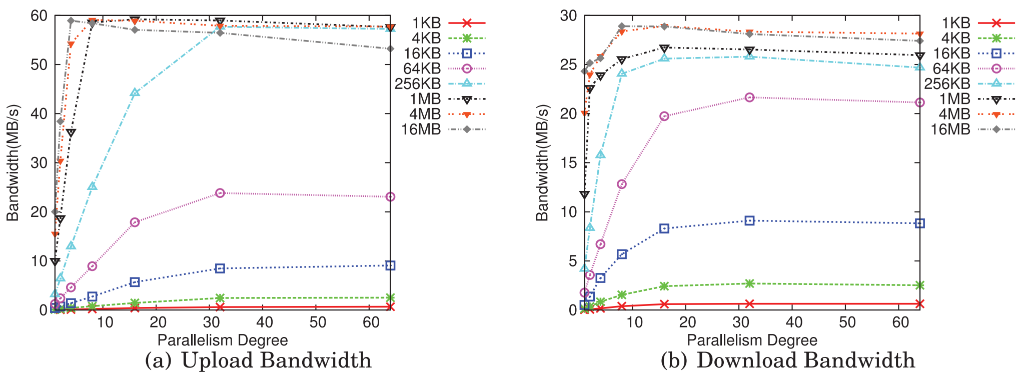


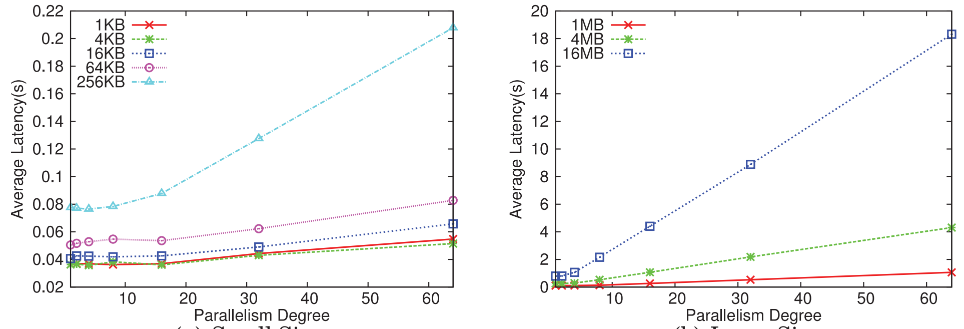
Fig. 1. Upload/download bandwidths on *Baseline*.

4.1.1. The Effect of Parallelism. I/O parallelization is crucial to exploiting the massively parallelized nature of cloud storage. In our experiments, we have observed a strong impact of parallelism to bandwidths and latencies perceived at the client side.

The effect of parallelism on bandwidth. *Proper parallelization can dramatically improve the bandwidth, while overparallelization may lead to bandwidth degradation.* As shown in Figure 1, for example, the bandwidth of 1KB upload requests can be improved up to 27-fold (from 0.025MB/s to 0.666MB/s), and the bandwidth of 1KB download requests can be improved up to 21-fold (from 0.03MB/s to 0.634MB/s). There are two reasons for this. One reason is due to the underlying TCP/IP protocol for communication. With TCP/IP, the client and the cloud have to send ACK messages to confirm the success of the transmission of data packets. With a high parallelism degree, multiple flows can continuously transmit data since the time taken by each parallel request to wait for the ACK messages overlaps. Another reason is that smaller requests often require fewer client resources, so the client can support a higher parallelism degree to saturate the pipeline until the effect of parallelization is limited by one of the major client resources.

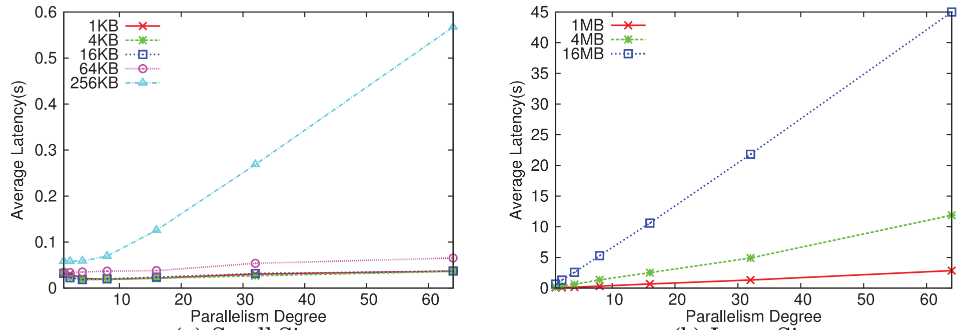
On the other hand, *overparallelization brings diminishing benefits and even negative effects*. For example, 16MB upload sees a slight performance degradation caused by overparallelization. This is related to the overhead of maintaining the thread pool when the CPU is overloaded. As observed, for 16MB uploading, when the parallelism degree increases from one job to eight jobs, the CPU utilization quickly grows from 23% to almost 100%. Under this condition, further increasing the parallelism degree will cause performance loss. In a later section, we will further study the effect of CPU.

The effect of parallelism on latency. *Appropriately parallelizing cloud storage I/Os may not significantly affect the latency (i.e., end-to-end request completion time), while overparallelization could lead to a substantial increase of latency*. As shown in Figure 2 and Figure 3, this speculation is confirmed by the tendencies of the growing average latencies for both upload and download requests as the parallelism degree increases. For example, for 4KB upload requests, when the parallelism degree increases from 1 to 16, the average latency basically remains the same (about 36ms). When the parallelism degree further increases from 16 to 64, the average latency increases by 43% (from 36.1ms to 51.5ms). For large requests, when the parallelism degree exceeds a threshold, the average latency increases linearly. For example, for 16MB upload requests, when the parallelism degree increases from 4 to 64 (16-fold), the average latency increases from 1.1s to 18.3s (17.3-fold). This implies that for latency-sensitive applications, overparallelization (especially for large requests) should be carefully avoided.



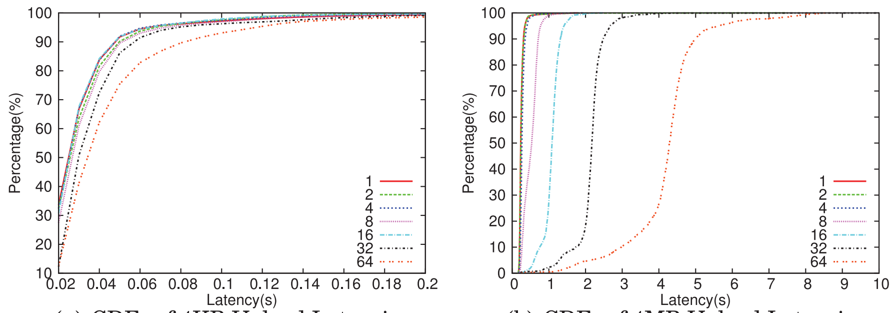
(a) Small Size (b) Large Size

Fig. 2. Average upload latencies on *Baseline*.



(a) Small Size (b) Large Size

Fig. 3. Average download latencies on *Baseline*.



(a) CDFs of 4KB Upload Latencies (b) CDFs of 4MB Upload Latencies

Fig. 4. CDFs of upload latencies with different parallelism degrees on *Baseline*.

Latency distributions. Another finding is that the latency distribution of large requests is more scattered, especially under a high parallelism degree. As shown in Figure 4 and Figure 5, the latency distributions of 4KB requests concentrate in a narrow range, while the latency distributions of 4MB requests spread in a wide range. For example, when the parallelism degree is 64, the distribution range of 4KB upload requests is from 20 to 200ms, while the range of 4MB upload requests is 200ms to 10s. This implies that in the scenario of parallelism, the latency of small requests is more predictable than that of large requests.

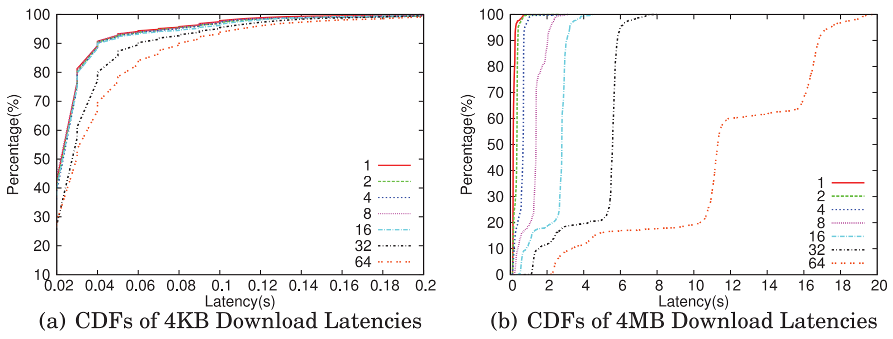


Fig. 5. CDFs of download latencies with different parallelism degrees on *Baseline*.

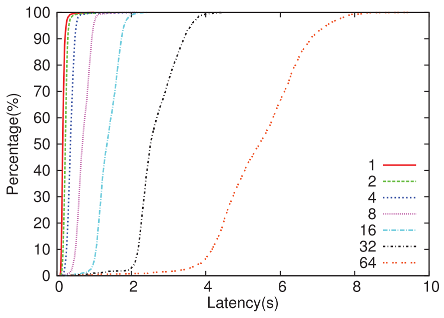


Fig. 6. CDFs of download latencies with different parallelism degrees on *Baseline* with *ramfs*.

We also note the "steps" in the latency CDFs of 4MB downloading requests, which are not as smooth as other CDFs. This is likely to be caused by the interference of the memory flushing behavior. Linux flushes dirty data in memory periodically to external disk. With a large memory buffer and moderate incoming traffic, such asynchronous memory-flushing operations can be quickly completed and hidden from the foreground I/Os. When the arrival rate is higher than the flushing rate, such a flushing operation may cause this effect. In this case, since the disk speed is limited (only 23MB/sec) and lower than the downloading speed (78MB/sec), the client memory will be quickly used up, and the disk flushing time would be reflected in the critical path and affect the user-perceived latency, causing the observed pattern. To confirm this speculation, we repeated the experiments by replacing the disk with *ramfs*, which removes the disk bottleneck, and we can see in Figure 6 that such "steps" disappear, which confirms our speculation.

4.1.2. The Effect of Request Size. In conventional storage, request size is crucial to organizing large and sequential I/Os and is important in amortizing the disk head seek overhead. A similar effect has also been observed in cloud storage.

The effect of request size on bandwidth. *As expected, increasing request size (i.e., the size of GET/PUT) can significantly improve bandwidth, but the achieved benefit diminishes as the request size exceeds a threshold*. As shown in Figure 1(a) and Figure 1(b), the peak bandwidths of large requests and small requests have a significant gap. For example, the peak bandwidth of 4MB upload requests is 23.5 times that of 4KB upload requests (58.9MB/s vs. 2.5MB/s); the peak bandwidth of 4MB download requests is 10.7 times that of 4KB download requests (28.9MB/s vs.

2.7MB/s). There are several reasons for this phenomenon. One reason is that larger I/O requests on client storage generally have higher I/O speeds than small ones (see Table II). Another reason is that larger requests have higher efficiency of data transmission via the network due to the packet-level parallelism [Huan 2002]. Also, a larger request size can better amortize the related overhead.

Similar to the effect of parallelization, increasing the request size cannot bring an unlimited bandwidth increase, due to the constraint of other factors. For example, the speed of client storage is limited. Uploaded objects need to be first read from the local device, and downloaded objects need to be written to the local device. As shown in Table II, when the request size grows from 4MB to 16MB, the speed of Magnetic improves slightly, which limits the I/O speed of the client side. Also, the maximum size of the TCP window is limited, though tunable [Amazon 2015d; Jacobson et al. 1992]. When the request size exceeds a certain threshold, the benefit brought by increasing the request size diminishes. In addition, other factors, such as the link bandwidth on the route, processing speed on the cloud side, and so forth, can also limit the achievable bandwidth. All these observations demonstrate that the benefit obtained by increasing the request size is significant but is not unlimited.

The effect of request size on latency. *Both increasing request size and parallelizing small requests can lead to increased latency.* For example, as shown in Figure 2 and Figure 3, when the parallelism degree is 1, the average latency of 4MB download requests is 192ms—5.8 times that of 1MB download requests (33ms). However, when taking parallelism degree into consideration, things become different. For example, the average latency of 4MB download requests at parallelism degree 1 is 192ms, which is 13.8 times lower than the average latency of 1MB download requests at parallelism degree 64 (2.9s). Therefore, *without considering the latency increase caused by overparallelization, it is difficult to assert that larger requests imply longer latencies*.

For small requests, even at the same parallelism degree, the latencies do not necessarily increase as the request size increases. Figure 2(a) shows that the average latencies of 1KB and 4KB upload requests are nearly the same. Similarly, in Figure 3(a), we find that the average latencies of 1KB, 4KB, and 16KB download requests are nearly equal. The request latency is mainly composed of three parts: data transmission time via network, client I/O time, and other processing time. For small requests, the data transmission time only accounts for a small portion of the overall latency, while the other two dominant parts remain mostly unchanged, which makes the latencies of small requests similar. Also, since the maximum TCP window is 64KB by default, considering parallelism of the network [Huan 2002], the transmission time of the data that are smaller than 64KB is supposed to be similar.

4.1.3. Parallelism Versus Request Size. In prior sections, we find that either increasing the parallelism degree or increasing the request size can effectively improve the bandwidth, but both of them have limitations. Here naturally comes an interesting question: does there exist a combination of parallelism degree and request size to achieve the optimal performance?

Answering this question has a practical value. Consider the following case: if we have a 4MB object to upload, we can choose to upload it by a single thread or split it into four 1MB chunks and upload them in parallel. Which is faster?

Figure 7 shows the performance under different combinations of parallelism degree and request size. Obviously, 256KB$\times$16 has the highest bandwidth (44.2MB/s), which is about 3 times of the lowest (14.5MB/s). This shows that a proper combination exists and can achieve optimal performance. This observation confirms that *appropriately combining request size and parallelism degree can sufficiently improve the bandwidth beyond optimizing only one dimension*.

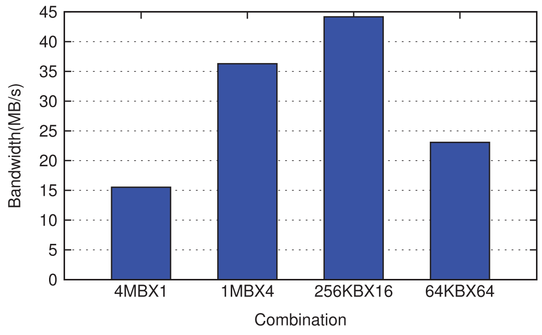


Fig. 7. Upload bandwidths of different combinations on *Baseline*.

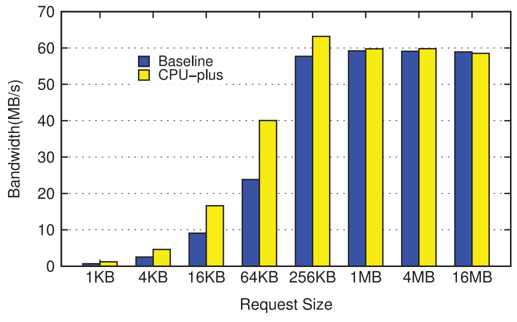
We also find that, *in some cases, either increasing the parallelism degree or increasing the request size by the same factor can achieve the same bandwidth improvement*. For example, for upload requests, 1KB×16, 4KB×4, and 16KB×1 have a similar bandwidth (0.4MB/s). Here comes another practical question: if we have a set of small files (e.g., 1KB), should we adopt a high parallelism degree (e.g., 16) or bundle small files together for creating a larger request size (e.g., 16KB)? From the perspective of improving bandwidth, either a high parallelism degree or large request size is feasible. However, from the perspective of the utilization of client resources, we find that a large request size requires fewer CPU resources. Through `vmstat` in Linux, we find that the CPU utilization of the previous three cases is 65%, 15%, and 5%, respectively. This indicates that for the combinations that can achieve comparable bandwidth, a larger request size consumes fewer CPU resources. That is because for a larger request size, fewer threads have to be maintained to achieve the similar bandwidth, which consequently reduces the CPU utilization.

4.1.4. Additional Remarks. To further verify our findings, we have also repeated the same experiments with two additional experiment settings. We first repeated the experiments on the *GEO-Ireland* client, which was configured the same as the *Baseline* client and accessed the S3 storage in Amazon's Ireland data center (s3-eu-west-1.amazonaws.com), and we had similar observations. We also obtained a similar finding in our experiments with the *Local-campus* client, which is a workstation on the LSU campus and accessed the S3 storage in Amazon's Oregon data center. In Section 4.2 and Section 4.3, we will further investigate how clients' capabilities and geographical distance affect the end-to-end performance.

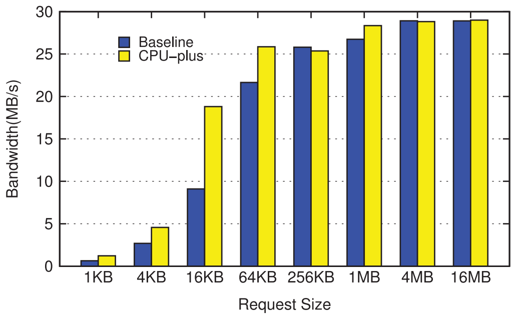
Summary: In this section, we investigate the effects of parallelism and request size on the access latency and bandwidth of cloud storage observed on the client side. Similar to some prior work (e.g., Ou et al. [2015]), we find that parallelism and request size are important to the perceived performance of cloud storage and also have several other interesting findings. For example, access latencies of small requests (e.g., smaller than 64KB) are comparable; parallelization may make the access latencies more unpredictable. Another practically useful finding is that a proper combination of parallelism degree and request size will be helpful to achieve desirable performance. Based on this observation, we also present a *sampling*- and *inference*-based approach for deciding proper combinations in Section 5.

4.2. Effects of Client Capabilities

Unlike conventional storage, cloud storage clients are very diverse. In this section, we study different factors affecting the client's capabilities of handling cloud storage I/Os, namely, CPU, memory, and storage. We compare the performance of three different



(a) Upload Bandwidth



(b) Download Bandwidth

Fig. 8. Peak bandwidth comparison (*Baseline* vs. *CPU-plus*).

clients, including *CPU-plus*, *STOR-ssd,* and *MEM-minus*, with the performance of the *Baseline* to reveal the effects of each factor.

4.2.1. The Effect of the Client CPU. In cloud storage I/Os, the client CPU is responsible for both sending/receiving data packets and client I/Os. In this section, we investigate the effect of client CPU by comparing the performance of *Baseline* (two CPUs) and *CPU-plus* (four CPUs).

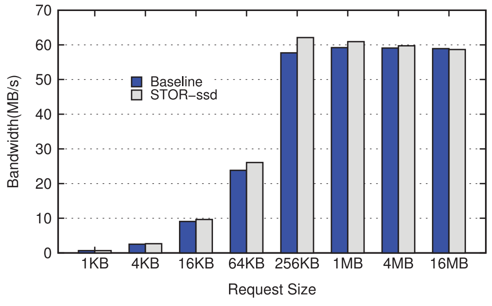
The effect of client CPU on bandwidth. *The client CPU has a strong impact on cloud I/O bandwidth, especially for small requests*. Figure 8 shows the peak bandwidth, which is the maximum achievable bandwidth with parallelized requests. We can see that the peak bandwidth of small requests (smaller than 256KB) increases significantly. Interestingly, as shown in Figure 8(b), the peak download bandwidth of 1KB, 4KB, and 16KB requests doubles, as the computation capability doubles (two CPUs vs. four CPUs). This vividly demonstrates that small requests are CPU intensive, and as so, small requests receive more benefits from a better CPU.

Large requests, compared to small ones, are relatively less sensitive to CPU resources, as the system bottleneck shifts to some other components. As shown in Figure 8, compared with *Baseline*, the peak upload and download bandwidth of large requests (256KB to 16MB) increases only slightly. The system bottleneck may result from the limitation of other factors, such as memory or storage, rather than CPU.

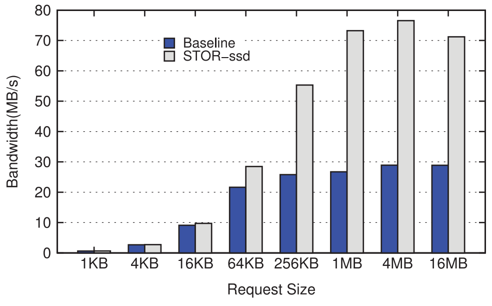
The effect of the client CPU on latency. In our tests, we find that *the client CPU does not have significant effects on average latency*. For small requests, the data transmission via network dominates the overall latency, while for large requests, the majority of the overall latency is the client I/O time (the I/O waiting time may be significant when client storage becomes the bottleneck) and the cloud response time. In the latter cases, a more powerful CPU does not help reduce the latency.

4.2.2. The Effect of Client Storage. Client storage plays an important role in data uploading and downloading: For uploading, the data is first read from the client storage; for downloading, the data is written to the client storage. To evaluate the effect of client storage, we set up a comparison client *STOR-ssd*. The only difference between *Baseline* and *STOR-ssd* is storage (Magnetic vs. SSD). Table II shows more details about the two client storages.

The effect of the client storage on bandwidth. We find that *client storage is a critical factor affecting the achievable peak bandwidth*. As shown in Figure 9, on *STOR-ssd*, the peak download bandwidth increases significantly. For example, the peak download bandwidth of 4MB requests increases by 165% (76.6MB/s vs. 28.9MB/s). On the other hand, we also notice that the upload bandwidth increases slightly. Different



(a) Upload Bandwidth



(b) Download Bandwidth

Fig. 9. Peak bandwidth comparison (*Baseline* vs. *STOR-ssd*).

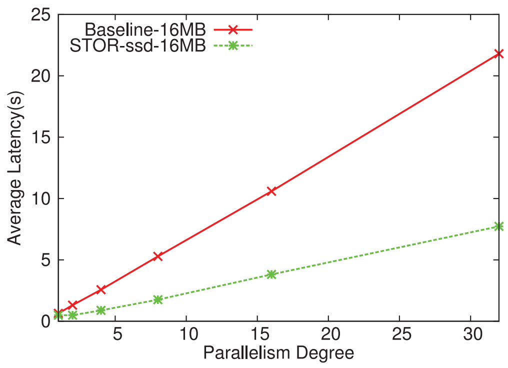


Fig. 10. Average 16MB download latency comparison (*Baseline* vs. *STOR-ssd*).

from the significant improvement of download bandwidth, for example, the peak upload bandwidth of 4MB requests increases only by 2% (60.3MB/s vs. 59.2MB/s). The reason *STOR-ssd* improves the upload bandwidth only slightly is that the Magnetic in our experiments can achieve a similar peak read speed as SSD with a sufficiently large request size and parallelism degree. In contrast, the download bandwidth is limited by the relatively slow speed of Magnetic on the client. To further investigate the effect of the client storage, we have also tested with *ramfs* on the *Baseline* client, which stores data in memory and removes the storage bottleneck. We find that the peak bandwidths can be further improved, but to a limited extent (77.2MB/s for uploading and 80.3MB/s for downloading). In this case, the bandwidth is close to the limit of the network bandwidth (82MB/s), indicating that the network becomes a bottleneck when the client storage is highly capable.

The effect of the client storage on latency. *Similar to bandwidth, we did not observe significant effects of client storage on small requests and large upload requests*. For small requests, client I/O is the minority of the overall latency. In this case, client storage is not a critical factor. For large upload requests, since Magnetic and SSD have similar read speeds, the latencies are comparable; however, for large download requests, *STOR-ssd* can substantially reduce the latency because *STOR-ssd* has significantly advantageous write speed. For example, as shown in Figure 10, when the parallelism degree is 1, *STOR-ssd* can reduce the latency by 24% (0.49s vs. 0.64s); when the parallelism degree is 32, the latency can be reduced by 65% (7.7s vs. 21.8s).

Table IV. Peak Upload Bandwidth Comparison (*Baseline* vs. *MEM-minus*)

	1MB	**4MB**	**16MB**
Baseline	59.2MB/s	59.1MB/s	58.9MB/s
MEM-minus	58.9MB/s	58.7MB/s	58.7MB/s

Table V. Peak Download Bandwidth Comparison (*Baseline* vs. *MEM-minus*)

	1MB	**4MB**	**16MB**
Baseline	26.7MB/s	28.9MB/s	28.9MB/s
MEM-minus	23.7MB/s	23.8MB/s	20.8MB/s

4.2.3. The Effect of Client Memory. Memory in the clients has two functions. First, memory is responsible for offering running space for parallel requests. Second, memory acts as a buffer for uploading and downloading. In this section, we shrink the memory of *Baseline* to investigate the performance differences. The only configuration difference between *MEM-minus* and *Baseline* is that *Baseline* has 7.5GB memory, while *MEM-minus* has only 3.5GB.

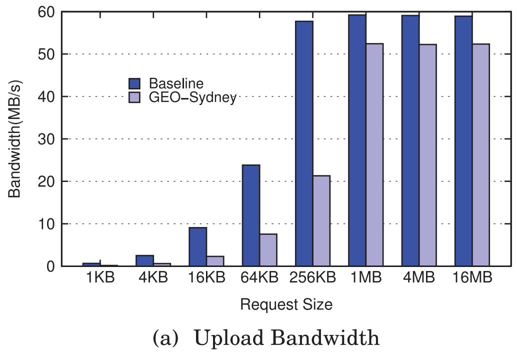
Since small requests are not memory intensive, the effect of memory is trivial. We only present the bandwidths of large requests. The peak upload bandwidth is basically the same (see Table IV), while the download bandwidth dropped heavily (see Table V). For example, on *MEM-minus*, the bandwidth of 16MB download is 20.81MB/s, which is 28.0% lower than that on *Baseline* (28.90MB/s). That is because the write speed of Magnetic is much lower than the read speed and thus more sensitive to the memory space. Therefore, large download requests, especially those involving intensive writes on the client, suffer more from limited memory.

4.2.4. Additional Remarks. To further assess the achievable performance of cloud storage services, we configured a highly powerful Amazon EC2 client located in Oregon (a c3.8xlarge instance) to largely remove the client-side bottleneck. This client is equipped with 32 CPUs, 60GB memory, and 10Gbps networking. By repeating the same experiments on this client with *ramfs* to access the cloud, we find that the maximum achievable bandwidth for uploading and downloading can reach close to 470MB/s, which demonstrates the great performance potential of cloud storage services and also clearly shows the important role of clients' capabilities in determining the user-perceivable performance.

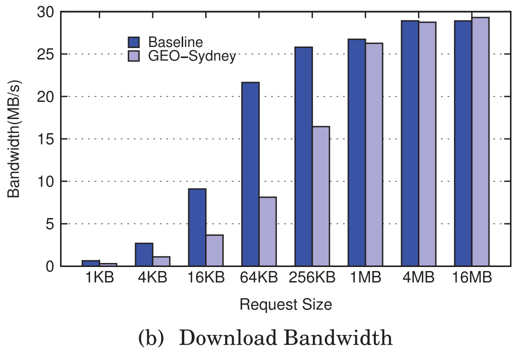
Summary: Clients' capabilities have a strong impact on the end-to-end user-perceivable performance of cloud storage. Based on our observations, CPU is important to small and highly parallelized requests, while storage and memory have significant effects on large requests. Understanding the effects of clients' capabilities can provide us guidance to properly reshape workloads (e.g., selecting a proper combination of parallelism degree and request size) to sufficiently exploit clients' capabilities or set up more reasonable hardware configurations (e.g., HDD vs. SSD) for target workloads. This also indicates that using the same optimization policy across various clients may not be desirable.

4.3. Effects of Geographical Distance

For cloud storage, the geographical distance between the client and the cloud determines the RTT, which accounts for a significant part of the observed I/O latency. The RTT between the *Baseline* client and the cloud is 0.28ms, as both are in the same Oregon data center. In contrast, the RTT between the *GEO-Sydney* client and the cloud in Oregon is about 628 times longer (176ms). This section discusses the effects of geographical distance.



(a) Upload Bandwidth



(b) Download Bandwidth

Fig. 11. Peak bandwidth comparison (*Baseline* vs. *GEO-Sydney*).

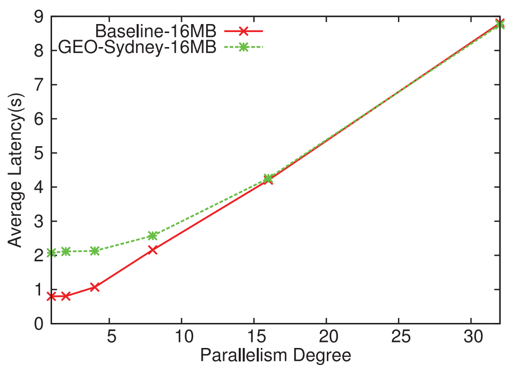


Fig. 12. Average 16MB upload latency comparison (*Baseline* vs. *GEO-Sydney*).

The effect of geo-distance on bandwidth. *The effect of geographical distance on the achievable peak bandwidth is weaker than expected*. As shown in Figure 11, the peak upload bandwidth of *GEO-Sydney* is close to that of *Baseline*. For example, the peak upload bandwidth of 4MB requests of *GEO-Sydney* is only 10% lower than that of *Baseline* (53.3MB/s vs. 59.2MB/s), while the peak download bandwidths of 4MB download requests are basically the same (29.3MB/s vs. 28.9MB/s). This means that RTT is not a critical factor affecting the peak bandwidth, which is mostly due to the Bandwidth-Delay Product (BDP) of the network and is also consistent with the conclusion obtained by Bergen et al. [2011] that the perceived bandwidth from the client is largely determined by the client's network capabilities and the network performance between the client and the cloud.

At the same time, it is also noticeable that *the achievable peak bandwidth of small requests (smaller than 1MB) is much lower with a long geo-distance*. That is because a long RTT needs a high parallelism degree to saturate the pipeline of parallel requests. However, as analyzed in Section 4.2.1, small requests with high parallelism are more CPU intensive; therefore, the CPU capability will become a critical bottleneck for the purpose of sufficiently saturating the pipeline.

The effect of geo-distance on latency. As expected, we also find that *the geo-distance would significantly increase the latency, and its impact on latency makes the client less sensitive to the negative effects caused by overparallelization to latency*. As shown in Figure 12, when the parallelism degree is 1, the average latency of 16MB upload requests on *GEO-Sydney* is 2.1s, which is about 2.6 times of the counterpart on

Baseline (0.8s); as the parallelism degree increases, the average latencies gradually get closer; when the parallelism degree reaches 16, the average latencies are comparable (4.3s vs. 4.2s). If we compare the two, *GEO-Sydney* shows a flatter curve than *Baseline*, because a long RTT needs a high parallelism degree to saturate the pipeline, so the negative effect of overparallelization appears later.

Additional remarks. We also set up another Amazon EC2 client in Europe, *GEO-Ireland*, to test the effect of geographical distance. *GEO-Ireland* has the same configurations as the *Baseline* client except the geographical location. The RTT between the *GEO-Ireland* client and the cloud in Oregon is 128ms, which is much higher than the RTT between the *Baseline* client and the cloud in Oregon (0.28ms). We repeated the same experiments on *GEO-Ireland*, and the experimental results have confirmed our findings shown earlier.

We further had a test on the *Local-campus* client, which has stronger capabilities (four-core 3.2GHz CPU, 8GB memory, 910GB disk, 1000Mbps network) but is remote to the Oregon data center. The observed peak bandwidths for uploading and downloading can reach close to 100MB/s, which is much higher than that achieved on the Amazon EC2 clients. This also shows that the user-perceived performance on cloud storage is significantly affected by clients' capabilities: the client with stronger capabilities can achieve much better performance, even when the client is more distant to the cloud.

Summary: In this section, we investigate the effects of geographical distance. Our experiments confirm the observation reported in prior work [Bergen et al. 2011] that a long geographical distance does not necessarily affect the achievable bandwidth. Besides, we also find that the negative effect of overparallelization may offset the advantage of a short geographical distance. In other words, a client that is far away from the cloud may achieve better performance with proper optimizations than a close client.

4.4. Interference of Mixed Requests

In Section 4.1, we discussed the interference observed among homogeneous requests (i.e., upload/download the objects of the same size). In this section, we further study the interference among mixed requests. In particular, we study the effect of mixed upload/download requests and mixed small/large requests.

In this set of experiments, we maintain two independent daemons to send different requests, and collect their bandwidths and average latencies to observe their interference. We call the daemon whose I/Os are being observed *foreground daemon* and call the other *background daemon*.

In our experiments, we choose 4KB as the representative of small requests and 4MB as the representative of large requests. Both daemons have the same parallelism degree in each run of the experiments. The parallelism degree ranges from 1 to 32. Considering the possible variance caused by thread competition, we repeat each run of the experiments 10 times.

4.4.1. The Interference of Uploading and Downloading. To observe the interference of small upload requests, we set one daemon to send 4KB upload requests and the other to send 4KB download requests; similarly, we set one daemon to run 4MB upload requests and the other to run 4MB download requests.

The interference of uploading and downloading to bandwidth. For small requests, as shown in Figure 13, the bandwidth of both upload and download requests can still increase, but the increasing rate is much slower than that without interference. Since both of their parallelism degrees increase, their bandwidth can be improved before the pipeline is sufficiently saturated. Due to the competition, the increased rate is relatively slower. *Interestingly, small download requests can obtain more bandwidth*

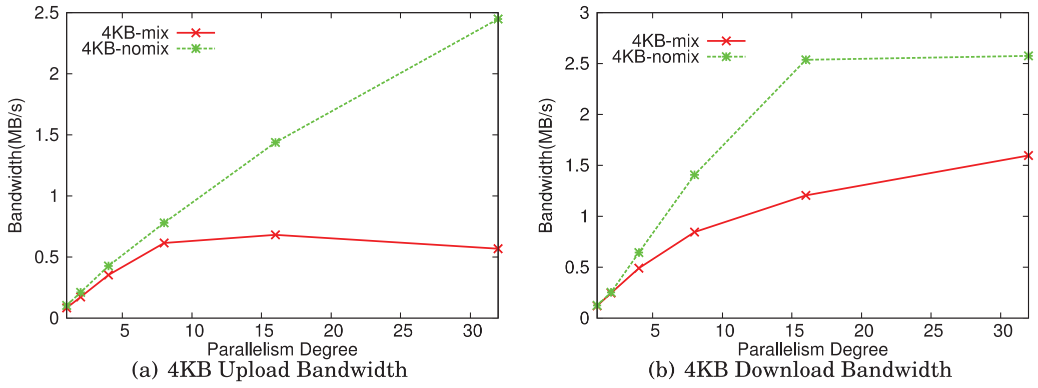


Fig. 13. Bandwidths of mixed upload/download (4KB).

Table VI. Mixed Versus Nonmixed Upload/Download Bandwidths (4MB)

	Mixed	**Nonmixed**
Upload	29.8MB/s	58.91MB/s
Download	18.2MB/s	28.9MB/s

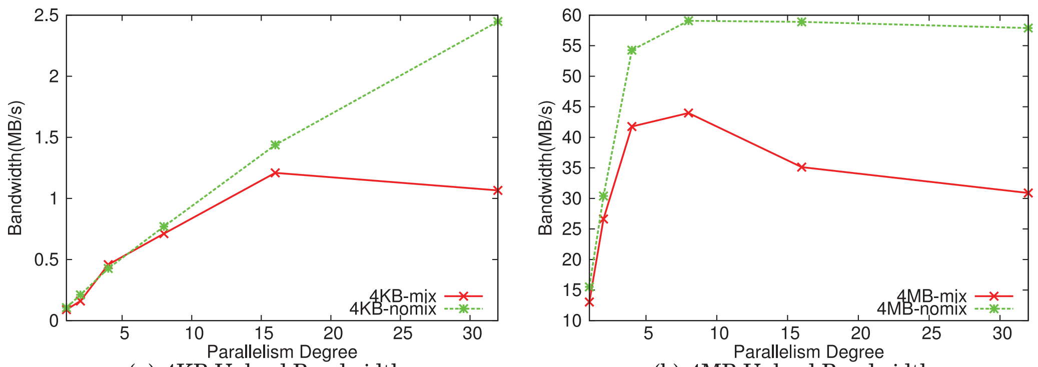
when competing with small upload requests. The bandwidth of 4KB upload requests stops increasing at parallelism degree 8 when being interfered with by 4KB download requests of parallelism degree 8, as shown in Figure 13(a), while the bandwidth of 4KB download requests can continuously increase, as shown in Figure 13(b). That is because the average latency of downloading requests is lower than that of uploading requests, so that the clients can finish more downloading requests from the mixed requests. This means that *small upload requests are more sensitive to the interference*.

For large requests, the bandwidth of both upload requests and download requests decreases dramatically. The peak bandwidth of both upload and download requests drops by 50%. The peak bandwidth of the mixed requests is much lower than the bandwidth of upload requests without interference. For example, as shown in Table VI, the bandwidth of 4MB uploading requests is 58.91MB/s without interference, while the total peak bandwidth of the mixed requests is 48MB/s (the bandwidth of upload requests is 29.8MB/s, and the bandwidth of download requests is 18.2MB/s), which is about 20% lower than the upload bandwidth without interference.

The bandwidth of download requests decreases dramatically even though the interference parallelism degree is low. For example, when the interference parallelism degree is 1, the bandwidth of download requests reduces by 50% (from 20MB/s to 10.7MB/s), while the upload bandwidth drops slightly (from 15.5MB/s to 13.5MB/s) under the same condition. This should be caused by the competition of client I/O resources. Since the write speed of Magnetic is much slower than the read speed, the downloads, especially those involving intensive write operations on the client, are more sensitive to the interference.

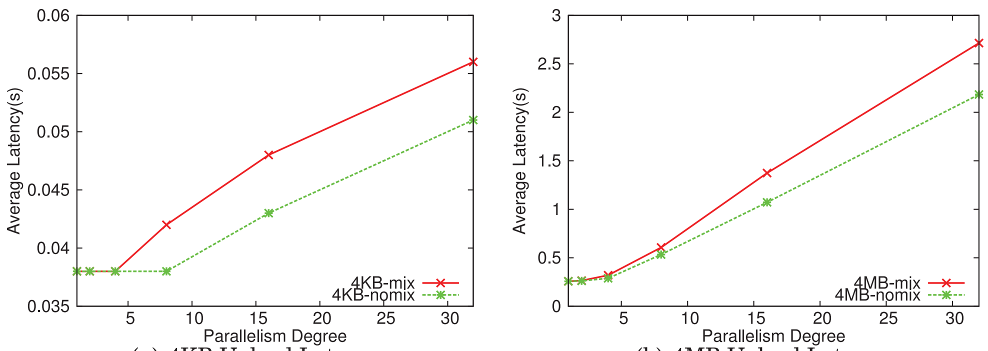
The interference of uploading and downloading to latency. With regard to the latency, when the interference parallelism degree is high, the average latency of both upload and download requests increases heavily, especially for mixed upload/download requests of large size. That is because the competition between the large requests is more intensive for client resources, which is consistent with our prior observations (see Section 4.1.2).

4.4.2. Interference of Large and Small Requests. To observe the interference among mixed-size upload requests, we set one daemon to run 4KB upload requests and the other one



(a) 4KB Upload Bandwidth (b) 4MB Upload Bandwidth

Fig. 14. Bandwidths of mixed 4KB/4MB upload.



(a) 4KB Upload Latency (b) 4MB Upload Latency

Fig. 15. Latencies of mixed 4KB/4MB upload.

to run 4MB upload requests; similarly, we set one daemon to run 4MB upload requests and the other to run 4KB upload requests to observe the interference among mixed-size upload requests. For brevity, we only present upload results here. Download requests show a similar trend.

The interference of small requests and large requests to the bandwidth. *Large requests are more sensitive to the interference*. For brevity, we only present the results of mixed upload requests. As shown in Figure 14(b), the bandwidth of large requests drops heavily when the interference parallelism is high. When the interference degree is low (<8), the bandwidth of 4MB upload requests can still increase, although the increase rate is much slower than that without upload interference. However, when the interference parallelism degree is high (>=8), the bandwidth of 4MB upload requests drops linearly. By contrast, as shown in Figure 14(a), the bandwidth of 4KB upload requests can still increase, although the increase rate is decreasing. The major reason is that small requests are CPU intensive, since small requests occupy a lot of CPU resources, and they have many chances to be scheduled and quickly completed during the competition. This takes away CPU resources from the large requests, which need more time to complete, and results in the observed significant bandwidth decrease.

The interference of small requests and large requests to latency. As shown in Figure 15, the average latencies of both small requests and large requests increase obviously when the parallelism degree and interference parallelism degree are high.

In particular, the average latency of large requests is also significantly affected by the interference of small requests, but not as significantly as the bandwidth is affected.

Summary: In this section, we investigate the interference among mixed requests on the *Baseline* client. We have observed two main phenomena: (1) large requests suffer more performance loss when being mixed with small requests, and (2) download requests are more sensitive to the competition of the corunning upload requests. The former is mostly caused by the competition of CPU resources, while the latter is mainly caused by the fact that the disk on the *Baseline* client in this case is a bottleneck and has faster read speed than write speed. Therefore, when client storage is not the bottleneck, the second phenomenon does not necessarily hold true. We have confirmed this by using *ramfs* on the *Baseline* client to repeat the mixed upload and download requests of 4MB and find that the second phenomenon is not obvious. This means that the interference among mixed requests is also client dependent.

4.5. Discussion

Through a comprehensive study on Amazon S3, we have investigated the I/O performance behaviors from a client's perspective. Besides common observations, we have also acquired several interesting and useful findings that have not been paid sufficient attention to in prior studies. In particular, we find that a proper combination of parallelism degree and request size helps fully exploit the great performance potential of cloud storage and optimize user experiences.

Our measurement work has several limitations. First, our current study is based on traditional clients, such as PCs and workstations. Ultra-mobile clients, such as smartphones and tablets, have very distinct characteristics, such as relatively weaker CPU, wireless networking, and flash-based storage. As we observed in the experiments, clients' properties are important and could lead to different performance perceived on the client side. Thus, we may have different observations on these mobile clients, which is worth a further study in the future. Second, the effect of clients' capabilities studied in this article focus more on CPU, memory, and storage. The network-related effect is mostly reflected in our study on geo-distance. More detailed analysis on the effect of networking capability can be found in prior work [Ou et al. 2015]. Third, we use Amazon S3 as our cloud storage service target for study. Although this work focuses more on performance analysis from the client's perspective, extending to other cloud storage services could be further studied in our future work.

5. DETERMINING PROPER COMBINATION OF PARALLELISM AND REQUEST SIZE

We have observed the tradeoff between parallelism and request size and their effects earlier. In this section, we make an attempt to identify a proper combination of parallelism degree and request size to find a "near-optimal" combination. Specifically, we propose a *sampling-* and *inference*-based approach.

5.1. Key Idea

The key idea is to leverage the known performance data of sample combinations (parallelism degree and request size) to make a speculation on proper combinations in a broader range, based on our understanding on the effects of parallelism and request size. This approach is primarily composed of two steps:

Sampling. We first assess the achievable performance for a set of sample combinations as our reference points for our speculation. Here the sample combinations refer to the combinations of typical request sizes (e.g., 16KB, 64KB, 256KB, 1MB, and 4MB) and parallelism degrees (e.g., from one job to 64 jobs). The performance data with different combinations can be obtained by purposefully running a simple test (as described in Section 3.3). We can also leverage some hints from the workload characteristics to

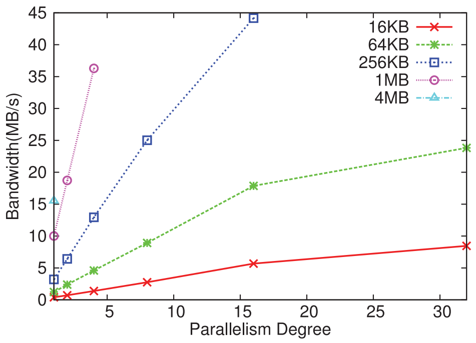


Fig. 16. Sampling for uploading 4MB object.

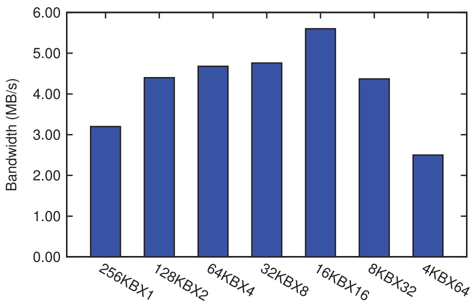


Fig. 17. Combinations for uploading 256KB object.

narrow down the sampling space and reduce the cost. For example, since web services are dominated by small objects, we can focus on small request sizes (e.g., 1KB, 4KB, 16KB, and 64KB) accordingly. We can also collect performance data by observing online traffic during runtime. Based on the obtained sample data, a profile can be created for the client and even shared among clients working in a similar environment.

Inferring. Based on the sample data, we can make a speculation and infer the proper combinations in a wider range, even without all test data available. Consider a simple example: if it is known that the combinations 4MB×4 and 1MB×8 can achieve the maximum uploading bandwidth on the client, we can speculate that the proper parallelism degree for 2MB is likely to be between 4 and 8, since it is too aggressive to select 8 and too conservative to select 4. Thus, a compromised choice is likely to be 6 (the average of 4 and 8). The rationale behind such a simple inference is that large requests are better to be combined with small parallelism degrees. In more complicated working scenarios, we can make the inference based on the performance curves of the sampled combinations.

For illustration, we present how to use this approach to achieve two different optimization goals, high bandwidth and low latency.

5.2. Bandwidth Optimization

The first optimization goal is to improve the overall *bandwidth*. There are two possible ways to enhance bandwidth: (1) chunking large objects into smaller ones to create more opportunities for parallelization or (2) merging small objects into larger ones to increase request size. Since the process of finding a proper combination for these two goals is similar, we take the first as an example.

For chunking, it is relatively easy to find proper combinations if the objects are large enough. On our testing platform (the *Baseline* client), if we upload a 256MB object, as shown in Figure 1(a), 4MB×4, 1MB×8, and 256KB×32 are regarded as "good" combinations, because they can achieve the peak bandwidth on the client.

It is more challenging when the objects are not large enough for creating combinations that can lead to the peak bandwidths. For such cases, the possible parallelism degrees are limited in a small range. For example, for uploading a 4MB object, the selectable parallelism degrees are limited: 1 (for 4MB chunks), 4 (for 1MB chunks), 16 (for 256KB chunks), and 64 (for 64KB chunks). As shown in Figure 16, 256KB×16 can achieve the highest bandwidth. Similarly, for uploading a 256KB object, 16KB×16 is the proper combination.

Evaluation. Figure 17 shows the bandwidth comparison of different combinations. Among these combinations, 256KB×1, 64KB×4, and 16KB×16 are sampling

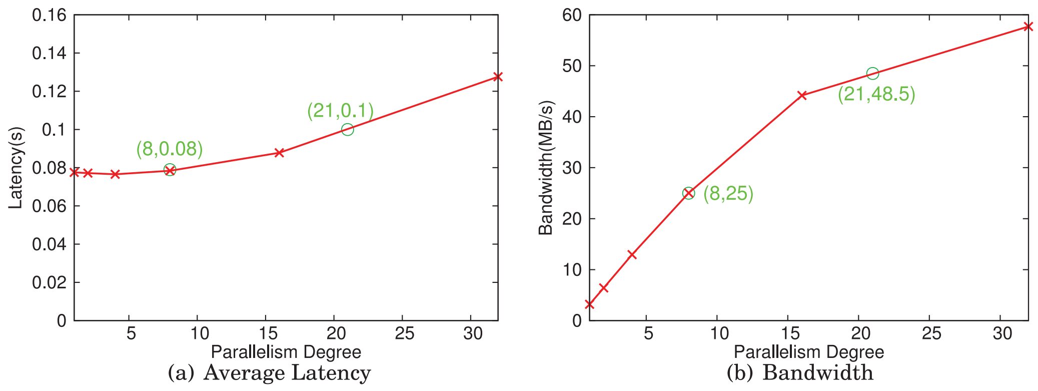


Fig. 18. An example of selecting a proper parallelism degree for uploading a set of 256KB objects. In Figure 18(a), the pair of numbers in parentheses presents the parallelism degree and the latency: the first number refers to the parallelism degree and the other refers to the latency in seconds. Similarly, in Figure 18(b), the first number in the parentheses presents the parallelism degree and the other presents the bandwidth in MB/s. In this example, the samples are the combinations of request size (256KB) and parallelism degrees (1, 2, 4, 8, 16, and 32).

combinations shown in Figure 16; 128KB×2, 32KB×8, 8KB×32, and 4KB×64 are additional tested combinations. From the bandwidth growing tendency of different combinations, the combination of 16KB×16 is a proper selection for uploading a 256KB object, which confirms our inference.

5.3. Latency Optimization

Another optimization goal is *latency*. Some applications may be in need of improving throughput but have a low tolerance to latency increase. We take the case of uploading a set of 256KB objects as an example to explain how to use the *sampling*- and *inference*-based approach to decide a proper parallelism degree.

The latency and bandwidth achieved by the sample combinations (256KB with parallelism degrees ranging from 1 to 32) are shown in Figure 18. With a single thread, the average latency of uploading 256KB objects is 0.078s, and the bandwidth is 3.2MB/s. If the application prefers to minimize the latency increase caused by parallelization, the parallelism degree 8 may be a proper choice (see Figure 18(a)), leading to a possibly maximum bandwidth of 25MB/s (see Figure 18(b)). If the applications are more tolerant to latency increase, a higher bandwidth can be achieved. For example, if the average latency is allowed to be increased to 0.1s, shown in Figure 18(a), the parallelism degree of 21 jobs can be selected, and the corresponding bandwidth is 48.5MB/s (see Figure 18(b)).

Evaluation. To evaluate the accuracy of our inference on the combination of 256KB×21, using our test tool, we conduct an experiment to measure the performance of the combination. In the experiment, 21 threads are created to upload 20,000 objects of 256KB in parallel. We repeat the experiment 5 times. The comparison of the inferred performance and the measured performance is shown in Figure 19. As measured, the average latency is 0.1002 second with the standard error of 0.00606 second, and the bandwidth is 50.11MB/s with the standard error of 2.92MB/s. Comparatively, our inference (0.1s for average latency and 48.5 MB/s for bandwidth) is quite accurate.

5.4. Discussion

We have shown how to adopt the *sampling*- and *inference*-based method to optimize for two different goals, latency and bandwidth. In fact, it is also possible to be used to

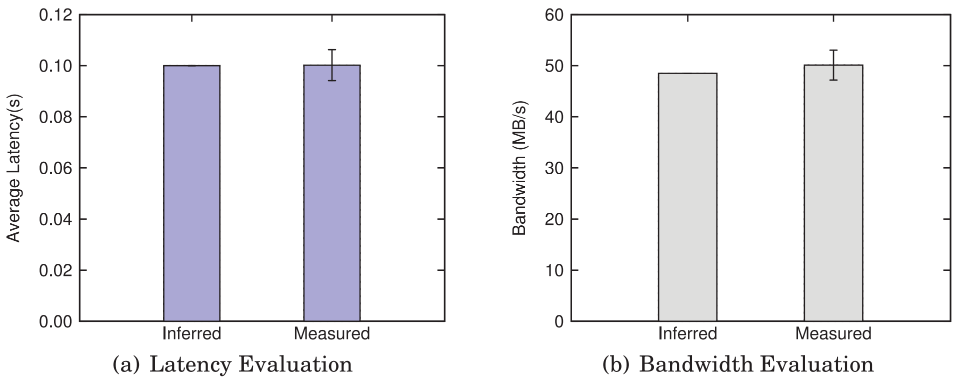


Fig. 19. The evaluation of an inferred combination 256KB×21. The `Inferred` bar presents the inferred value of the combination. The `Measured` bar presents the measured results with a range.

estimate a proper combination by using the profile collected from one client for another. For example, if we have the sample combination that 8 is the proper parallelism for 16KB on the client with 2 units of CPU capability, we can speculate that a proper combination for the client with 4 units of CPU capability is likely to be 16KB×16, since the CPU resource is doubled. We can also speculate that the proper parallelism degree for 16KB should not be smaller than 16 for the client that is more distant from the cloud, since a long distance requires more aggressive parallelization. Similarly, our observations on the interference among mixed requests also could provide us hints to infer a proper combination from one case to another.

We would also like to point out here that this *sampling-* and *inference*-based approach cannot warrant a perfect estimation for finding an optimal combination; however, it shows that there exists a feasible method to make a reasonable speculation based on the data that we have. In practice, client-side applications also need to consider other factors, such as caching and prefetching on the client side, which may affect the accuracy and make the inference more complicated to make a proper decision. We will further present our case studies on cloud-based applications in Section 6.

6. CASE STUDIES

In this section, we present a set of case studies in typical application scenarios to show the benefits and challenges of exploiting the I/O characteristics of cloud storage in practice. It is worth noting that our goal is not to provide a complete design of solutions to address the specific problems but to show via these case studies how the I/O characteristics of cloud storage may offer new opportunities to deal with the performance issues of cloud-based storage systems. We carefully select three data-intensive applications running in the scenarios of cloud storage to investigate how to exploit the I/O characteristics of cloud storage.

Applications: The applications we select are `grep`, `tar`, and `filesystem`. From the perspective of performance, the first two applications are more bandwidth oriented for achieving a short overall execution time, while the last requires low average request latencies. By investigating these cloud-based applications, we focus on studying how to properly take advantage of chunking and parallelization to optimize classic client-side techniques including informed prefetching, data synchronization, and caching and prefetching for file systems with the workloads collected from the real world.

Platform: In these case studies, we use Amazon S3 (in Oregon) as the cloud storage services and a workstation on our campus (in Louisiana) as the client. The client is a

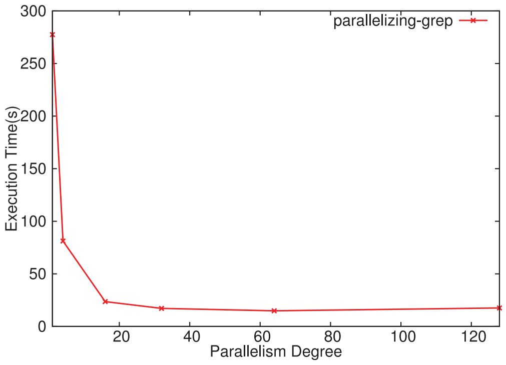


Fig. 20. Execution time of `grep` with different parallelism degrees.

PC equipped with a two-core 3.3GHz Intel Core i5 CPU, 8GB memory, and a 450GB disk drive, and installed with Ubuntu 12.04.5 LTS and Ext4 file system.

6.1. Parallelizing Informed Prefetching

Informed prefetching refers to the prefetching method that is based on hints given by the applications. Since the to-be-accessed dataset is informed in advance, we can sufficiently parallelize the downloading process to efficiently load the data from the cloud to the client.

To investigate the performance of parallelizing cloud storage I/Os for informed prefetching, we choose `grep`, one of the most typical and common applications that can benefit from informed prefetching, to showcase the effect of parallelizing prefetch I/Os. The function of `grep` is to search for a certain string in a dataset (a file or a directory) that is explicitly given as a parameter. Specifically in the scenario of cloud storage, since the cloud storage providers generally do not support content-based searching APIs in the cloud servers, the data to be searched need to be downloaded to the client first. Once a `grep` application is launched, we can get the path of the dataset from the command parameters and thus download the dataset in parallel if necessary.

In our experiments, we use `grep` to search for the string "prefetch" in the directory "linux-4.3/fs," which is a subdirectory of the source code tree of the Linux kernel (version 4.3) stored in the cloud. The execution time of `grep` with different parallelism degrees (see Figure 20) shows that via sufficiently parallelizing the downloading processes, the overall execution time can be reduced up to 94.65% (277.51s vs. 14.86s). This is in accordance with our expectation. Since cloud storage is parallelization friendly and the average size of files in the dataset is small (18.44KB), parallelizing the downloading requests can significantly improve the bandwidth and thus reduce the overall downloading time. Comparatively, the time taken by the `grep` application to complete the searching process after the dataset is loaded to the client is trivial (about 0.05s).

We also note that when the parallelism degree is excessively high (e.g., 128), the execution time increases by 18.2% compared to the lowest time (17.56s vs. 14.86s). However, the execution time with overparallelization is still much lower than that under insufficient parallelization. For example, the execution time with parallelism degree 128 is 24.3% lower than that with parallelism degree 32 (17.56s vs. 23.59s) and 93.7% lower than that with a single thread (17.56s vs. 277.51s). Therefore, for applications like `grep` aiming at high bandwidth, a relatively high parallelism degree is acceptable. To achieve the optimal parallelism degree, we can also analyze the average file size of the dataset before deciding the parallelism degree. For example, on our client, for a small file size (e.g., 4KB), we may use a relatively high parallelism (e.g., 64);

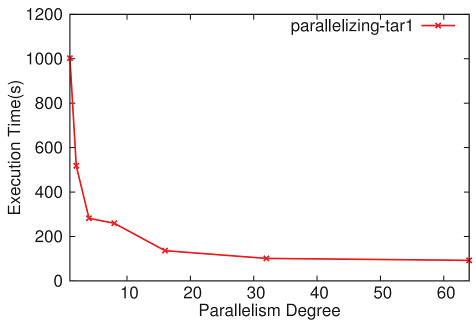


Fig. 21. Execution time of `tar` (Trace 1).

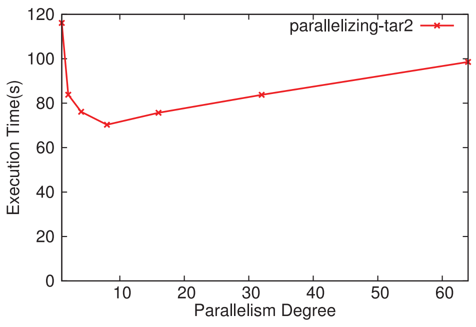


Fig. 22. Execution time of `tar` (Trace 2).

for a large file size (e.g., 2MB), the parallelism degree can be relatively low (e.g., 8). In the meantime, the overhead of analyzing the dataset that is related to the metadata operations (e.g., `HEAD`) to the cloud should not be ignored. Specifically in this case, it takes about 1.36s to get the metadata of the files and calculate the average file size, which is not significant but deserves our attention.

6.2. Parallelizing Synchronization

For many cloud-based applications, the data is maintained in a local directory called the "syncing directory" and the client periodically syncs local data to the cloud. In such applications, the most frequent operation is to upload data from the client to the cloud, which is generally called "sync." Therefore, the efficiency of sync is critical to user experiences. Let us consider a typical use case: when we execute a `tar` command to unpack a compressed file (e.g., ".tar," ".tgz," and ".zip") in the syncing directory of a cloud storage application (e.g., Google Drive, Dropbox, and OneDrive), a bunch of unpacked files need to be uploaded to the cloud. Similar to our parallelization for informed prefetching, parallelizing the uploading process would dramatically improve the bandwidth and thus reduce the overall execution time.

To further illustrate this, we collect two practical traces of `tar` applications: *Trace 1* is to unpack a ".tar" file that contains 5,233 text documents and images of a latex project; *Trace 2* is to unpack a ".tar" file that contains 100 ".pdf" files of typical conferences and journals. Via replaying these two traces in the syncing directory, we can investigate the effects of parallelizing sync operations.

The overall execution time of Trace 1 (Figure 21) shows that as the parallelism degree increases, the overall execution time continues to decrease. In contrast to working with a single thread, the overall execution time decreases by 98.2% (92.71s vs. 1002.85s). That is because the average size of the documents of the latex project in our experiments is rather small (60.5KB), and parallelizing the uploading process can achieve much higher bandwidth, and thus significantly lowers the overall execution time. Similar to parallelizing the informed prefetching process, the performance degradation caused by overparallelization in this case is not significant.

The overall execution time of Trace 2 with different parallelism degrees is shown in Figure 22. When the parallelism degree is low (<=8), the overall execution time decreases significantly. For example, the execution time with parallelism degree 8 decreases by 39.5% (70.29s vs. 116.13s). However, different from the results of Trace 1, the overall execution time of Trace 2 increases linearly when the parallelism degree exceeds 8 (see Figure 22). That is mostly because the average file size in Trace 2 is relatively large (2.64MB). Thus, the performance degradation caused by overparallelization is significant.

The results of Trace 1 and Trace 2 demonstrate that parallelization is the key to improving the sync performance; however, when the request size is relatively large, the negative effects caused by overparallelization should be given sufficient attention. In addition to the performance degradation, the waste of the client resources should not be ignored either. For example, the CPU utilization with parallelism degree 64 is 61% higher than that with parallelism degree 2 (100% vs. 62%); however, the execution time with parallelism degree 64 is 17% higher than that with parallelism degree 2 (116.13s vs. 98.58s). This means overparallelization consumes more client resources but results in worse performance.

Similar to parallelizing informed prefetching, analyzing the average file size of the dataset before deciding the parallelism degree is a feasible way to avoid overparallelization. However, comparatively, the cost of analyzing the dataset for sync is trivial. That is because the data to be uploaded resides in the local client and can be traversed and analyzed directly without communicating with the remote repository on the cloud like the `grep` application. In this case, it only takes several milliseconds to analyze the dataset (Trace 1 and Trace 2), which is negligible to the overall execution time.

6.3. Caching and Prefetching for File Systems

In cloud storage, client-side caching and prefetching are two basic schemes for enhancing the user experience. In this case study, we make attempts to show that cloud storage I/O performance could be affected by optimizing caching and prefetching. Specifically, we will discuss two key techniques, *chunking* and *parallelization*.

6.3.1. Experimental Setup. To evaluate the effects of chunking and parallelized prefetching for cloud-based file systems, we build an emulator to implement the basic read/write operations of a typical cloud-based file system with disk caching on the client. In the emulator, we focus on two requests: `read` and `write`. Each request has three parameters: `file_id`, the unique ID of the target file; `offset`, the offset of the first byte of requested data in the file; and `length`, the length of the requested data.

Upon each request, the simulator works as follows: (1) it first checks the `file_id` of the object in the client cache; (2) upon a cache hit, it returns the requested data by accessing the cached object; and (3) upon a cache miss, it first downloads the object from the cloud to the client disk cache, and then returns the requested data by accessing the object in local cache.

The local cache is managed by the standard LRU cache replacement algorithm. Similar to the Linux write-back policy, the emulator uploads the dirty files that have resided on the local cache for more than 30 seconds periodically (every 5 seconds) by a background daemon. The emulator also collects the latency of each request (end-to-end completion time) and the number of cache hits. After all the requests are completed, it reports the average access latency and the cache hit ratio.

To drive this experiment, we use an object-based trace by converting a segment of an NFS trace, which is a mix of email and research workload collected at Harvard University [Ellard et al. 2003]. We extract the object sizes and the sequence of read and write requests from the trace. The total volume of referenced data is 4.8GB, the average file size is 12.9MB, and the distribution of file sizes is shown in Figure 23.

6.3.2. Proper Chunk Size for Caching. Chunking is an important technique used in cloud storage. In S3Backer, for example, the space of the cloud-based block driver is formatted with a fixed block size that can be defined by the user [S3Backer 2015]. The choice of chunk sizes can affect caching performance: the smaller the chunk is, the less a cache miss cost would be, but the more cloud I/Os could be generated.

Although it is difficult to accurately determine the optimal chunk size, our findings about the effect of chunk size on the performance of cloud storage can guide us to

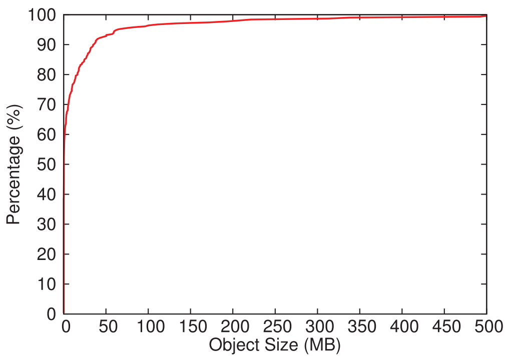


Fig. 23. CDF of file sizes.

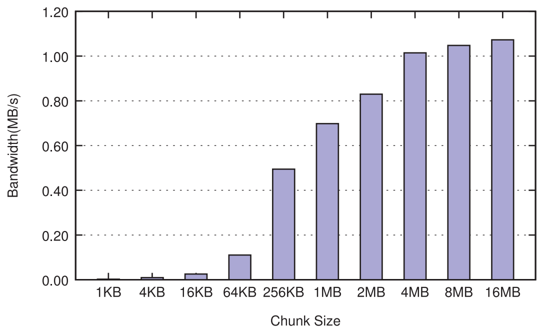


Fig. 24. Download bandwidths of different chunk sizes with a single thread.

roughly choose a proper, if not optimal, chunk size. Specifically, we aim to identify a relatively small chunk size for reaching an approximately maximum bandwidth by making a reasonable tradeoff between the cache hit ratio and cache miss penalty.

We first examine the bandwidth of typical chunk sizes (from 1KB to 16MB) with a single thread. Figure 24 shows that when the chunk size exceeds 4MB, the download bandwidth is close to the maximum achievable bandwidth with a single thread. Based on this, we speculate that the proper chunk size is possibly around 4MB. This is for two reasons. First, further increasing the chunk size over 4MB (e.g., 8MB or 16MB) cannot deliver a higher bandwidth. For example, upon a cache miss of 8MB data, downloading one 8MB chunk takes almost an equal amount of time as downloading two 4MB chunks, while using 8MB chunks increases the risk of downloading irrelevant data. Second, if the chosen chunk size is excessively smaller than 4MB (e.g., 64KB or 1MB), the cache may suffer from a high cache miss ratio and cause too many I/Os.

To verify this speculation, we adopt the standard LRU algorithm with asynchronous writeback (for the purpose of generality). Every 30 seconds, we flush dirty data back to the cloud. The cache size is set as 200MB disk space. Besides a 4MB chunk size, for a comparison, we choose two smaller chunk sizes, 64KB and 1MB, and two larger chunk sizes, 8MB and 16MB, to study the effect of the chunk sizes.

The average access latencies with different chunk sizes are shown as Figure 25. It clearly shows that the lowest average read/write latencies are achieved at 4MB, which confirms our speculation. When the chunk size increases from 64KB to 4MB, the average read latency decreases by 47.3% (from 95.2ms to 50.2ms), and the average write latency decreases by 40.4% (from 109.9ms to 65.5ms). This benefit is due to the increase of cache hit ratio: The read hit ratio increases from 77.8% to 98.4%, and the

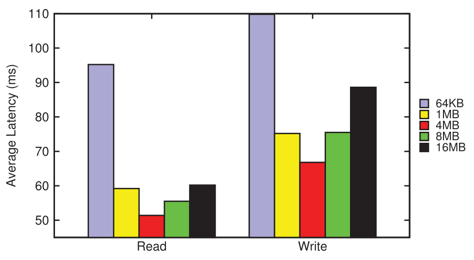


Fig. 25. Average latencies with different chunking.

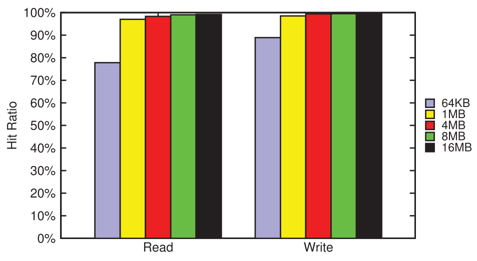


Fig. 26. Hit ratios with different chunking.

write hit ratio increases from 88.9% to 99.4% (see Figure 26). This is mostly because using a relatively large chunk size allows preloading the useful data, and consequently improves the cache hit ratio and the overall performance. However, when the chunk size exceeds a certain threshold, further increasing chunk size may cause undesirable negative effects. Figure 26 shows that the cache hit ratios increase slightly with a large chunk size. The increased cache miss penalty with a large chunk size is responsible for the slowdown. Specifically, it takes 4s to load a 4MB chunk, while it needs 14.2s for a 16MB chunk. Consequently, the average access latencies increase. Additionally, the interference among uploading and downloading threads may also increase the average access latencies in the case of overparallelization (i.e., large request size with high parallelism degree).

The previous analysis has shown how to determine the proper chunk size for a certain client. Specifically, 4MB is the proper chunk size on our client for the testing workload. For the workloads with weak spatial locality, the proper chunk size should be correspondingly smaller. In general, an excessively large chunk size is not desirable, as it increases the risk of unnecessary overhead with no extra benefit.

6.3.3. Proper Parallelization for Prefetching. Prefetching is another widely used technique in cloud storage clients. Since objects can be downloaded (prefetched) in parallel, a proper parallelism degree can enhance performance, but overparallelization raises the risk of misprefetching and resource waste.

In order to determine a proper parallelism degree for a certain chunk size, it is important to ensure that on-demand fetching would not be significantly affected by prefetching. To avoid a significant increase of average fetching latencies, we can perform an exhaustive search on the client, which is feasible but inefficient. Based on our findings, in fact, we can greatly simplify the process of identifying a proper parallelism degree. To show how to achieve this, we take the chunk sizes (64KB, 1MB, and 4MB) as examples. We may first choose a 4MB chunk with parallelism degree 1 and then gradually increase the parallelism degree step by step (2, 4, and 8) for testing. For smaller chunk sizes, we only need to test from a larger parallelism degree, since small chunks are more parallelism friendly and it is unlikely to achieve higher performance at a low parallelism degree as large chunks. Figure 27 gives such an example: four parallel jobs for 4MB, eight parallel jobs for 1MB, and 16 parallel jobs for 64KB are the best choices.

To illustrate the actual effect of parallelization to prefetching, we implement an adaptive prefetching algorithm in our emulator. We adopt the history-based prefetching window to determine the prefetching granularity, which is similar to the file prefetching scheme used in the Linux kernel. A prefetching window is maintained to estimate the best prefetching degree. The initial window size is 0 and enlarged based on the detected sequentiality of observed accesses. Assuming chunk `n` of an object is requested, if chunk `n-i`, chunk `n-i+1`, ... , and chunk `n-1` ($1 \le i \le n$) are detected to be sequentially accessed,

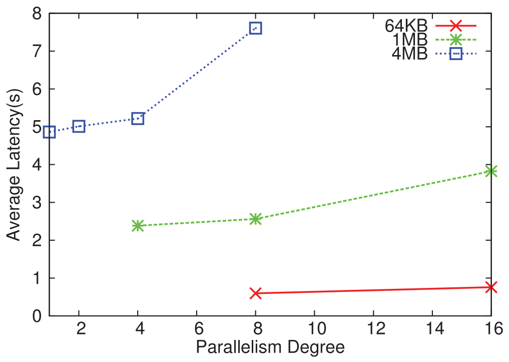


Fig. 27. Average download latencies with different parallelism degrees.

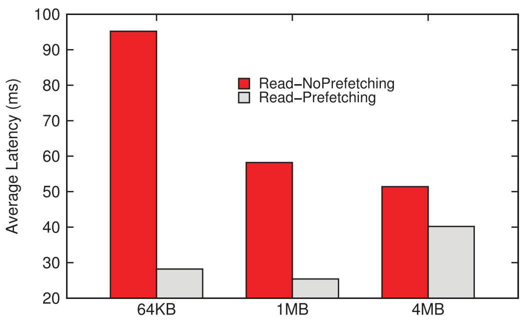


Fig. 28. Average read latency comparison.

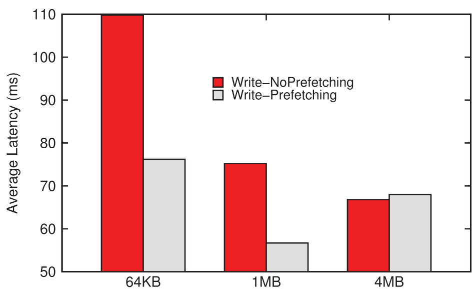


Fig. 29. Average write latency comparison.

Table VII. Average Latency Reduction by Prefetching

Chunk Size	Read Latency Reduction	Write Latency Reduction
64KB	70.4%	31.1%
1MB	56.9%	24.6%
4MB	22.6%	−1.2%

the size of the prefetching window grows to 2^{i-1}. We set the maximum prefetching window size (i.e., parallelism degree of prefetching) for all chunk sizes (i.e., 64KB, 1MB, 4MB) to 8.

The performance comparisons of no-prefetching and prefetching are shown in Figure 28 and Figure 29. We can see that, with prefetching, the optimal chunk size is 1MB. Obviously, a small chunk size benefits more from the prefetching (as we see in the prior sections, small objects benefit more from parallelism), and the relative benefits decrease as the chunk size increases (see Table VII).

Surprisingly, with prefetching, the average write latency of 4MB increases by 1.2%. This means that the prefetching granularity in our experiment is so aggressive that the negative effects of prefetching outweigh the benefits. The negative effects may result from two factors. First, a lot of unnecessary data is prefetched so that the cache efficiency is reduced, leading to a lower cache hit ratio. Second, the competition among parallel prefetching threads may increase the average downloading latency (i.e., the average penalty of cache miss). Specifically for the case of the average write latency of 4MB, the performance degradation is mainly caused by the second factor since the cache hit ratio remains high (close to 98.7%). As a rule of thumb, we should set a small prefetching degree for large chunk sizes (e.g., 4MB) to avoid the intensive competitions among the parallelized downloading threads. For example, we can limit the growing speed of the prefetching window or cap the maximum prefetching window size. On the contrary, the prefetching granularity of small chunk sizes (e.g., 64KB) can be more aggressive. This also confirms our speculation about the proper parallelism degrees for different chunk sizes (i.e., 16 for 64KB chunks, 4 for 4MB chunks).

In summary, our case studies further show that the real-world implementation of client-side management should carefully consider the factors that we have studied in the prior sections, particularly parallelism degree and request size. Other issues, such as client capabilities and geo-distances, would also inevitably further complicate the design consideration.

7. SYSTEM IMPLICATIONS

With these experimental observations, we are now in a position to present several important system implications. This section also provides an executive summary of our answers to the questions we asked earlier.

Appropriately combining request size and parallelism degree can maximize the achievable performance. This is sometimes a tradeoff between the two factors. By combining the chunking/bundling methods with parallelizing I/Os, the client can enhance bandwidth in two different ways: we can increase the parallelism degree for small requests or increase the request size at low parallelism degree. Both can achieve comparable bandwidth, but interestingly, we also find that compared to increasing parallelism degree, increasing the request size can achieve another side benefit: reduced CPU utilization. This means that for some weak-CPU platforms, such as mobile systems, it is more favorable to create large requests with a low parallelism degree. On the other hand, we should also consider several related side effects of bundling/batching small requests. For instance, if part of a bundled/batched request failed during the transmission, the whole request would have to be retransmitted. Also, it is difficult to pack a bunch of small requests to different buckets or data centers together. In contrast, parallelizing small requests is easier and more flexible. Therefore, there is no clear winner between the two possible optimization methods (i.e., creating large requests and parallelizing small requests). An optimal combination may vary from client to client in terms of clients' capabilities and geographical distance, but the *sampling*- and *inference*-based method helps make a proper decision (see Section 5).

The client's capability has a strong impact on the perceived cloud storage I/O performance. CPU, memory, and storage are the three most critical components determining a client's capability. Among the three, CPU plays the most important role in parallelizing small requests, while memory and storage are critical to large requests, especially large download requests. A direct implication is that for optimizing the cloud storage performance, we must also distinguish the capabilities of clients, and one policy will not be effective for all clients. Due to the cross-platform advantage, many personal cloud storage applications can run on multiple platforms (from PCs to smartphones). Such distinction among clients will inevitably affect our optimization policies. For example, for a mobile client with a weak CPU, we should avoid segmenting objects into excessively small chunks, since it is unable to handle a large number of parallel I/Os, although this is not a constraint for a PC client. Given the diversity of cloud storage clients, we believe that a single optimization policy is unlikely to succeed across all clients.

Geographical distance between the client and the cloud plays an important role in cloud storage I/Os. For cloud storage, the geographical distance determines the RTT. We find that a long RTT has distinct effects on bandwidth and latency. In particular, with a long RTT, we still can achieve a similar peak bandwidth as the case of a short RTT, but the cloud I/O latency is significantly higher. The implications are twofold. First, to tackle the long latency issues, it is a must-have to use effective caching and prefetching for latency-sensitive applications. Second, for the clients far from the cloud, we should proactively adopt large request sizes and high parallelism degrees to fully saturate the pipeline and exploit available bandwidth as much as we can. In other words, by sufficiently exploiting the I/O characteristics of cloud storage, if bandwidth is the main requirement (e.g., video streaming), choosing a relatively distant data center of the cloud storage is a viable option and a high bandwidth is still achievable with appropriate client-side optimizations.

Parallel cloud storage I/Os generated by a client may interfere with each other, and the effect is workload and client dependent. We find that large requests and download requests are particularly sensitive to the interference caused by their corunners, small requests (competing for client CPU resources), and upload requests (competing for client storage bandwidth), respectively. This implies that a scheduling policy for cloud storage I/Os is needed to avoid intensive interference, particularly when client CPU or client storage is the bottleneck. We may batch and parallelize requests sharing similar patterns, rather than randomly dispatch requests, just like some local I/O schedulers. Such a mixed effect also has implications for the client to choose caching or syncing for optimizations. Most personal cloud storage clients adopt the syncing approach. Since the client maintains a complete copy of the data, and only changes are uploaded to the cloud periodically in batches, the traffic pattern is relatively simple and the interference among mixed requests is expected to be low. In contrast, for a caching approach, cache misses will trigger download requests, which may conflict with the upload requests generated by periodic flushing, and this will lead to the interference. Also, if a cloud-based system adopts an adaptive chunking policy, the interference between small and large requests should also be considered.

In essence, cloud storage represents a drastically different storage model for its diverse clients, network-based I/O connection, and massively parallelized storage structure. Our observations and analysis strongly indicate that fully exploiting the potential of cloud storage demands careful consideration of various factors.

8. RELATED WORK

Most prior studies focus on addressing various issues of cloud storage, including performance, reliability, and security (e.g., Abu-Libdeh et al. [2010], Bonvin et al. [2010], Brad et al. [2011], Chen et al. [2014], Cui et al. [2015], Ford et al. [2010], Gulati et al. [2011], Higgins et al. [2012], Hu et al. [2012], Li et al. [2013], and Zhang et al. [2011, 2014]). Some other work studies the design of cloud-based file systems to better integrate cloud storage into current storage systems (e.g., Bessani et al. [2014], Dong et al. [2011], and Vrable et al. [2012]). Our work is orthogonal to these studies and focuses on understanding the behaviors of cloud storage from the client's perspective.

Our work is related to several prior measurement works on cloud storage. Li et al. [2010] compared the performance of major cloud providers: Amazon AWS, Google App-Engine, and Rackspace CloudServers. Bermudez et al. [2013] presented a characterization of Amazon's Web Services (AWS). Ou et al. [2015] compared a CloudFuse-based filesystem for OpenStack Swift, an open-source cloud storage, with two other IP-based storage, NFS and iSCSI. Cooper et al. [2010] benchmarked cloud storage systems with YCSB. Meng et al. [2010] presented a benchmarking work on cloud-based data management systems to evaluate the effects of different implementations on cloud storage. Bocchi et al. [2014] presented a comprehensive performance comparison of four cloud storage services, namely, Amazon S3, Amazon Glacier, Windows Azure Blob, and Rackspace Cloud Files, by using generic workloads. He et al. [2013] investigated how modern web services use Amazon EC2 and Windows Azure as their infrastructure and tried to identify ways to improve the deployments of the cloud-based services.

These prior studies investigate the performance of cloud storage mostly from the perspective of the server side. Our work, unlike these server-oriented studies, focuses on studying the role of clients in the observed end-to-end cloud storage performance. In addition to some common observations, such as the effect of parallelism on achievable bandwidth [Ou et al. 2015], we have made findings specifically from a client's perspective. For example, in our preliminary study [Hou et al. 2016], we have already found that a reasonable tradeoff could be made between parallelism degree and request size for achieving the desirable performance in terms of bandwidth and latency. In this article, we further present a sampling- and inference-based approach to demonstrate that it is feasible to identify a proper combination of parallelism degree and request size. We have also studied the important factors affecting the parallelization of cloud storage I/Os, such as the interference among corunning cloud storage I/Os. As for the effect of geographical distance, we also show that the advantage of a short geo-distance for accessing the cloud may be offset by the negative effect caused by overparallelization. Additionally, the client's capabilities should not be ignored when deciding specific optimization policies, and proper optimization on the client side can be much more efficient and effective to improve user experiences than selecting a faster or a closer (maybe more expensive) cloud storage provider. Our case studies, such as parallelizing the informed prefetching and synchronization, further illustrate possible ways of leveraging parallelization for various system optimization purposes.

Several other measurement works studied the cloud storage client applications (e.g., Drago et al. [2013, 2014, 2012], Hu et al. [2010], Mager et al. [2012], Wang et al. [2012], Gracia-Tinedo et al. [2013], and Bocchi et al. [2015]). These studies focus on measuring the performance of client applications of cloud storage, which are mostly proprietary products, such as Dropbox, Wuala, Google Drive, Box, SugarSync, and so forth. Different from these studies, our goal is not to test a specific client application; rather, we aim to reveal the key factors affecting the data exchange between the client and the cloud from a client's perspective with controlled comparison, from which we hope to gain insight as guidance for optimizing application design on the client side. Therefore, our testing tool is designed to observe the direct communication to the cloud and purposefully bypasses any client-side optimization techniques (e.g., caching, prefetching, compression, deduplication).

Some observations reported in prior work also imply that our findings can be used to improve the design and implementation of cloud storage clients. For example, as reported by Bocchi et al. [2015], among the 11 tested cloud storage client applications (including Dropbox, Google Drive, Box, Copy, etc.), seven applications adopt the approach of splitting large files into fix-sized smaller chunks, while others do not chunk files. Our findings suggest that proper combination of chunk size and parallelism degree can accelerate data transmission, and such a combination is supposed to be client dependent. Our case studies on several typical working scenarios of cloud storage further demonstrate the efficiency and effectiveness of our findings for optimization on the client side of cloud storage for data-intensive workloads.

Some prior work studied the performance variance issues of cloud storage. For example, Wu et al. [2015] characterized the latency variance of accessing cloud storage based on the observations on Amazon S3 and Windows Azure, and proposed CosTLO, which is a system aiming to reduce such variance by using different levels of redundancy. It mostly focuses on access latency and does not consider the effects of clients' capabilities and client-side optimizations (parallelism and request size). In contrast, our work focuses on characterizing both bandwidth and access latency and also investigates the effects of various and particularly client-related factors. As for optimizing user experiences, we also show a sampling- and inference-based method to determine a proper combination of parallelism degree and request size on the client side. This is also different from CosTLO, which adopts data redundancy on the server side of cloud storage to reduce latency variance.

Besides object-based storage, some service providers also provide block- and file-level storage services, such as Amazon Elastic Block Store (EBS) [Amazon 2015a] and Elastic File System (EFS) [Amazon 2015b]. These services are more similar to conventional IP-based storage, such as iSCSI and NFS. Although this work focuses on the HTTP-based object storage, represented by Amazon S3, the other cloud-based services represented by EBS and EFS are worth further studies in the future.

9. CONCLUSIONS

We present a comprehensive experimental study on Amazon S3, a typical cloud storage provider, to investigate the critical factors affecting the perceived performance of cloud storage from a client-side perspective. Our experiments show several important characteristics of cloud storage, such as benefits of parallelizing cloud storage I/Os, the latency impact of overparallelization, the effect of clients' capability on achievable performance, the interference among corunning requests, and more. Based on these findings, we propose a sampling- and inference-based method to determine a proper combination of parallelism degree and request size. We also present a set of case studies in typical cloud storage scenarios. We hope our observations and the associated system implications can provide guidance to help practitioners and application designers exploit various optimization opportunities for cloud storage clients.

REFERENCES

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)*. Indianapolis, IN.

Amazon. 2010. Amazon S3 Object Size Limit Now 5 TB. Retrieved from https://aws.amazon.com/blogs/aws/amazon-s3-object-size-limit/.

Amazon. 2015a. Amazon EBS. Retrieved from https://aws.amazon.com/ebs/.

Amazon. 2015b. Amazon EFS. Retrieved from https://aws.amazon.com/efs/.

Amazon. 2015c. Amazon S3. Retrieved from https://aws.amazon.com/s3/.

Amazon. 2015d. Amazon S3 TCP Window Scaling. Retrieved from http://docs.aws.amazon.com/AmazonS3/latest/dev/TCPWindowScaling.html.

Andreas Bergen, Yvonne Coady, and Rick McGeer. 2011. Client bandwidth: The forgotten metric of online storage providers. In *Proceedings of 2011 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PacRim'11)*.

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 IEEE International Conference on Computer Communications (INFOCOM'13)*.

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)*.

Enrico Bocchi, Idilio Drago, and Marco Mellia. 2015. Personal cloud storage benchmarks and comparison. *IEEE Transactions on Cloud Computing* 99 (2015), 1–14.

Enrico Bocchi, Marco Mellia, and Sofiane Sarni. 2014. Cloud storage service benchmarking: Methodologies and experimentations. In *Proceedings of the 2014 IEEE 3rd International Conference on Cloud Networking (CloudNet'14)*. 395–400.

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)*.

Boto. 2015a. An Introduction to Boto's S3 Interface. Retrieved from http://boto.readthedocs.org/en/latest/s3_tut.html.

Boto. 2015b. S3 API Reference. https://boto.readthedocs.org/en/latest/ref/s3.html.

Calder Brad, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, and Yikang Xu et al. 2011. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles (SOSP'11)*. 119–132.

Feng Chen, David A. Koufaty, and Xiaodong Zhang. 2009. Understanding intrinsic characteristics and system implications of flash memory based solid state drives. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS'09)*. ACM Press.

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)*.

Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC'10)*. ACM Press.

Yong Cui, Zeqi Lai, Xin Wang, Ningwei Dai, and Congcong Miao. 2015. QuickSync: Improving synchronization efficiency for mobile cloud storage services. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking (MobiCom'15)*. 582–603.

Xiaoning Ding, Song Jiang, Feng Chen, Kei Davis, and Xiaodong Zhang. 2007. DiskSeen: Exploiting disk layout and access history to enhance I/O prefetch. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC'07)*. USENIX Association.

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. In *SIGOPS Operating System Review*, Vol. 45. 101–111.

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

Idilio Drago, Enrico Bocchi, Macro Mellia, Herman Slatman, and Aiko Pras. 2014. Modeling the dropbox client behavior. In *Proceedings of the 2014 IEEE International Conference on Communications (ICC'14)*.

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 Conference on Internet Measurement Conference (IMC'12)*.

Dropbox. 2015. Dropbox. Retrieved from https://www.dropbox.com/.

Daniel Ellard, Jonathan Ledlie, Pia Malkani, and Margo Seltzer. 2003. Passive NFS tracing of email and research workloads. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST'03)*. USENIX Association.

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)*.

Google. 2015. Google Drive. https://www.google.com/drive/.

Raul Gracia-Tinedo, Marc Sanchez Artigas, Adrian Moreno-Martinez, Cristian Cotes, and Pedro Garcia Lopez. 2013. Actively measuring personal cloud storage. *Proceedings of the 2013 IEEE 6th International Conference on Cloud Computing (CLOUD'13)*. 301–308.

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)*.

Keqiang He, Alexis Fisher, Liang Wang, Aaron Gember, Aditya Akella, and Thomas Ristenpart. 2013. Next stop, the cloud: understanding modern web service deployment in EC2 and azure. In *Proceedings of the 2013 Conference on Internet Measurement Conference (IMC'13)*. ACM, 177–190.

Brett D. Higgins, Jason Flinn, T. J. Giuli, Brian Noble, Christopher Peplin, and David Watson. 2012. Informed mobile prefetching. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys'12)*. 155–158.

Binbing Hou, Feng Chen, Zhonghong Ou, Ren Wang, and Michael Mesnier. 2016. Understanding I/O performance behaviors of cloud storage from a clients perspective. In *Proceedings of the 32nd International Conference on Massive Storage Systems and Technology (MSST'16)*.

Wenjin Hu, Tao Yang, and Jeanna N. Matthews. 2010. The good, the bad and the ugly of consumer cloud storage. In *ACM SIGOPS Operating Systems Review*, Vol. 44:3.

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)*.

Liu Huan. 2002. A trace driven study of packet level parallelism. In *Proceedings of the IEEE International Conference on Communications (ICC'02)* 4, 1, 2191–2195.

IHS. 2012. Subscriptions to Cloud Storage Services to Reach Half-Billion Level This Year. Retrieved from https://technology.ihs.com/410084/subscriptions-to-cloud-storage-services-to-reach-half-billion-level-this-year.

Van Jacobson, Robert Braden, Dave Borman, M. Satyanarayanan, J. J. Kistler, L. B. Mummert, and M. R. Ebling. 1992. RFC 1323: TCP Extensions for High Performance.

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)*. USENIX Association.

Ang Li, Xiaowei Yang, and Ming Zhang. 2010. CloudCmp: Comparing public cloud providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC'10)*. ACM Press.

Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y. Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. 2013. Efficient batched synchronization in dropbox-like cloud storage services. In *Proceedings of International Middleware Conference (Middleware'13)*.

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)*.

MarketsandMarkets. 2015. Cloud Storage Market by Solutions. (August 2015). http://www.marketsandmarkets.com/Market-Reports/cloud-storage-market-902.html.

Xiaofeng Meng, Ying Chen, Jianliang Xu, and Jiaheng Lu. 2010. Benchmarking cloud-based data management systems. In *Proceedings of the 2nd International Workshop on Cloud Data Management in Cloud Systems (CloudDB'10)*. ACM Press

Microsoft. 2015. OneDrive. https://onedrive.live.com/.

OpenStack. 2011. OpenStack Swift. http://www.openstack.org/.

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)*.

S3Backer. 2015. S3Backer. Retrieved from https://code.google.com/p/s3backer/.

S3FS. 2015. S3FS. Retrieved from https://code.google.com/p/s3fs/.

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)*.

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 International Workshop on Quality of Service (IWQoS'12)*.

Zhe Wu, Curtis Yu, and Harsha V. Madhyastha. 2015. CosTLO: Cost-effective redundancy for lower latency variance on cloud storage services. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI'15)*. 543–557.

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 International IEEE Conference on Cloud Computing (CLOUD'11)*.

Yupu Zhang, Charis Dragga, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2014. ViewBox: Integrating local file systems with cloud storage services. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST'14)*. 119–132.

Received July 2016; revised February 2017; accepted March 2017